Thesis for the Degree of Doctor of Philosophy

Gate Level Description of Synchronous Hardware and Automatic Verification Based on Theorem Proving

Per Bjesse

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, May 2001

Gate Level Description of Synchronous Hardware and
Automatic Verification Based on Theorem Proving
Per Bjesse
ISBN 91-7291-018-6

Doktorsavhandlingar vid Chalmers Tekniska Högskola,
Ny serie nr 1702
ISSN 0346-718x

Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, 2001

Gate Level Description of Synchronous Hardware and
Automatic Verification Based on Theorem Proving
Per Bjesse
Computing Science
Chalmers University of Technology and Göteborg University

Abstract

Today's hardware development industry faces enormous problems. The primary reason for this is that the complexity of state-of-the-art hardware devices is growing faster than the capacity of the tools that are used to check that they are correct. This problematic situation is further aggravated by an increasing pressure to make the development time as short as possible. As a consequence, components under design are more likely to contain errors, while less time can be spent making sure that finished products are correct.

In this thesis, we contribute to improved hardware design methods in two ways.

First, we present Lava, a hardware description and verification platform that is embedded in the functional language Haskell. Lava uses the capabilities of the host language to express synchronous circuits in a mathematically precise way, and allows easy connection to external verification tools. Lava also uses the capabilities of Haskell to allow the designer to devise interconnection patterns, and to write parametrised circuit descriptions. We illustrate the power of Lava by describing and verifying hardware components for computing the Fast Fourier Transform (FFT).

Second, we present a number of techniques and case studies that demonstrate how automatic theorem proving can be used to prove correctness and find bugs in synchronous hardware. We show how verification can be done both at the level of complex arithmetic, and at the boolean level. In the case of the verification at the arithmetic level, we use Lava to construct special purpose proof strategies that interface with a first order logic theorem prover. In the case of the verification at the boolean level, we convert a number of standard finite state verification methods to use propositional logic theorem provers. The resulting converted methods are shown to give order of magnitude speedups compared to current state-of-the art verification techniques.

Keywords: hardware description, hardware verification, model checking, reachability analysis, symbolic trajectory evaluation, theorem proving, satisfiability, induction, signal processing, functional languages.

This thesis collects together five articles that are published or accepted for publication, which appear as Chapters 2, 3, 5, 6, and 7:

- *Lava: Hardware Design in Haskell*, written together with Koen Claessen, Mary Sheeran and Satnam Singh, published in 1998 [11]
- *Automatic Verification of Combinational and Pipelined FFT Circuits*, published in 1999 [25]
- *Symbolic Reachability Analysis Based on SAT-Solvers*, written together with Parosh Aziz Abdulla and Niklas Eén, published in 2000 [2]
- *SAT-based Verification without State Space Traversal*, written together with Koen Claessen, published in 2000 [10]
- *Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers*, written together with Tim Leonard and Abdel Mokkedem, accepted for publication in 2001 [12]

The thesis also contains a technical report, Chapter 4, and a journal paper submitted for publication, Chapter 8:

- *Symbolic Model Checking with Sets of States Represented as Formulas*, published in 1999 [9]
- *SAT-based Model Checking: A Tutorial and Overview*, written during 2000 together with Mary Sheeran and Gunnar Stålmarck [13]

The following outlines my participation on the articles in this thesis that have more than one author:

Lava: Hardware Design in Haskell, written together with Claessen, Sheeran, and Singh: I participated in discussions, did the FFT modelling and verification, and wrote Section 4 of the paper.

Symbolic Reachability Analysis Based on SAT-Solvers, written together with Abdulla and Eén: I participated in discussions, did some of the experimental work, and wrote Sections 1-3 and 6-8 of the paper.

SAT-based Verification without State Space Traversal, written together with Claessen: I participated in discussions, implemented the core algorithms, and wrote half the paper.

Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers, written together with Leonard and Mokkedem: I participated in discussions, implemented the SAT-based verification algorithms, and wrote the whole paper.

SAT-based Model Checking: A Tutorial and Overview, written together with Sheeran: I participated in discussions and wrote the whole paper with the exception of Section 3 and Section 4.3.

Contents

Chapter 1

Introduction

1.1 The Hardware Development Crisis

In 1965, Gordon Moore gave a talk where he predicted that the complexity of hardware devices would double every 18 months. Moore's prediction has turned out to be very accurate, and as a consequence, the current complexity of hardware is staggering.

The resulting constantly increasing pressure on designers is further aggravated by the fact that today's industry-standard hardware description languages are verbose and complex languages that do not have a well-defined semantics. This means that the interpretation of a given language differs significantly between tools from different vendors. The interpretation might even differ between different versions of the same tool; what worked two weeks ago may very well result in nonsense after an upgrade. The learning time to be able to produce usable designs with a language is also very long, and efficient use of a language relies heavily on intimate knowledge of the quirks of a particular tool suite.

Moreover, today's designers must not only cope with very complex designs using less than perfect languages; the development time for a given device must also be kept to an absolute minimum to reduce the time-to-market. This has led to a paradoxical situation, where the frequency of errors constantly goes up as larger designs have to be produced in shorter time, while less time can be spent on *validation*—making sure that what has been built indeed corresponds to the intended design.

The standard way of making sure that hardware is correct is to run a finished design, or a model of it, on a set of test vectors. This is a simple, natural, way of making sure that the correct design has been built. Unfortunately this validation method has a major drawback: Only extremely simple designs can be simulated exhaustively so that the full functionality of the chip is tested. To realise this, consider that a simple combinational gate with n inputs has to be tested on 2^n input patterns. This means that in order to check that an integer multiplier correctly multiplies two 32-bit words, we have to test it on 2^{64} input vectors. Assuming that we can test a million patterns a second, we would need more than half a million years to test the full functionality of the multiplier. Exhaustive testing of realistic combinational circuits is hence intractable, and the general case of testing sequential circuits is harder by far.

Unfortunately, failing behaviour even on a small number of vectors can be disastrous, as illustrated by the failure of Intel's Pentium chip to correctly divide a handful of floating point numbers. The cost of recalling the faulty chips was 475 million dollars.

Today's hardware development industry is thus in a bothersome situation, which needs to be addressed in at least two ways. First, it is vital that hardware description languages are improved so that they do not hinder the designer. The most urgent of the necessary improvements is to construct languages that have a mathematically precise meaning. Second, the support for verifying that what has been designed is correct must be improved significantly. It is important to find alternative ways of increasing the confidence in designs other than careful engineering together with simulation. The question is how this can be done.

1.2 Formal Methods

In this thesis, we will consider *formal methods* as a complement to today's industry-standard development methods.

1.2.1 Formal Methods—What are they?

A formal method ideally consists of three components: (1) A formal language; that is, a language where every well-formed statement has a mathematically defined meaning; (2) tools that help the user describe systems and requirements in the formal language; and (3) a proof system that allows the user to reason about statements in the formal language. In practice, not all of these need to be present.

The purpose of the three components of the formal method are as follows: The language, together with the tools, should let the designer write or translate a model of the system, and optionally also construct a specification. When this is done, the proof system should be used to do a system analysis. Hopefully, this process will allow the designer to predict whether the underlying system has unintended properties that will cost money, or put humans at risk.

There are two main approaches to formal methods: (1) specification oriented, and (2) verification oriented.

A *specification oriented* formal method provides a rich and mathematically precise language that is used to write down a specification for a given system. The process of writing down formal specifications can bring benefits to a project, as it forces the specifier to think through the system carefully in order to make a precise description. Furthermore, as the specification is written in a language that has a formally defined meaning, it is possible to reason about the specification. The purpose of this is to *validate* the specification—to make sure the specification matches the intention of the designer. This can for example be done by simulating the specification, by proving that the specification is consistent, or by deriving consequences of the specification that can be checked for their reasonableness.

A *verification oriented* method, in contrast, has the goal of making sure that an implementation conforms to some specification. In order to do this, the designer writes a specification of the system, and a model of an implementation at a lower level of abstraction. Verification tools are then used to try to establish conformance. This can be done in a number of ways, a few of which we will consider in Section 1.2.3. If necessary, the process can then be reiterated by writing a new specification at an even lower level and establishing conformance between the old implementation-level and the new lower level. If the verification is *post-hoc*, meaning that the system already has been built, the goal is to reach the level of the existing system model. On the other hand, if the system has not been built yet, the lowest level of specification can be used as a template for automatic or manual implementation.

It is important to realise that formal methods are concerned with *models* of systems. The analysis tools can only be applied to these models, not to the real systems themselves. This means, for example, that a proof that the model conforms to the specification implies precisely this, and nothing else. If the model does not adequately describe the real system, or if we have written a specification that is different from our intended specification, then the proof is worthless. If we model a circuit at the level of boolean gates, we can clearly not account for errors at the transistor level.

1.2.2 A Brief History of Formal Methods

Formal methods are intimately connected with logic, as logics are particular formal languages that are used to model and reason about problems. The first recorded thoughts about logic date back to Aristotle's time. Aristotle modelled situations as a collection of informal statements, and applied what is known as syllogistic reasoning to analyse the problem at hand.

The first modern proponent of formal methods was Leibnitz (1646-1716), who dreamt of constructing a language rich enough to model any kind of phenomena: the *characteristica universalis*. Any kind of question that a human could enquire about should be possible to formulate in this language. A given question should then be decided using a device Leibnitz called the *calculus ratiocinator*. In this way everyday questions would be decided in a scientific manner by the application of algorithmic techniques.

Leibnitz' dream was never appreciated by his contemporaries; it would take almost two hundred years before similar ideas resurfaced. However, at the end of the 19th century, many logicians started to think along the lines of Leibnitz. Foremost of these early pioneers was Gottlob Frege, who made what would be the first characterisation of a general purpose formal language rich enough to devise a theory of sets and elementary arithmetic [37]. That Frege was interested in this was not very surprising; his overall goal was to show that all mathematical reasoning could be reduced to logic. Unfortunately, in 1931 Kurt Gödel delivered a crushing blow to this hope by proving that no finite system of axioms and inference rules could be devised that would generate all the theorems of elementary arithmetic, and no non-theorems [39].

Gödel's result sparked a lot of research into the limits of automated procedures for deciding whether formal language statements were theorems or not. A whole sequence of similar results showed that the power of completely automated procedures was severely restricted. For example, in 1953 Rice proved that *every* non-trivial (in a precise, technical sense) property of programs is impossible to decide by automated analyses [80]. Much of the computation-oriented research during the 1950s and 1960s was hence centered around deriving "impossibility results", and few researchers were interested in pursuing Leibnitz' dream.

It was not until the late 1960s and early 1970s that researchers once again got seriously interested in applying formal techniques to reasoning and the research took off. It was about this time that programmable computers started to become interesting to industry, and many researchers were involved in creating different kinds of formal languages: programming languages. With this came a new upsurge of interest in rigour; many research languages were given mathematically defined semantics, and this allowed proofs about the languages, and about individual programs.

The newfound interest in rigour led to the establishment of many new research fields, and it was during this time the foundations for much of today's formal methods were laid, both in the domain of mathematical models of software and hardware systems, and the domain of computer aided reasoning about the models.

Some of the first attempts to reason formally about programs were undertaken by Floyd, Hoare, and Dijkstra [36, 52, 31]. These researchers were interested in correctness proofs for imperative programs, and developed methods based on program annotations with assertions about the program state. They then used informal reasoning in first order logic to prove that the state before the execution of a program fragment guaranteed the asserted state after the execution.

The techniques for imperative program verification were extended by Owicki and Gries to the more general case of parallel programs [73]. Totally different approaches were taken by Hoare and Milner, who developed special purpose calculi for specifying and reasoning about concurrent processes [53, 70]; Misra and Chandy, who invented the UNITY logic [71]; and Lamport, who invented the Temporal Logic of Actions [61].

While some researchers were focusing on program verification, others were developing powerful specification methods like Z, VDM, and Larch [91, 58, 44]. These methods were used successfully to specify and perform analysis on large software systems. Early success stories include IBM's use of VDM to define the programming language ALGOL 60 [19].

The efforts on program specification and verification also inspired research into the modelling and verification of hardware. Some researchers attempted to use extensions of the techniques devised for program verification [30, 88], while others developed new formalisms aimed specifically at hardware and hardware-like system. Early attempts to model and reason about circuits include Sheeran's work on μFP [83], Johnson's work on circuit derivation [57], Daeche and Hanna's work on Veritas [49], Wagner's work on transformational correctness proofs for circuits [98], and Milne's work on CIRCAL [69]. In the mid-eighties, hardware verification techniques were starting to be applied to systems of industrial relevance and size. Examples includes Hunt's modelling and verification of the FM 8501 microprocessor [55] and Cohns' correspondence proof between different levels of descriptions of the VIPER microprocessor [23].

1.2.3 Formal Hardware Verification Methods

Today's hardware verification methods can be divided into *interactive*, and *automatic* methods.

In interactive methods, the user guides the verification process. Often the specification language is rich enough to be undecidable; fully automatic verification is therefore not even attainable in theory.

A prime example of an interactive method is hardware verification by theorem proving in higher order logic [40]. In this approach, the model of the circuit, and the specification is expressed as statements in a very powerful logic. The goal of the process is to establish that the compound statement that expresses that the model implements the specification is a logical truth in the system. This is done by starting from axioms of the logic, and applying inference rules to generate new, logically true, statements. The nature of the inference rules guarantees that if the correctness statement is derivable by a finite chain of applications of rules, starting from axioms only, then the model must implement the specification.

Even between different interactive methods, the level of interactiveness can vary substantially. For example, certain higher order logic theorem provers, such as PVS [29], contains automatic decision procedures that can be applied to statements of particular forms. Such automated decision procedures exist for example for Presburger arithmetic [76] (arithmetic statements using addition but not multiplication). Other interactive theorem provers, such as Isabelle [75], provide much less automation; they mainly help the user with the bookkeeping by making sure that every application of an inference rule is valid, and that only axioms are used as starting points for proofs.

Automatic methods, in contrast, attempt to separate the user from the individual verification steps. This means that designers, whose area of expertise is the construction of the circuits themselves, can do their own verification without spending a long period of time learning about the theory underlying the method, much like designers do not need to fully understand how a hardware synthesiser works to use it. However, to continue the analogy with circuit synthesis, the user must still know something about the tool to get sufficiently good results.

In order to achieve full automation, it is necessary to restrict the checking to relatively weak properties. For example, full higher order logic will probably never be used successfully as a specification language aimed at automatic proofs, as it is too complex. There exists a clear tradeoff between the complexity of the modelling language and the properties to check, and the degree of automation that can be achieved. For example, due to Rice's theorem, if the modelling and specification language is strong enough to model programs and properties of programs, then there are many questions that cannot be decided automatically.

One of the most widely used automatic methods of verifying hardware is *model checking* [21, 79]. The aim of this verification method is to take a description of a hardware component as a finite state machine, and establish whether properties expressed in a weak logic hold by exhaustively searching through the state space of the model. Examples of properties that can be checked are "at all times, signal x is true", "if signal x is true, then signal y must be false after some finite period of time". The specification language is called a *temporal logic*, as it can express properties of systems that change over time. However, the properties that can be expressed are still so limited that the logic itself can be decided (there exists a procedure that always is able to tell whether a statement is a logical truth or not).

Independent of the verification method itself, circuits can be modelled at different levels. The level of electron flow is beyond the reach of today's verification technology, as the mathematics that is involved is too complicated. However, it is important to realise that this also is true on the design side of hardware development. When individual components are designed from transistors, idealised models are used for the transistors in order to facilitate speedy analysis.

The lowest level of modelling that is routinely processed today is transistor level descriptions. Models at this level are still often simplified by describing the transistors as devices that propagate either of the values 0, 1, or X (unknown) according to a "truth table", without considering any other possibilities. Verification at this level may consist in establishing that simple temporal properties hold.

The next level up in the hierarchy is the level of gate level designs. Here the individual building blocks are logical gates such as and-gates and or-gates. At this level, verification may consist of establishing that a transistor level implementation implements a gate network, or checking that a temporal specification holds.

From the gate level up, there exists a multitude of levels that circuits can be modelled at. For example, at the level of arithmetic, an addition of two numbers from an infinite domain is a monolithic operation. Just as Gödel's theorem predicts, this is the level where automation starts to become troublesome, and users often turn to methods such as interactive theorem proving.

1.3 Aim of the Thesis

In this thesis, we contribute to better methods for designing hardware in two ways.

First, we present a hardware description and verification platform, Lava, embedded in the functional language Haskell. This platform supports mathematically precise descriptions of synchronous hardware, and allows the user to interface to verification tools through a programmable interface. The thesis of this work is that functional languages offer a clean way of describing synchronous hardware, and that modern programming language features are useful tools also for hardware designers. We also show how useful it is that Lava is a programmable verification platform, which allows us to devise special purpose verification strategies for particular families of hardware components.

Second, we focus on how automatic theorem proving can be used as a tool for the verification of sequential hardware at different levels on abstraction. The major thesis of this part of the work is that automatic theorem proving can be used as an efficient vehicle for proving correctness and finding bugs in hardware. We illustrate this by presenting case studies and new techniques that give order of magnitude speedups compared to state-of-the-art automatic verification methods.

1.4 Synopsis

Paper A

The first paper in this thesis is called "Lava: Hardware design in Haskell". The paper appeared in the International Conference on Functional Programming 1998, and is joint work with Koen Claessen, Mary Sheeran, and Satnam Singh.

The paper contains an overview of the Lava hardware description and verification platform, which is specially designed for describing, and analysing synchronous hardware at the gate level.

Key elements of the Lava approach is a clean way of describing circuits that allows the designer to easily construct parametrised circuits, and define interconnection patterns—meta-circuits that given a number of other circuits generate a new circuit by plugging the building blocks together. A crucial feature of

Lava is the possibility to interpret hardware descriptions in multiple ways. For example, the same description can be simulated by interpreting it in one way, and verified by interpreting it in another way that produces formulas that can be sent off to external theorem provers.

The presentation includes a case study: The specification and verification of a parametrised combinational circuit for performing the Fast Fourier Transform (FFT). The FFT is a particular weighted sum of complex numbers that for instance can be used to do quick multiplication of polynomials, and to extract frequency information from sampled signals. The fact that the circuit operates on the level of complex numbers means that automatic verification is troublesome, as many standard techniques are unable to cope with arithmetic.

The case study is joint work with Ericsson Cadlab, who wanted to study how Lava could be used to model and verify the kinds of components they were interested in.

Paper B

Paper B is called "Automatic Verification of Combinational and Pipelined FFTs", and it appeared in the International Conference on Computer Aided Verification 1999.

In this paper we continue the work on describing and verifying FFT circuitry, by using Lava to implement a sequential, pipelined, version of the combinational FFT we proved correct in paper A.

In order to verify the pipelined implementation, we devise a special purpose proof technique for establishing correspondence between a combinational and a pipelined circuit implementation. This proof technique relies on symbolic simulation and induction, in addition to the theorem proving techniques we applied in order to verify the combinational FFT component.

We take advantage of the ease with which Lava can be extended with custom analyses, and implement the proof technique as an analysis parametrised by a sequential and a combinational circuit.

Paper C

Paper C is called "Symbolic Model Checking with Sets of States Represented as Formulas". This paper is a revised version of technical report CS-1999-103, Chalmers University of Technology.

Here, inspired by the work we did on verifying sequential circuits by theorem proving in Paper B, we present an idea for how standard symbolic model checking can be converted to use propositional logic theorem proving rather than the more standard choice of Binary Decision Diagrams (BDDs).

The paper presents a conversion of the standard symbolic model checking algorithms to use formulas as a symbolic representation for the system that is analysed. We remove boolean quantifiers that arise during state space traversal by applying a naive unfolding rule. In order to make sure that the representations do not grow too much, we suggest an algorithm for minimising formulas that uses Stålmarck's method of propositional proof to detect equivalent subformulas.

We conclude the paper by discussing how the idea to use propositional formulas as a representation, and theorem proving to implement operations on the representation can be extended to stronger logics, and the possible benefits thereof.

Paper D

This paper is called "Symbolic Reachability Analysis based on SAT-solvers". It appeared in the International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2000, and is joint work with Parosh Abdulla and Niklas Eén.

In this paper we present FixIt, a practical implementation of the idea of converting standard model checking algorithms to use formulas and propositional logic theorem proving. Here, we restrict the presentation to the case of symbolic reachability analysis (a single operator from the temporal logic that was covered in Paper C), but our implementation handles the full logic.

Key elements of our approach are to use a special purpose representation for formulas that does simple reductions automatically, and to use some extra rules in the quantifier translator instead of the single rule we suggested in Paper C.

We illustrate that the resulting, fairly naive, model checker can still cope with some examples that are very hard for two mature BDD-based model checkers.

Paper E

This paper, "SAT-based Verification without State Space Traversal", is joint work with Koen Claessen. It appeared in the International Conference on Formal Methods in Computer Aided Design 2000.

The translation of boolean quantifiers necessary for state space traversal in the algorithms we presented in Paper D works very poorly for certain examples. Algorithms that avoid the boolean quantification, but still use propositional logic theorem proving, are therefore very interesting.

In this paper we convert Van Eijk's algorithm, a previously presented method for doing sequential equivalence checking without state space traversal, to use Stålmarck's algorithm rather than BDDs. We also strengthen the algorithm so that it can find more information, and make it complete. Furthermore, by examining the connection between the algorithm and overapproximative symbolic reachability analysis, we devise an analogous dual algorithm and a mutual improvement algorithm.

We illustrate the power of the resulting analyses on some benchmarks, and compare results with two related methods based on propositional logic theorem proving, and a mature BDD-based model checker.

Paper F

Paper F, "Finding Bugs in an Alpha Microprocessor using Satisfiability Solvers", is joint work with Tim Leonard and Abdel Mokkedem. It is accepted for publication at the International Conference on Computer Aided Verification 2001.

Here we present the results of using a combination of two model checking techniques based on satisfiability solvers to find bugs in the memory subsystem of a next-generation microprocessor.

The first of these techniques is called bounded model checking. We compare the performance of bounded model checking to BDD-based model checking, and demonstrate that it can be used to reduce the runtime necessary for finding certain bugs in real designs from days to minutes.

The second technique is symbolic trajectory evaluation based on SAT-solving. This analysis is a version of model checking that consists of a blend of abstract interpretation and symbolic simulation. We demonstrate that SAT-based symbolic trajectory evaluation can be used to find very complex bugs, in terms of the length of minimum failure traces, using negligible runtimes. The tradeoff compared to bounded model checking is that we have to spend more time developing specifications.

We conclude the paper by presenting some advice on using a combination of the two model checking techniques for industrial-strength verification.

Paper G

This paper is called "Symbolic Model Checking based on SAT-solvers: An overview and Tutorial". It is joint work with Mary Sheeran and Gunnar Stålmarck, and is submitted for publication.

Here we give an overview of the field of model checking based on propositional logic theorem proving by presenting three different approaches that have appeared in the literature.

Our presentation is given in a tutorial style, and we put emphasis on demonstrating how the methods are related. We present some techniques that are used to implement the analyses in the research model checker FixIt we presented in Paper D, and some conclusions we have come to while working with the three analyses. Finally, we outline what we think might be interesting directions for further research into the area.

Chapter 2

Lava: Hardware Design in Haskell

Lava is a tool to assist circuit designers in specifying, designing, verifying and implementing hardware. It is a collection of Haskell modules. The system design exploits functional programming language features, such as monads and type classes, to provide multiple interpretations of circuit descriptions. These interpretations implement standard circuit analyses such as simulation, formal verification and the generation of code for the production of real circuits.

Lava also uses polymorphism and higher order functions to provide more abstract and general descriptions than are possible in traditional hardware description languages. Two Fast Fourier Transform circuit examples illustrate this.

This chapter was written together with Koen Claessen, Mary Sheeran and Satnam Singh. It was published at the International Conference on Functional Programming 1998 [11].

2.1 Introduction

The productivity of hardware designers has increased dramatically over the last 20 years, almost keeping pace with the phenomenal development in chip technology. The key to this increase in productivity has been a steady climb up through levels of abstraction. In the late seventies, designers sat with 'coloured rectangles' and laid out individual transistors. Then came the move through gate-level to register-transfer level descriptions, and the important step from schematic capture to the use of programming languages to describe circuits. Standard Hardware Description Languages like VHDL and Verilog have revolutionised hardware design.

However, problems remain. VHDL was designed as a simulation language, but now subsets of it are used as input to many kinds of tools, from synthesis engines to equivalence checkers. VHDL is poorly suited to some tasks, for example formal verification.

Ideally, we would like to be able to describe hardware at a variety of levels of abstraction, and to analyse circuit descriptions in many different ways. The analyses (or *interpretations*) that we consider to be essential are simulation (checking the behaviour of a circuit by giving it some inputs and studying the resulting outputs), verification (proving properties of the circuit), and the generation of code that allows a physical circuit to be produced. We want to be able to perform all of these tasks on one and the same circuit description.

The temptation to go away and design yet another hardware description language is strong, but we have resisted it. Instead, we would like to see how far we can get using the functional programming language Haskell. We call our design system Lava. The idea of using a functional hardware description langauge is, of course, not new, and the work described here builds on our earlier work on μFP [84] and Ruby [60], and on the use of non-standard interpretation in circuit analysis [90].

What is new about Lava is that we have built a complete system in which real circuits can be described, verified, and implemented. An earlier version of the system was used to generate filters and Bezier curve drawing circuits for implementation in a Field Programmable Gate Array based PostScript accelerator. Using the current system, very large combinational multipliers have been verified [85]. The largest formula produced so far from a circuit description had almost a million connectives. The system is constructed in a way that systematically makes use of important features of Haskell: monads, type classes, polymorphism and higher order functions.

We use ideas from Ruby, for example the use of combinators to build circuits, but in using Haskell, we gain access to a fully fledged programming language, with a rich type system and higher order functions. Having higher order functions available has greatly eased circuit description in real circuit examples. Circuits themselves still correspond to first order functions, but we use higher order functions to construct circuit descriptions. Although we knew in theory that it is a good idea to have circuits as first class objects, we were surprised by how useful it is in practice. For example, higher order functions make it very easy to describe circuits containing look-up-tables. VHDL descriptions of such circuits tend to be long and hard to read, precisely because of the absence of suitable combinators. And even in Ruby, it is hard to deal with circuits that have a regular structure but components that vary according to their position in the structure.

Although we have moved from a relational to a functional programming language, we can retain as much of the generality of relations as we need, because the logical interpretation described later produces formulas that are relational, in that they do not distinguish between input and output. What we have lost, in moving away from Ruby, is machine support for high level design [82].

After choosing to use Haskell for hardware description, we again had two options: to make a Haskell variant and write specialised tools (compilers, synthesis engines and so on) to process it, or to make use of existing Haskell compilers by embedding a hardware description language in Haskell. Launchbury and his group are investigating the first option [24]. We chose the second.

2.2 Overview of the System

This section presents the types and abstractions used in the Lava system.

2.2.1 Monads

Dealing with an embedded language in a functional language requires a significant amount of information plumbing. A good way to hide this is to use *monads* [97]. Defining a monad means defining the language's features; a monadic expression is a program in the embedded language. Moreover, Haskell provides syntactic support and general combinator libraries for monads.

Let us take a look at a small example, and see how we can define a *half adder* circuit (figure 2.1):

```
halfAdd :: Circuit m => (Bit, Bit) -> m (Bit, Bit)
halfAdd (a, b) =
  do carry <- and2 (a, b)
     sum   <- xor2 (a, b)
     return (carry, sum)
```

This circuit has two input wires (bits) and two output wires. By convention, wires are grouped together so that a circuit always has one input value, and one output value. The `halfAdd` circuit consists of an `and` gate and an `xor` gate.

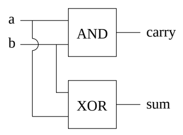

Figure 2.1: A half adder circuit

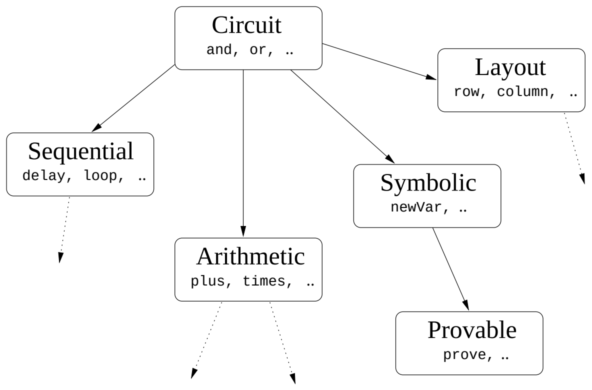

Figure 2.2: Type Class structure for Interpretations

Note that the type of a circuit description contains a type variable `m`, indicating that it is overloaded in the underlying monad. This means that we can later decide how to interpret the description by choosing an appropriate implementation of `m`. The same description can be interpreted in many ways, giving various different semantics to the embedded language. Examples of such *interpretations* are simulation (where we run the circuit on specific values), and the symbolic evaluation that is used to produce VHDL code.

2.2.2 Type Classes

Some circuit operations are meaningful only to certain interpretations; Lava is therefore structured with type classes (see figure 2.2). For example, a higher-level abstract circuit can deal with arithmetic operators, such as `plus` and `times`, where a physical circuit has no notion of numbers at all. We can point out groups of operations, which are supported by some interpretations but not by others, thus forming a hierarchy of classes.

The base class of the hierarchy is called `Circuit`. To be a `Circuit`, means to be a `Monad`, and to support basic operations like `and` and `or`.

```
class Monad m => Circuit m where
  and2, or2 :: (Bit, Bit) -> m Bit
  ...
```

Subclasses of `Circuit` are for example the `Arithmetic` class, for higher-level interpretations supporting numbers, and the `Sequential` class, for interpretations containing delay operations.

```
class Circuit m => Arithmetic m where
  plus, times :: (NumSig, NumSig) -> m NumSig
  ...

class Circuit m => Sequential m where
  delay :: Bit  -> Bit    -> m Bit
  loop  :: (Bit -> m Bit) -> m Bit
  ...
```

A circuit description will typically be constrained in the type to indicate what interpretations are allowed to run the description. The following circuit can only be run by interpretations supporting arithmetic:

```
square :: Arithmetic m => NumSig -> m NumSig
square x = times (x, x)
```

The architecture of the system makes it easy for the user to add new classes of operations to the hierarchy, and new interpretations that give semantics to them (see section 2.4.1).

2.2.3 Primitive Data Types

We use the datatype `Bit` to represent a bit. For now, this datatype can be regarded as just a boolean value, but we will slightly extend the datatype later (see section 2.3.2). We provide two constant values of this type:

```
data Bit = Bool Bool | ...

low, high :: Bit
low  = Bool False
high = Bool True
```

To describe circuits at a higher level, we add another primitive datatype, a `NumSig`, which represents an abstract wire through which numbers (integers) can flow. The `NumSig` wires will of course never appear in a physical circuit as an interpretation needs to be in the type class `Arithmetic` to handle this datatype.

```
data NumSig = Int Int | ...

int :: Int -> NumSig
int n = Int n
```

It is possible for the user to add other datatypes to Lava (see section 2.4.1).

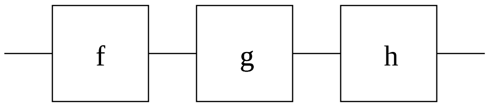

Figure 2.3: `compose` [f,g,h]

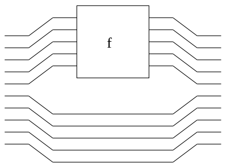

Figure 2.4: `one` f

2.2.4 Combinators

Common circuit patterns are captured using combinators which allow the designer to describe regular circuits compactly and in a way that makes the patterns explicit. This section describes some simple combinators that will be useful later.

The composition combinator `>->` passes the output of the first circuit as input to the second circuit. We also provide a version that works on lists (figure 2.3).

```
(>->) :: Circuit m
      => (a -> m b) -> (b -> m c) -> (a -> m c)

compose :: Circuit m => [a -> m a] -> (a -> m a)
compose = foldr (>->) return
```

The combinators `one` and `two` build circuits operating on $2n$-lists from circuits operating on n-lists. While `one f` applies the circuit `f` to one half of the wires and leaves the rest untouched, `two f` maps it to both halves (see figure 2.4 and 2.5).

```
one :: Circuit m
     => ([a] -> m [a]) -> ([a] -> m [a])

two :: Circuit m
     => ([a] -> m [b]) -> ([a] -> m [b])
```

Repeated application of a function is captured by `raised`: The expression `raised 3 two f` results in 8 copies of the `f` circuit, each applied to one eighth of the input wires.

```
raised :: Int -> (a -> a) -> (a -> a)
raised n f = (!! n) . iterate f
```

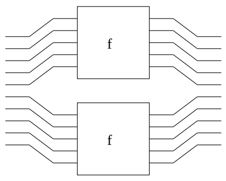

Figure 2.5: `two` f

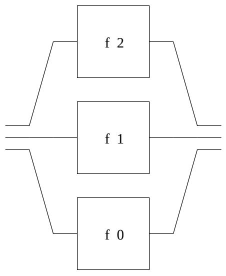

Figure 2.6: `decmap` 3 f

The circuit `decmap n f` processes an n-list of inputs by applying `f (n-1)`, `f (n-2)`, .. , `f 0` consecutively to each element (see figure 2.6).

```
decmap :: Circuit m
       => Int -> (Int -> a -> m b) -> ([a] -> m [b])
decmap n f = zipWithM f [n-1,n-2 .. 0]
```

The user can define new combinators as needed.

2.3 Interpretations

In this section, we present some interpretations dealing with concrete circuit functionality. *Standard* interpretations calculate outputs of a circuit, given input values. *Symbolic* interpretations connect Lava to external tools, by generating suitable circuit descriptions.

2.3.1 Standard Interpretation

The standard interpretation we present here is one that can only deal with *combinational* circuits, which have no notion of time or internal state. In this case, it suffices to use the identity monad since no side effects are needed.

```
data Std a = Std a
```

```
simulate :: Std a -> a
simulate (Std a) = a

instance Monad Std where ...
```

The resulting `Std` interpretation is integrated into the system by specifying the `Circuit` operations.

```
instance Circuit Std where
  and2 (Bool x, Bool y) = return (Bool (x && y))
  ...

instance Arithmetic Std where
  plus (Int x, Int y) = return (Int (x + y))
  ...
```

We can now simulate the example of section 2.2.1.

```
Hugs> simulate (halfAdd (high, high))
(high, low)
```

To deal with time and state, we can *lift* a combinational circuit interpretation into a sequential one. How this is done is beyond the scope of this paper.

2.3.2 Symbolic Interpretation

Lava provides connection to external tools through the symbolic interpretations. These generate descriptions of circuits, rather than computing outputs. External tools process these descriptions, and in turn give feedback to the Lava system. The tools we focus on in this paper are theorem provers. We briefly sketch other possibilities in section 2.3.5.

A circuit description is symbolically evaluated by providing abstract variables as input. The result of running the circuit is a symbolic expression representing the circuit. To implement this idea, we need some extra machinery. First of all, the signal datatypes are modified by adding a constructor for a variable, since a signal in this context can be both a value and a variable:

```
type Var = String

data Bit              data NumSig
  = Bool Bool           = Int Int
  | BitVar Var          | NumVar Var
```

It is important to keep the constructors of these datatypes abstract as the `Std` interpretation is unable to handle variables. By introducing the class `Symbolic`, we ensure that functions for variable creation are only available in interpretations which recognise variables.

```
class Circuit m => Symbolic m where
  newBitVar :: m Bit
  newNumVar :: m NumSig
```

When a circuit operation is applied to symbolic inputs, we create a fresh variable, and remember internally in the monad how this variable is related to the parameters of the operation.

An implementation for this interpretation is a state monad in an (infinite) list of unique variables, and a writer monad in a list of assertions. The type `Expression` is left abstract here.

```
type Sym a = [Var] -> (a, [Var], [Assertion])

data Assertion  = Var := Expression
type Expression = ...
```

The instance declaration for `Circuit Sym` is:

```
instance Circuit Sym where
  and2 (a, b) =
    do v <- newSymbol
       addAssertion (v := And [a,b])
       return (BitVar v)
  ...
```

When this interpretation is run on the half adder from section 2.2.1, the following internal assertion list is generated:

```
[ "b3" := And [ BitVar "b1", BitVar "b2" ]
, "b4" := Xor [ BitVar "b1", BitVar "b2" ]
]
```

The inputs to the circuit are called `"b1"` and `"b2"`.

2.3.3 Using a Symbolic Circuit

How can we now prove properties of circuits? We need to be able to formulate the circuit properties we want to verify. To do this, we create an *abstract* circuit that contains both the circuit and the property we want to prove.

To show a full adder with its leftmost bit set to False equivalent to a half adder, we write the question:

```
type Form = Bit

question :: Symbolic m => m Form
question =
    do a <- newBitVar  -- free variables
       b <- newBitVar

       out1 <- halfAdd (a, b)
       out2 <- fullAdd (low, a, b)

       equals (out1, out2)
```

Two fresh variables `a` and `b` are given as inputs to both the half adder and the restricted full adder. The resulting formula (of type `Form`) is true if the outputs of these circuits are the same. The type `Form` is the same as `Bit`, so that we can use the logical operators (`and2`, `or2`, etc.) on both types.

The function `question` is polymorphic in the underlying interpretation; any symbolic interpretation is applicable. Here, we shall instantiate `m` with `Sym`.

2.3.4 Verification

The `Sym` interpretation is not very interesting on its own; it needs to be connected to the outside world in some way. The function `verify` takes a description of a question (which is of type `m Form`) and generates a file containing a (possibly very large) logical formula. This file is then processed by one of the automatic theorem provers that is connected to Lava by means of the `IO` monad.

```
  verify :: Sym Form -> IO ProofResult

  data ProofResult
    = Valid
    | Indeterminate
    | Falsifiable Model
```

The result from the theorem prover interaction has type `ProofResult` and indicates whether the desired formula was valid or not. If a countermodel (a valuation making the formulas false) can be found, it is also returned.

Using Hugs, the user of Lava can run proofs from inside the interpreter.

```
  Hugs> verify question >>= print
  Valid
```

This invocation generates input for a theorem prover, containing the variable definitions and the question, separated by an implication arrow:

```
  AND( b3 <-> b1 & b2,       b4 <-> (b1 #! b2)
     , b5 <-> FALSE & b1,   b6 <-> (FALSE #! b1)
     , b7 <-> b6 & b2,       b8 <-> (b6 #! b2)
     , b9 <-> b5 # b7,       b10 <-> (b3 <-> b9)
```

```
, b11 <-> (b4 <-> b8), b12 <-> b10 & b11
) -> b12
```

Currently Lava interfaces to the propositional tautology checker Prover [92] and the first order logic theorem provers Otter [66] and Gandalf [95].

2.3.5 Other Interpretations

Using the same idea, we can generate input for other tools as well. An interesting target format is VHDL, which is one of the standard hardware description languages used in industry. There are many tools that can process VHDL, for purposes such as synthesis and efficient simulation.

Running the `Vhdl` interpretation on the half adder circuit (section 2.2.1) produces structural VHDL:

```
-- Automatically generated by Lava --
library circuit; use circuit.all;
entity halfadd is
  port ( b1, b2 : in std_logic;
         b3, b4 : out std_logic );
end halfadd;

library circuit; use circuit.all;
architecture structural of halfadd is
begin
  comp1 : and2 port map (b3, b1, b2);
  comp2 : xor2 port map (b4, b1, b2);
end structural;
```

An extended form of symbolic evaluation generates *layout* information. This is done by not only keeping track of how the components of a circuit are functionally composed, but also how they can be laid out on a gate array. `A >-> B` in this interpretation also indicates that `A` should be laid out to the left of `B`. Similarly, `row 5 fa` makes 5 full adders and lays them out horizontally with left to right data-flow.

The layout interpretation can generate VHDL and EDIF (another standard format) containing layout attributes that give the location of each primitive component.

Combining layout and behaviour in this way allows us to give economical and elegant descriptions of circuits, which in VHDL would require the user to attach complicated arithmetic expressions to instances.

2.4 An Example: FFT

This section illustrates how Lava is extended for signal-processing applications by the introduction of a complex number datatype and new combinators that allow two FFT circuits to be described.

The work presented here builds on previous work on deriving the FFT within Ruby [59] and specifying signal processing software in Haskell [8].

2.4.1 Complex Numbers

Two flavours of complex numbers are needed for simulation and verification: concrete values and variables representing complex numbers. The implementation datatype `CmplxSig` reflects this:

```
data CmplxSig
  = Abstract NumSig
  | Concrete (Complex Double)
```

A complex datatype has to support operations like addition and multiplication. The FFT circuits also need *twiddle factors*, constants computed by `w` (see section 2.4.2). The appropriate operations are grouped together into a class.

```
class Arithmetic m => CmplxArithmetic m where
  cplus  :: (CmplxSig, CmplxSig) -> m CmplxSig
  ctimes :: (CmplxSig, CmplxSig) -> m CmplxSig
  ...
  w      :: (Int,Int) -> m CmplxSig

  cplus  = clift plus  (+)
  ctimes = clift times (*)
  ...

instance CmplxArithmetic Std where ...
instance CmplxArithmetic Sym where ...
```

To extend the existing interpretations with the complex datatype, we must write appropriate instance implementations. In this case it is simple, as the complex arithmetic operations can be implemented by lifting the existing arithmetic operations on symbolic `NumSig` variables and concrete `Complex Double` values. The twiddle factors have different meanings for different interpretations: the `Std` interpretation will get constant complex values, while `Sym` expects symbolic values.

2.4.2 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) computes a sequence of complex numbers X, given an initial sequence x:

$$X(k) = \sum_{n=0}^{N-1} x(n) \times W_N^{kn}, \qquad k \in \{0 \dots N-1\}$$

where the constant W_N is defined as $e^{-j2\pi/N}$.

Each signal in the transformed sequence $X(k)$ depends on every input signal $x(n)$; the DFT operation is therefore expensive to implement directly.

The Fast Fourier Transforms (FFTs) are efficient algorithms for computing the DFT that exploit symmetries in the *twiddle factors* W_N^k. The laws that state these symmetries are:

$$\begin{aligned} W_N^0 &= 1 \\ W_N^N &= 1 \\ W_n^k \times W_n^m &= W_n^{k+m} \\ W_n^k &= W_{2n}^{2k}, \qquad (n, k \leq N) \end{aligned}$$

We will later use the fact that W_4^1 equals $-j$.

These laws, together with a restriction of sequence length (for example to powers of two), simplify the computations. An FFT implementation has fewer gates than the original direct DFT implementation, which reduces circuit area and power consumption. FFTs are key building blocks in most signal processing applications.

We discuss the description of circuits for two different FFT algorithms: the Radix-2 FFT and the Radix-2^2 FFT [51].

2.4.3 Two FFT Circuits

The *decimation in time* Radix-2 FFT is a standard algorithm, which operates on input sequences of which the length is a power of two [77]. This restriction makes it possible to divide the input into smaller sequences by repeated halving until sequences of length two are reached. A DFT of length two can be computed by a simple *butterfly* circuit. Then, at each stage, the smaller sequences are combined to form bigger transformed sequences until the complete DFT has been produced.

The Radix-2 FFT algorithm can be mapped onto a combinational network as in figure 2.7, which shows a size 16 implementation. In this diagram, digits and twiddle factors on a wire indicate constant multiplication and the merging of two arrows means addition. The bounding boxes contain two FFTs of size 8.

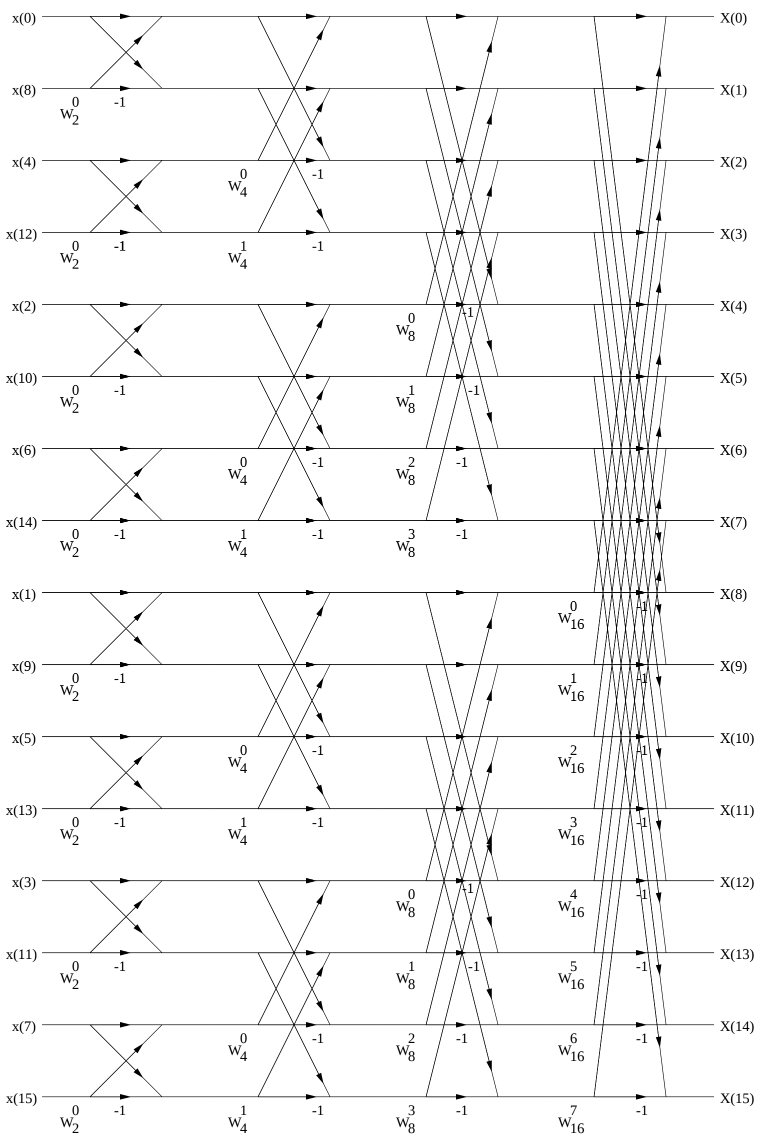

Figure 2.7: A size 16 Radix 2 FFT network

A less well-known algorithm for computation of the DFT is the *decimation in frequency* Radix-2^2 FFT, which assumes that the input length N is a power of four.

The corresponding circuit implementation (in figure 2.8) is also very regular and might be mistaken for a reversed Radix-2 circuit at a passing glance. However, it differs substantially in that *two* different butterfly networks are used in each stage, the twiddle factor multiplications are modified, and $-j$ multiplication stages have been inserted.

2.4.4 Components

We need three main components to implement FFT circuits. The first is a *butterfly circuit*, which takes two inputs x_1 and x_2 to two outputs $x_1 + x_2$ and $x_1 - x_2$ (see figure 2.9). It is the heart of FFT implementations since it computes the 2-point DFT. Systems of such components will be applied to the in-signals in many stages (figures 2.7 and 2.8).

The FFT butterfly stages are constructed by riffling together two halves of a sequence of length k, processing them by a column of $k/2$ butterfly circuits, and unriffling the result (see figure 2.10). Here `riffle` is the shuffle of a card sharp who perfectly interleaves the cards of two half decks.

```
bfly :: CmplxArithmetic m
     => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
  do o1 <- csubtract (i1, i2)
     o2 <- cplus (i1, i2)
     return [o1, o2]

bflys :: CmplxArithmetic m
      => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
  riffle >-> raised n two bfly >-> unriffle
```

Another important component of an FFT algorithm is multiplication by a complex constant, which can be implemented using a primitive component called a twiddle factor multiplier. This circuit maps a single complex input x to $x \times W_N^k$ for some N and k. The circuit `w n k` computes W_N^k.

```
wMult :: CmplxArithmetic m
      => Int -> Int -> CmplxSig -> m CmplxSig
wMult n k a =
  do twd <- w (n, k)
     ctimes (twd, a)
```

The multiplication of complete buses with $-j$ is defined as follows, using the fact that W_4^1 equals $-j$.

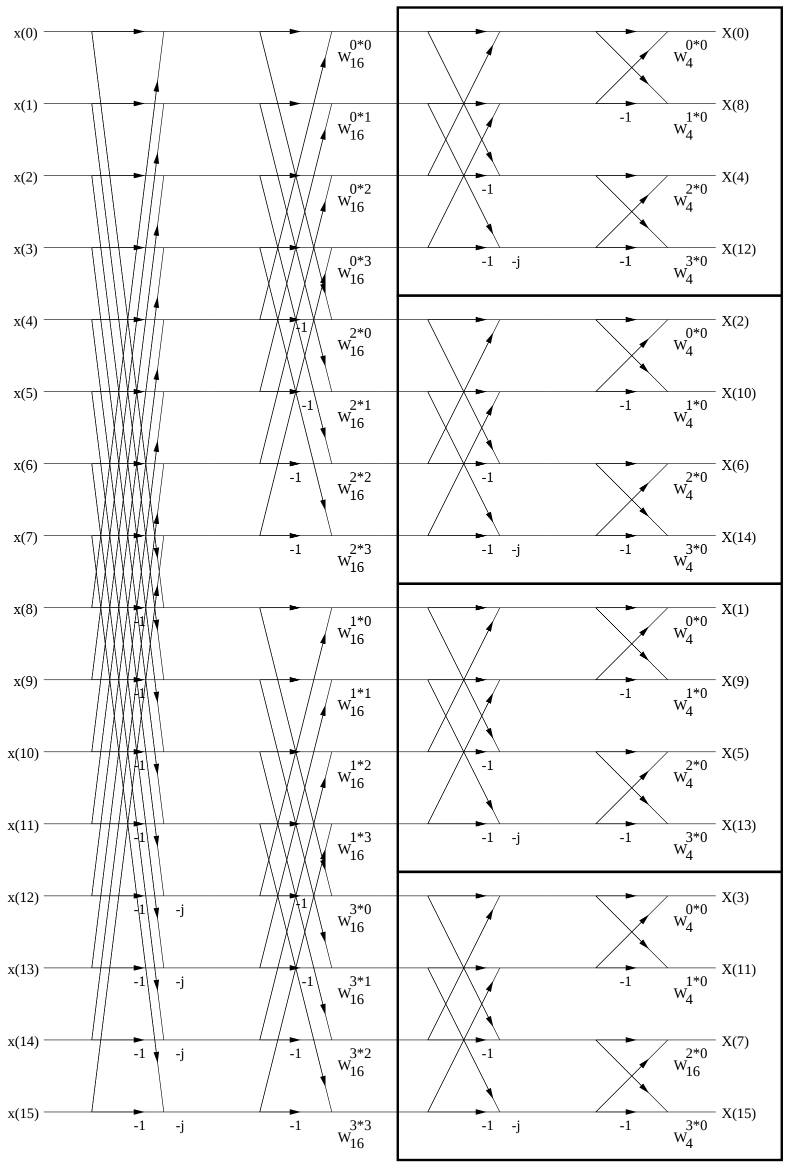

Figure 2.8: A size 16 Radix-2^2 FFT network

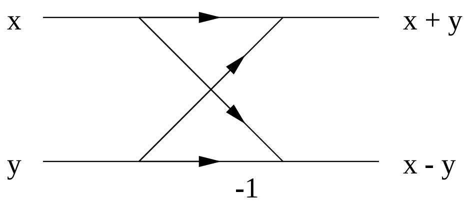

Figure 2.9: A butterfly

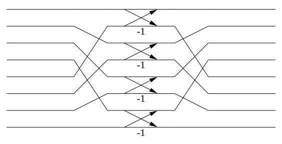

Figure 2.10: A butterfly stage of size 8 expressed with riffling

```
minusJ :: CmplxArithmetic m
       => [CmplxSig] -> m [CmplxSig]
minusJ = mapM (wMult 4 1)
```

Another useful component is the *bit reversal permutation*, used in the first or last stage of the FFT circuits. A new wire position is the reversed binary representation of the old position [77]. The permutation can be expressed using `riffle`:

```
bitRev :: Monad m => Int -> [a] -> m [a]
bitRev n =
  compose [ raised (n-i) two riffle
          | i <- [1..n]
          ]
```

Note that these components are not shown in the diagrams; either the in-data is permuted from the start, or the out-sequence needs to be rearranged.

2.4.5 The Circuit Descriptions in Lava

Inspired by the circuit diagrams we describe the two FFT circuits in Lava using higher-order combinators.

We begin by defining the type of an FFT parameterised by the interpretation monad `m`. A circuit description takes the exponent of the size of the circuit, and the list of inputs, and returns the outputs.

```
type Fft m = Int -> [CmplxSig] -> m [CmplxSig]
```

The Radix-2 FFT is a bit reversal composed with the different stages.

```
radix2 :: CmplxArithmetic m => Fft m
radix2 n =
  bitRev n >-> compose [ stage i | i <- [1..n] ]
 where
  stage i = raised (n-i) two
              $ twid i
             >-> bflys (i-1)

  twid  i = one (decmap (2^(i-1)) (wMult (2^i)))
```

The Radix-2^2 FFT is the sequence of stages composed with the final bit reversal.

```
radix22 :: CmplxArithmetic m => Fft m
radix22 m =
  compose [ stage i | i <- [m,m-1..1] ]
    >-> bitRev (2*m)
 where
  stage i = raised (m-i) (two.two)
              $ bflys (2*i-1)
             >-> one (one minusJ)
             >-> two (bflys (2*i-2))
             >-> twid i

  twid  i = column
             [ decmap (4^(i-1))
                         (wMult (4^i) . (wt *))
             | wt <- [3,1,2,0]
             ]
```

The corresponding VHDL descriptions would be several times longer.

2.4.6 Running Interpretations

We can now run some interpretations on our FFT circuits. Simulation is possible in the standard interpretation, if we provide an exponent and specific inputs to the circuit.

```
input :: [CmplxSig]
input = map cmplx [1:+4,2:+(-2),3:+2,1:+2]

Hugs> simulate (radix2 2 input)
[1.0:+6.0,(-1.0):+(-6.0),(-3.0):+2.0,7.0:+6.0]
```

The symbolic interpretation can be applied to verify that two circuit instances are equivalent, using the first order theorem prover Otter [66]. We create an abstract circuit stating the equivalence:

```
fftSame :: (Symbolic m, CmplxArithmetic m)
        => Int -> m Form
fftSame n =
  do inp <- newCmplxVector (4^n)

     out1 <- radix2 (n*2) inp
     out2 <- radix22 n inp

     equals (out1, out2)
```

The `newCmplxVector` function generates a list of complex symbolic variables. After applying both of the circuits to these inputs, we ask if the outputs are the same.

Before we can verify this equation, we have to add some knowledge to Otter: laws about complex arithmetic, and in particular the laws about twiddle factors. This information is added in the form of *theories*, which are defined by the user in Lava, and given to the prover as a proof option. Otter now shows circuit equivalence for size 4 FFTs (we have proven circuits of size 16 and 64 equivalent as well).

```
options :: [ProofOptions]
options = [ Prover otter
          , Theory arithmetic
          , Theory (twiddle 4)
          ]

Hugs> verify' options (fftSame 1) >>= print
Valid
```

Figure 2.11 shows the formula that is generated as input to Otter (notice the arithmetic and twiddle factor theory).

2.4.7 Related Work on FFT Description and Verification

The equivalence of a Radix-2 FFT algorithm and the DFT has been shown using ACL2, a descendant of the Boyer-Moore theorem prover [38]. Our approach in the example is slightly different in that we want to show automatically generated logical descriptions of *circuits* of a fixed size equivalent, rather than proving mathematical theorems about the *algorithms*. The verifications are similar however, in that both methods use relationships between abstract twiddle factors.

2.5 Related Work

In this section, we discuss related work on the use of functional languages for hardware description and analysis.

The work described here has its basis in our earlier work on μFP, an extension of Backus' FP language to synchronous streams, designed particularly for describing and reasoning about regular circuits [84]. We continue to use combinators for describing the ways in which circuits are built. What we have gained through the embedding in Haskell, is the availability of a full-blown programming language. The synchronous programming languages Lustre, Esterel and Signal can all be used to describe hardware in much the style used here. Further experiments in this direction are being carried out in the EU project SYRF.

A source of inspiration has been John O'Donnell's Hydra system [72]. In Hydra, circuit descriptions are more direct because they are written in 'ordinary' Haskell. There are no monads cluttering up the types, and this must be an advantage. It is our use of monads, however, that makes Lava easily extensible, while Hydra is less so. The Hydra system has not, as far as we know, been used to generate formulas from circuit descriptions, for input to theorem provers, although the idea of having multiple interpretations has been a recurring theme in O'Donnell's work.

Launchbury and his group are experimenting with a different approach to using Haskell for hardware description [24]. In Hawk, a type of signals and Lustre-like functions to manipulate it are provided. Circuits are modelled as functions on signals, and the lazy state monad is used locally to express sequencing and mutable state. The main application so far has been to give clear and concise specifications of superscalar microprocessors. Simulation at a high level of abstraction has been the main circuit analysis method. Work on using Isabelle to support formal proof is under way, however. Also, it seems likely that Lava input could be generated from Hawk circuit descriptions. We plan to explore this possibility in a joint project. Hawk has, at present, no means of producing code for the production of real circuits, although work on circuit synthesis is in progress.

Keith Hanna has long argued for the use of a functional language with dependent types in hardware description and verification [49]. Hanna's work inspired much research on using Higher Order Logic for hardware verification. The PVS theorem prover, which is increasingly used in hardware verification [28], is also based on a functional language with dependent types. We do not know of work in which circuit descriptions written in this language are used for anything other than proof in PVS.

HML is a hardware description language based on ML, developed by Leeser and her group [62]. The language benefits from having higher order functions, a strong type system and polymorphism, just as ours does. The emphasis in HML is on simulation and synthesis, and not on formal verification.

2.6 Conclusions

The Lava system is an easily extensible tool to assist hardware designers both in the initial stages of a design and in the final construction of a working circuit. The system allows a single circuit description to be interpreted in many different ways, so that analyses such as simulation, formal verification and layout on a Field Programmable Gate Array are supported. Furthermore, new interpretations can be added with relatively little disturbance to the existing system, allowing us to use Lava as the main workbench for our research in hardware verification methods for combinational and sequential circuits. To be able to provide these features, we rely heavily on advanced features of Haskell's type system: monads for language embedding, polymorphism and type classes to support different interpretations, and higher order functions for capturing regularity.

The system is an interesting practical application of Haskell, which has proved to be an ideal tool, both as a hardware description language and as an implementation language. As demonstrated in the FFT examples, our circuit descriptions are short and sweet, when one can find a suitable set of combinators. Our experience with Ruby indicates that each domain of application (such as signal processing, pipelined circuits or state machines) gives rise to a small and manageable set of combinators.

The largest circuit that has been tackled so far is a 128 bit by 128 bit combinational multiplier. To deal with this circuit, we needed to use a Haskell compiler (HBC) rather than Hugs.

Writing the Lava system has been an educational exercise in software engineering. More than once, we have thrown everything away and started again. The latest version exploits Haskell's type system to impose a clear structure on the entire program, in a way that we find appealing. We have all been taught to think about types very early in the design of a system. Lava demonstrates the advantages of doing so.

2.7 Future Work

We are continuing to develop the Lava system; this paper is a report of work in progress rather than a description of a finished project. Until recently, we had several specialised versions of Lava, each concentrating on a particular aspect of design such as verification or the production of VHDL. The work of merging these versions has only just begun; it was really the need for fusion that pushed us towards the current system design. Incorporating the interpretation that takes care of layout production is a non-trivial task, as this code is necessarily large and complicated. This may lead to further changes to the top level design of Lava.

To make the system more usable, we need to add many new interpretations. For example, we would like to work on test pattern generation and testability analysis, using earlier work by Singh as a basis [90]. All of these interpretations must be tested on real case studies.

We would be able to generalise our system further if multiple parameter type classes were provided in Haskell. At present, all of the interpretations share the same primitive datatypes. Using multiple parameter type classes, each interpretation could support its own data types, with the required features.

In the area of verification, we are working on interpretations involving sequential operations, such as delay, and on related methods to automatically prove properties of sequential circuits. We are working on a case study of a sequential FFT implementation provided by Ericsson CadLab. Inspired by the Hawk group, we find it hard to resist investigating verification of the next generation of complex microprocessors. In particular, we are interested in the question of how to *design* processors to enable verification to proceed smoothly.

```
%% Automatically generated by Lava %%

%% THEORY Arithmetic %%
list(demodulators).
eq(tim(x, plus(y, z)), plus(tim(x, y), tim(x, z))).
eq(tim(x, sub(y, z)), sub(tim(x, y), tim(x, z))).
eq(tim(1, x), x).
eq(tim(x, tim(y, z)), tim(tim(x, y), z)).
end_of_list.

%% THEORY Twiddle Factors size 4 %%
list(demodulators).
eq(W(x, 0), 1).
eq(W(x, x), 1).
$LE(x, 4) -> eq(W(x, y), W($PROD(2,x), $PROD(2,y))).
eq(tim(W(x,y),W(x,z)),W(x,$SUM(y,z))).
end_of_list.

%% SYSTEM + QUESTION %%
list(sos).
eq(x,x).
-eq(sub(sub(a4, tim(W(2,0), a2)), tim(W(4,1),
   sub(a3,tim(W(2,0), a1)))), tim(W(4,0),
   sub(sub(a4, a2),tim(W(4, 1), sub(a3, a1))))) |
-eq(sub(plus(a4, tim(W(2,0), a2)), tim(W(4,0),
   plus(a3, tim(W(2,0), a1)))), tim(W(4, 0),
   sub(plus(a4,a2), plus(a3, a1)))) |
-eq(plus(sub(a4, tim(W(2,0), a2)), tim(W(4,1),
   sub(a3,tim(W(2,0), a1)))), tim(W(4,0),
   plus(sub(a4, a2),tim(W(4,1), sub(a3, a1))))) |
-eq(plus(plus(a4, tim(W(2,0), a2)), tim(W(4,0),
   plus(a3, tim(W(2,0), a1)))), tim(W(4, 0),
   plus(plus(a4,a2), plus(a3, a1)))).
end_of_list.
```

Figure 2.11: Otter input for size 4 FFT comparison

Chapter 3

Automatic Verification of Combinational and Pipelined FFT Circuits

We describe how three hardware components (two combinational circuits and one pipelined) for computing the Fast Fourier Transform have been proved equivalent using an automatic combination of symbolic simulation, rewriting techniques, induction and theorem proving. We also give some advice on how to verify circuits operating on complex data, and present a general purpose proof strategy for equivalence checking between combinational and pipelined circuits.

This chapter was published at the International Conference on Computer Aided Verification 1999 [25].

3.1 Introduction

FFT components are a challenge to verify as they compute complex functions involving many arithmetic operations. Bit-level correctness proofs for such circuits are not within the reach of today's technology; an appropriate level of modelling is therefore on the level of individual arithmetic operations on signals carrying numerical data.

In order to make verification techniques industrially interesting, it is generally agreed that a high degree of automation is desirable. Unfortunately classical automatic methods such as propositional logic tautology checking or model checking can not be immediately applied at this level of abstraction. Different extensions of model checking with uninterpreted functions encoded in BDDs have been proposed [96]; we instead use theorem proving, but in such a way that no user guidance is needed during the proofs.

As we aim for verification at the arithmetic level, it is imperative to structure the proofs to be as simple as possible; we therefore devise heuristics for the particular class of circuits we verify and apply automatic analyses that aim to reduce the work that has to be done in the theorem prover. For this end we use the Lava hardware development platform that has a powerful language in which we can implement our analyses and write parametrisable scripts that control complex theorem prover interactions [11].

The work described is an industrial case study with Ericsson Cadlab, Stockholm.

3.2 The Lava Hardware Development Platform

Lava is a hardware description language and a framework for hardware verification developed at Chalmers and Xilinx [11]. One of the principal uses of Lava is as a platform for hardware verification experiments.

Lava is embedded in the functional language Haskell; all aspects of the development of hardware from descriptions down to the interfacing to layout tools are expressed in the same language. The use of a polymorphic high level language that supports higher order functions gives very concise hardware descriptions and allows us to devise combinators that capture common design patterns.

The circuit descriptions can be interpreted by symbolic evaluation in a number of different ways; examples of built in standard analyses are circuit simulation, generation of logical formulas in formats suitable for external theorem provers and generation of VHDL. The verification interpretation is parametrised over the proof procedure and allows the passing of optional proof parameters; a user can therefore quickly retarget from one proof procedure to another without losing fine grain control.

3.3 The Fast Fourier Transforms

The Fast Fourier Transforms (FFTs) are efficient algorithms for computing a length N sequence of complex numbers X given an initial sequence x and a constant W_N defined as $e^{-j2\pi/N}$:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \qquad k \in \{0 \ldots N{-}1\}$$

The FFTs exploit symmetries in the *twiddle factors* W_N^k together with restrictions of sequence lengths (for example to powers of two) to reduce the number of necessary computations. Examples of twiddle factor laws that express useful symmetries are

$$\begin{aligned} W_N^0 &= 1 \\ W_N^N &= 1 \\ W_n^k \cdot W_n^m &= W_n^{k+m} \\ W_n^k &= W_{2n}^{2k}, \qquad (n, k \leq N) \end{aligned}$$

The FFT algorithms are often implemented in combinational hardware, and are key building blocks in signal processing applications; the FFTs are rumoured to be the worlds most implemented algorithms in hardware.

The reference FFT is the *decimation in time* Radix-2 algorithm, which operates on input sequences whose length is a power of two [77]. If the input length also is a power of four, the *decimation in frequency* Radix-2^2 FFT can be applied [51]. From a designer's point of view the question is whether the combinational circuits that implement these algorithms are equivalent. As the networks are fundamentally different, verification of equivalence is a non-trivial undertaking.

Combinational implementations are not the only ones possible; pipelined sequential designs can use less circuit area by trading space for time. A pipelined implementation of a size 2^n Radix-2^2 FFT (see figure 3.1) consists of two simple kinds of combinational components ($C1$ and $C2$) that together form a stage; a whole circuit consists of $n/2$ stages. Each primitive block is controlled by synchronisation signals generated by an n-bit counter. This counter also addresses a multi port memory that outputs streams of twiddle factors that are multiplied together with the outputs of each stage.

Figure 3.2 shows how the pipelined FFT circuit simulates the corresponding combinational circuit over time by reading the inputs in the first sequence of input values $IF(0)$ while spitting out undefined outputs until time *lag* ($2^n - 1$ for a size 2^n FFT) when the first element of the output sequence $OF(0)$ is generated; the lag time is always constant. At the same time as the outputs are produced, inputs from a new input sequence are read so that the circuit continuously processes data.

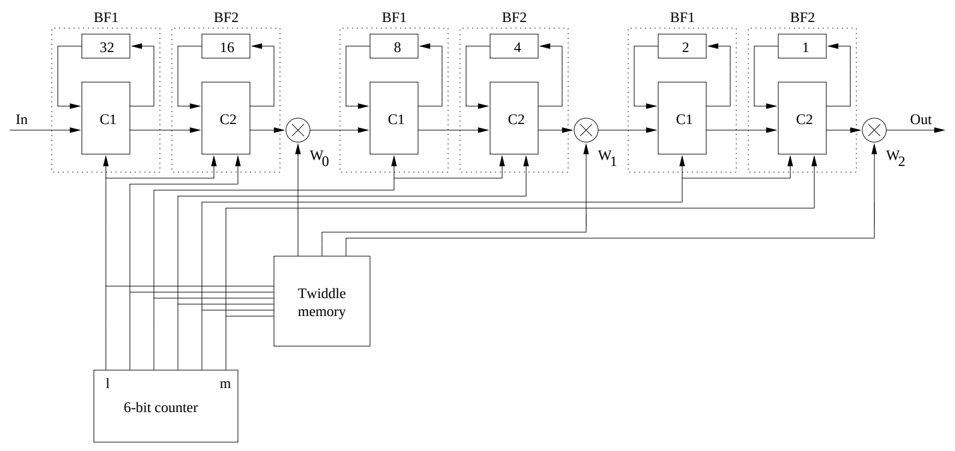

Figure 3.1: Structure of pipelined implementation of a size 64 Radix-2^2 FFT

3.4 FFT Low-level Descriptions

The FFT descriptions are parameterised by the circuit size and are formulated using a number of simple circuits and combinators that are useful for signal processing applications.

A key point is that the regularity of the combinational networks makes the circuits very easy to describe in Lava; the description of the Radix-2 FFT in terms of the signal processing combinators is just 3 lines long (see appendix 3.10).

The Lava circuit descriptions can be used to automatically generate structural VHDL for all parts of the implementations with the exception of the multi port memory component.

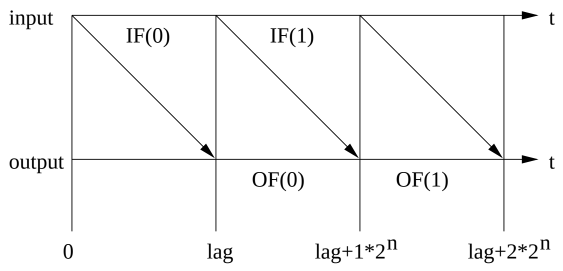

Figure 3.2: Operation of the pipelined circuit

3.5 Verification of Components

As we want automatic proofs, we will only be concerned with equivalence checking for fixed size circuits. We will also exploit designer knowledge and use Lava analyses in order to make the proofs tractable for the external proof procedure. The circuits are modelled on the level of operations on infinite precision complex numbers; this modelling is appropriate as finite representations of complex numbers only can be used for approximate calculation of the FFT. A reasonable notion of implementation equivalence must therefore be defined in terms of infinite precision complex arithmetic.

As a shorthand, we adopt the convention that

$$F(x,y) \equiv F(x(0)..x(i-1), y(0)..y(i-1))$$

if $F \in Form$ (the set of first order logic formulas) and $x, y \in S^i$ where S is any non-empty set.

3.5.1 Theoretical Basis of the Verifications

Combinational circuits can be viewed as functions f from input to output. Lava's symbolic evaluation can generate formulas δ_f that define the functions we are concerned with in the sense that $T \vdash \delta_f(I,O) \Rightarrow f(I) = O$ if T is a theory containing theorems that are true in a standard interpretation of complex arithmetic.

The formulas that are constructed in the following verifications are expressed in first order logic with equality, and contain variables and two-place function symbols *plus*, *sub*, *tim* and W. The circuit equivalence checking problem is reduced to showing that certain formulas that capture implementation equivalence are members of the theory T which we give axioms for. The axioms are well-known properties of complex arithmetic and some twiddle factor identities. We know that the axioms hold in the interpretation $\mathfrak{I}$ that complies with the following conditions

- The domain is the set of complex numbers
- *plus* designates complex addition
- *sub* designates complex subtraction
- *tim* designates complex multiplication
- W designates the function $f_w(k,N) = e^{-j2\pi k/N}$

All formulas that are derivable from the axioms in a sound proof system are therefore also true in $\mathfrak{I}$.

3.5.2 Combinational FFT Verification

Are the abstract implementations of the Radix-2 and the Radix-2^2 FFT equivalent for sizes that are an exponent of four?

The fixed size FFT circuits are functions $F_1(I)$ and $F_2(I)$ from complex input sequences to complex output sequences. Lava's symbolic evaluation can generate formulas δ_1 and δ_2 that define these functions. Our criterion for equivalence of the combinational FFT is that

$$\delta_1(I, O_1) \wedge \delta_2(I, O_2) \;\rightarrow\; O_1 = O_2$$

Instead of generating the two defining formulas individually and then combining them together to a resulting formula, we can construct a test bench circuit that directly generates the correctness formula when interpreted symbolically:

```
fftSame n =
  do inp <- newCmplxVector (4^n)
     out1 <- radix2 (2*n) inp
     out2 <- radix22 n inp
     equals (out1,out2)
```

The test bench builds a vector of unrestricted complex variables, which are given to both FFT implementations. The resulting output sequences are then point-wise compared to each other for equality. If the formula describing this system is derivable by the theorem prover using the axioms for the theory T, then it is true in the model $\mathfrak{I}$ and the implementations are equivalent.

Lava's verification interpretation takes a test bench circuit and a proof procedure with some arguments, and automatically generates formulas and runs the proof. The manual step that has to be taken is to choose a prover and possibly give proof options. In this case, we have to choose a first order logic theorem prover, and specify some axioms. These include some simple algebraic laws for the arithmetic operators, such as distributivity of multiplication over addition and that 1 is a unit element for multiplication. The twiddle factor identities from section 3.3 are also necessary.

Although these axioms with any first order logic prover are in theory sufficient to prove the circuits equivalent, the number of consequences grows very quickly if the rules are applied mindlessly. This combined with the fact that the FFT circuits generate formulas that for larger sizes grow to be megabytes big means that we must give extra proof options in order to make the proofs tractable. Symbolic evaluation of the FFTs for 4 abstract inputs reveals some interesting circuit properties (the input and output vectors are indexed backwards):

```
Lava> symbolic_eval (radix2 2)
[(x3 - W(2, 0) * x1) - W(4, 1) * (x2 - W(2, 0) * x0),
 (W(2, 0) * x1 + x3) - W(4, 0) * (W(2, 0) * x0 + x2),
 W(4, 1) * (x2 - W(2, 0) * x0) + (x3 - W(2, 0) * x1),
 W(4, 0) * (W(2, 0) * x0 + x2) + (W(2, 0) * x1 + x3)
]

Lava> symbolic_eval (radix22 1)
[W(4, 0) * ((x3 - x1) - W(4, 1) * (x2 - x0)),
 W(4, 0) * ((x1 + x3) - (x0 + x2)),
 W(4, 0) * (W(4, 1) * (x2 - x0) + (x3 - x1)),
 W(4, 0) * ((x0 + x2) + (x1 + x3))
]
```

The lack of control logic in the combinational FFT components causes the circuit outputs to be polynomials in the inputs and twiddle factors only. Rewriting of the expressions by simplifying away twiddle factors that are equal to 0 or 1, conversion of the remaining twiddle factors to the form W_N^x and restructuring of arithmetic expressions to sum of products form makes it possible to show the two results equal by syntactic equality alone.

The rewriting has to be done in a particular way for it to be applicable to the larger circuits. If the axioms are given as standard equalities, they can be used in both directions. This is not how the most efficient proof would proceed, as it suffices to use all the rules in one direction only: expand out the polynomials, take away trivial twiddle factors and rewrite the others.

Unidirectional rules are therefore more suitable for our purposes. The theorem prover Otter has efficient such rules that are called demodulators [66]; the use of a demodulation rule can be unconditional or restricted by predicates on terms. An important property of these rules is that they are used as often as possible *without accumulating intermediate results.* This reduces the number of consequences and makes normalisation of large expressions tractable.

The demodulation proof rules are specified inside Lava and passed to Otter as two theories. The actual proofs are done by calling the verification interpretation on the test bench and the proof configuration:

```
options = [Prover otter, Theory arithmetic, Theory (twiddle 4)]

Lava> verify options (fftSame 1)
Valid
```

In this way the equivalence of circuits up to size 256 is proven automatically. Statistics for the resulting proofs and some system formula measures such as the number of primitive logical and arithmetic operations are given in table 3.1. The running times are measured on a 300 MHz Sun Enterprise 450.

FFT size	Verification time (sec)	Formula size (bytes)	# of variables	# of formula operations
4	0.09	1179	33	59
16	0.39	10 761	233	433
64	10.31	172 088	1334	2529
256	827.01	2 886 561	6939	13 313

Table 3.1: Statistics for verification of equivalence between combinational FFTs

3.5.3 Pipelined FFT Verification

We would now like to verify that the sequential pipelined implementation of the Radix-2^2 is equivalent to the combinational circuit. We employ a strategy that is optimised for equivalence checking of combinational and constant delay ("lag") pipelined circuits.

The presentation is divided into two parts: The first part describes the strategy and the second demonstrates how it applies to the particular case of our FFT verification.

A Strategy for Pipeline Equivalence Proofs. If we observe the pipelined circuit for a single clock period, it is a function from a starting state S and input I to a finishing state S' and a resulting output O.

$$(O, S') = ppl(I, S)$$

We use the term "frame" to refer to a complete in- or output data sequence for the combinational or pipelined circuit. Lava can generate a defining formula $\delta_{ppl}(I, S, O, S')$ for the $ppl(I, S)$ transition function that captures how the circuit behaves over a single clock tick. The objective is to show equivalence between the two implementations for any number of successive frames starting from a (partially) specified initial state, using the following verification strategy which we refer to as $Equiv_\omega$:

1. Generate the defining formula $\delta_{ppl}(I, S, O, S')$ of the pipelined circuit.

2. Define l to be the number of inputs that the pipelined circuit has to consume before it can read the first input of the second frame.

3. Define m as the least number of time steps that the pipelined circuit has to run to allow an observer to deduce that the output from the sequential circuit matches a single frame of output from the combinational implementation.

4. Let $k = max(l, m)$.

5. Let δ_{ppl}^k be the following formula that expresses what behaviour a length k trace of the sequential circuit exhibits

   $$\delta_{ppl}(I_0, S_0, O_0, S_1) \wedge \delta_{ppl}(I_1, S_1, O_1, S_2) \wedge \ldots \wedge \delta_{ppl}(I_{k-1}, S_{k-1}, O_{k-1}, S_k)$$

   This is the k-step unrolling of the pipelined transition function.

   We refer to a trace that is a model for δ_{ppl}^k as a T trace, and observe the following:

   - If we define an initialisation state as a state that immediately precedes the processing of a new frame, both S_0 and S_l are initialisation states on all T traces. Furthermore, S_l is the closest initialisation state to S_0.
   - Any infinite trace of the system is made up from infinitely many concatenated T traces; given that $l < k$ successive traces tr_n and tr_{n+1} also overlap with $tr_n(l \ldots k-1) = tr_{n+1}(0 \ldots k-l-1)$.

6. Generate a defining formula for the combinational circuit, $\delta_{cmb}(I, O)$.

7. From δ_{ppl}^k and δ_{cmb}, construct a formula λ that expresses implementation equivalence for a single frame of inputs

8. A proof of λ without any assumptions at all on the initialisation state S_0 implies $\forall S_0.\lambda$. This corresponds to equivalence for any number of time frames as the circuits will behave in the same way regardless of the initialisation state values before a new frame is processed; a direct proof of λ is hence not realistic. Therefore strengthen the assumptions on S_0 by a formula ϕ that restricts some of the S_0 variables to the initial values given in the pipelined circuit description. If now

   $$\phi(S_0) \ \rightarrow \ \lambda$$

   is provable, the circuits are equivalent for any number of time frames under the assumption that ϕ is always true in initialisation states. Refer to this assumption as assumption A

9. Try to prove assumption A valid by a proof of

   $$\phi(S_0) \wedge \delta_{ppl}^k \ \rightarrow \ \phi(S_l)$$

   As ϕ holds in the initial state of the circuit, this formula implies A as it asserts that ϕ will hold in the state S_l (that is reached immediately before a new processing cycle is initiated) if ϕ is true in S_0 (that was reached immediately before this frame was processed); A is therefore entailed by induction.

10. If step 8 and step 9 were successful, deduce multi frame equivalence

A valid question is, of course, "Why is it reasonable to assume that a part of the pipelined circuit always is in a state where ϕ holds before a new frame is read?". This is probable as the pipelined circuit is supposed to repeat the frame processing behaviour again; the registers in the control logic should therefore have similar contents in the initialisation states as in the specified initial circuit state.

By having reduced the problem to two simple proofs we have devised a simple strategy for showing pipelined circuits with a fixed lag equivalent to combinational implementations. This strategy is implemented in an automatic Lava proof script that is parameterised over circuit descriptions, frame length, the constant lag and a proof configuration for the frame equivalence proof. This script automatically generates and reduces all formulas as much as possible before calling the theorem prover specified in the proof configuration; the only manual steps are to choose which state variables to restrict and to select a proof procedure. Any prover and extra proof options can be specified in the proof configuration; the pipelined circuit description can also have as many or as few initial values given as desired.

Application to the Pipelined Radix-2^2 FFT. The script that implements $Equiv_\omega$ proves pipeline equivalence for the FFT circuits with the automatically generated equivalence formula λ defined as

$$\delta^k_{ppl}(I_0..I_{k-1}, S_0..S_k, O_0..O_{k-1}) \wedge \delta_{cmb}(I_0..I_{i-1}, O'_0..O'_{i-1}) \rightarrow O_{lag}..O_{k-1} = O'_0..O'_{i-1}$$

where $lag = 2^N - 1$, $i = 2^N$ and $k = 2^N + lag$.

A sufficient restriction ϕ on the initial state of the pipelined FFT circuit is that the n-bit counter is initialised to 0. The reason why this simple assertion is strong enough to prove the FFT implementations equivalent is that at re-initialisation the rest of the pipeline state is unimportant, new values have to be read for processing anyway. This is likely to hold for most pipelined implementations of combinational circuits.

The initialisation information ϕ is always used by the Lava script to reduce the generated formulas as much as possible while they are produced. This reduction computes the values of logical expressions whenever possible and propagates the resulting new information. As a consequence, the formulas that specify the behaviour of the control logic inside the pipelined FFT are evaluated away and the re-initialisation invariant in step 9 of $Equiv_\omega$ is proved by syntactic equality. The equivalence checking problem for the pipelined FFT is therefore reduced back to a proof of an equivalence formula that turns out to be amenable to normalisation with the theories used for the combinational equivalence checking. The complexity of the resulting proofs are indicated in table 3.2.

FFT size	Verification time (sec)	Formula size (bytes)
4	0.05	1227
16	0.61	10 045
64	22.26	162 862
256	1361	2 797 617

Table 3.2: Statistics for verification of pipelined equivalence

3.5.4 Manual Preparation

Approximately two weeks was spent on studying the FFT implementations, devising signal processing combinators and writing circuit descriptions. The addition of support in Lava's interpretations for complex numbers and the writing of the symbolic simulation interpretation with automatic formula reduction took one week of work each.

Finding the proof procedure was the creative step for the combinational FFT verification. Two other theorem provers, Prover [92] and Gandalf [95], was tried before Otter. Prover lacked crucial arithmetic laws, and Gandalf did not support the unidirectional rules that were needed to make the proofs scale up. A correct set of rewrite rules took some hours work by two users, Koen Claessen and Tanel Tammet, who were unfamiliar with the FFT but knew Otter well. Any other applicable proof procedure would also have needed rewrite rules for the twiddle factors, so we believe that this degree of manual work is unavoidable.

Once the symbolic simulation interpretation with formula reduction was written, a first (more involved) pipeline proof script could be constructed in half an hour. This strategy was successful the first time it was tried; we later simplified the heuristic to the presented form. The only non-reusable steps of the combinational and pipelined verifications were to choose Otter with rewrite rules as the proof procedure and to restrict the synchronisation counter state to the initial state 0.

3.6 Lessons Learned

The FFT circuits are representatives for a general class of circuits that compute complex functions without using a large amount of boolean control logic. In general, a few guidelines for proofs of circuit equivalence for such circuits can be drawn out of the FFT work:

- For each problem domain, it might be possible to find a small number of generalised proof scripts that can be powerful enough for a particular class of problems to make proofs automatic in most cases. These scripts should be parametrisable by proof options so that they not are too blunt to be reusable.

- As the proofs that have to be done when operations like arithmetic are involved are relatively complex, the prover's job must be simplified as much as possible. The use of automatic partial evaluation and formula reduction can in some cases lessen the need for prover inferences drastically. A tool like Lava that supports analyses like simplification of formulas by propositional reasoning and cone-of-influence analysis can help the designer simplify the problem at hand.

- It is not always necessary to explore the state space of a design. Ordinary induction can sometimes avoid very complex or intractable computations, and make for uncomplicated proofs.

- Normal form rewriting is a powerful technique that can be implemented very efficiently using modern rewrite engines. However, the use of unidirectional rules is crucial to make the strategy applicable to larger circuits.

3.7 Related Work

The Radix-2 FFT algorithm has previously been verified against the DFT using the ACL2 theorem prover [38]. The level of abstraction in this verification was high and the proof thus required substantial user interaction. In contrast, we have aimed for fully automatic proofs, and verified the hardware FFTs at the netlist level. Our proofs are only for equivalence of fixed size circuits, but are not reliant on circuit regularity.

The pipeline proof principle bears some resemblance to the refinement mapping approach to pipelined microprocessor control verification [17, 27]. However, as we are comparing a pipelined circuit against a combinational one, we cannot directly associate a single sequential step with the combinational implementation; we instead correlate whole frames. We also exploit the fact that constant lag pipelined circuits are targeted.

There are alternatives to Otter as a proof procedure: the Stanford validity checker decides quantifier free first order logic with linear arithmetic and uninterpreted functions by boolean case splitting (backtracking), rewrites and congruence closure [4]. SVC has been used extensively in hardware verification, and is used as the decision procedure in the Burch and Dill approach to microprocessor verification [17]. Multiway decision graphs are a variation on the ROBDD theme that accommodates abstract data types, uninterpreted function symbols and rewrite rules [101]; this data structure has been used to verify non-pipelined microprocessors and an ATM switch [94]. MDGs give a canonical representation for a fragment of quantifier free first-order formulas and support exploration of abstract state spaces (but do not guarantee convergence of fixpoint computations). As we have demonstrated, it is not always necessary to do such expensive computations; induction and normalising can be both sufficient and efficient.

Both MDGs and SVC need the user to provide rewrite rules or a normaliser for new theories. This means that the manual step of finding a normal form for twiddle factors is also necessary with these proof procedures.

3.8 Conclusions

This paper has shown how some FFT circuits have been verified from within the hardware development tool Lava after the existing system was extended with complex numbers and a general purpose strategy for equivalence checking of combinational and fixed lag pipelined circuits. The verification has been automatic in the sense that the only manual proof steps has been to select the proof procedure, rewrite rules and the initial state variables to restrict. The proofs are at a relatively low level, which should give a high confidence in the correctness of the modelled circuits; the logical formulas has been generated by symbolic evaluation of the hardware descriptions. No part of the verification has relied on the specific way that the arithmetic operators are implemented, or the representation of complex numbers. However, the proofs are not general in the size of the FFT; different instances have to be proved separately.

We have also presented an induction principle that exploits the problem structure of equivalence checking between a pipelined circuit and a combinational reference circuit, and contributed some suggestions for verification of circuits that contain little control logic but do complicated computations expressed in abstract operations.

3.9 Future Work

Lava is optimised for developing and verifying hardware. We pay for the strength we gain by limiting the problem domain, however, by presently being unable to reason internally about the proof strategies. Instead we have to go outside the system to a general purpose interactive theorem prover and do high level proofs there. We would like to have Lava integrated with a proof system that would allow us to do this kind of reasoning.

The counter examples that are produced by proof procedures are formatted and passed back to the user by Lava; unfortunately many first order logic theorem provers (including Otter) lack such capabilities. For verification with normal form rewriting to be smooth, it must be easy to find a rewriting theory quickly. It is therefore imperative to have some tool that analyses the output of a failed proof and allows the user to deduce what rules are missing, or gives the user good clues to why the two formulas are not equivalent. This is something that should (and will) be implemented in Lava as a proof analysis.

3.10 Appendix

3.10.1 The Radix-2 FFT description

Figure 3.3 shows a size 16 Radix-2 FFT network, where merging arrows indicate addition and constants under a wire indicate multiplication. The Lava description of the size 2^n Radix-2 FFT circuit follows the network structure closely, and is parametrised by n:

```
radix2 n =
  bitRev n >-> compose [ stage i | i <- [1..n] ]
 where
  stage i = raised (n-i) two (twid i >-> bflys (i-1))
  twid  i = one (decmap (2^(i-1)) (wMult (2^i)))
```

The FFT circuit is made up from the sequential composition of an initial bit reversal permutation network (not shown in the picture) and n circuit stages. Stage i is a column of 2^{n-i} components that each contains a twiddle factor multiplication stage sequentially composed with a butterfly network. Given that $x = 2^{i-1}$, a size i multiplication stage performs multiplications with $W_{2^i}^0 ... W_{2^i}^{x-1}$ on the respective wires of one half of a bus, while passing the other half through unchanged.

More information on the signal processing building blocks and the descriptions of the combinational circuits can be found in [11].

Acknowledgements

Thanks to Koen Claessen and Tanel Tammet for finding the Otter rewriting theory, and to Mary Sheeran and Byron Cook for careful readings of earlier drafts.

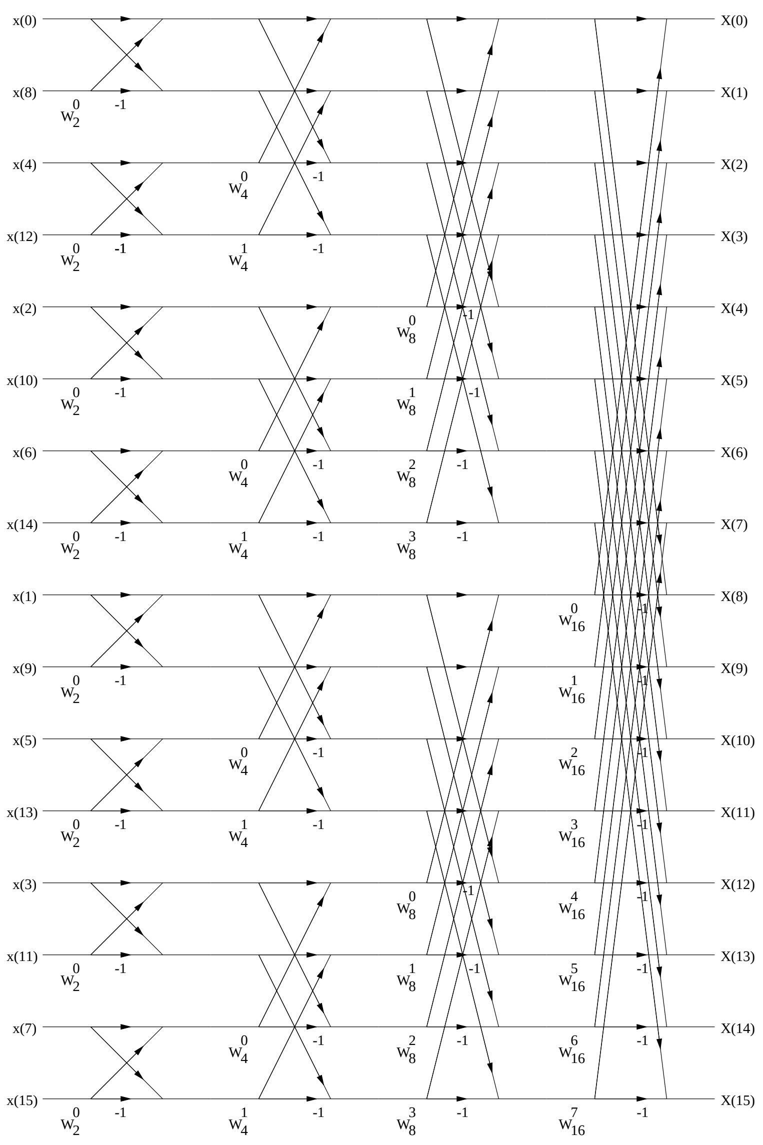

Figure 3.3: The structure of a size 16 Radix-2 FFT

Chapter 4

Symbolic Model Checking with Sets of States Represented as Formulas

We analyse traditional symbolic model checking to isolate the operations that a representation for sets of states need to support and present a different view, where sets of states are represented by propositional logic formulas. High level algorithms for symbolic model checking based on Stålmarck's method are then presented, which results in a model checking procedure closely related to bounded model checking. Our method lifts the requirement on finding bounds and allows intermediate minimisations of state set representations. We also sketch how to generalise the approach to stronger logics.

This chapter is a revised version of technical report CS-1999-103, Chalmers University of Technology, 1999 [9].

4.1 Introduction

Propositional temporal logic theorem proving is PSPACE-complete, and often infeasible for industrial-size systems. Model checking is a reduction of this complex problem to a possibly exponential number of subproblems that hopefully have better characteristics.

Different model checking approaches achieve success with different classes of systems; the suitability of a particular method is governed by the tradeoff between space complexity and the time complexity of the operations. One of the most important determining factors for the time/space ratio is the choice of set representation.

We will present a new view of states and sets of states where the verification problem is reduced to formula minimisation and tautology checking on a sequence of propositional logic formulas. This approach has the advantage that the representation can be made very compact; some of the operations are on the other hand complex since tautology checking is coNP-complete. The time/space ratio will therefore be very different from that of many traditional model checking methods.

The logical view of set representation has the benefit of allowing generalisations to richer logics, which for instance makes direct model checking of infinite state spaces possible. Our purpose in this paper is to present a different angle on model checking, introduce the new family of representations, and demonstrate some of the possibilities.

4.2 Overview

The presentation is organised as follows: In section 4.4 and 4.5 we analyse CTL model checking and extract requirements on representations for sets of states, section 4.6 introduces the idea of encoding sets of states as propositional logic formulas and relates it to BDDs. Section 4.7 presents algorithms that use Stålmarck's method to implement the ideas in practice, and relates the approach to bounded model checking. Section 4.8 finally sketches how to extend the ideas to two other logics and the paper is concluded by some remarks about time/space tradeoffs.

4.3 Conventions

Infinite composition of a function f is denoted f^ω, where the value of $f^\omega(x)$ is equal to $f^k(x)$ if there is a k such that $f^k(x) = f^{k+1}(x)$ and otherwise is undefined. The notation $g : x \longmapsto e(x)$ means that g is a function that maps every x to $e(x)$.

We presume basic knowledge of first order logic (FOL) and propositional logic (PROP). The propositional constants (true,false) are denoted $(\top, \bot)$ and substitution of e for every occurrence of p in ϕ is written as $\phi[e/p]$. The size of a formula ϕ, $|\phi|$, is the sum of the number of variable occurrences and the number of connectives in ϕ ($\top$ and $\bot$ therefore have size 0).

We also make some use of an extension of propositional logic called quantified boolean formulas, QBF. This logic is propositional logic enriched with quantifiers that range over propositions, with the semantics of quantifiers defined by

$$\exists p.\phi \equiv \phi[\top/p] \vee \phi[\bot/p]$$
$$\forall p.\phi \equiv \phi[\top/p] \wedge \phi[\bot/p]$$

These identities make it possible to map any QBF formula to an equivalent propositional logic formula.

4.4 CTL Model Checking

Model checking is the process of determining the set of states in a Kripke structure that satisfies a temporal logic formula. For simplicity, our investigations are based on computational tree logic (CTL) model checking by fixpoint computations.

4.4.1 Syntax and Semantics

The language of CTL ($\mathfrak{L}_{CTL}$) is freely generated from a finite set of atomic state propositions, AP, according to the following rules:

$x \in \mathfrak{L}_{CTL}$	if $x \in AP$
$\neg x \in \mathfrak{L}_{CTL}$	if x $\in \mathfrak{L}_{CTL}$
$x \# y \in \mathfrak{L}_{CTL}$	if # is one of the connectives $\vee$, $\wedge$ or $\rightarrow$ and $x, y \in \mathfrak{L}_{CTL}$
$\Pi\, x \in \mathfrak{L}_{CTL}$	if Π is one of the operators AX, EX, AG or EG and $x \in \mathfrak{L}_{CTL}$
$\Pi[x, y] \in \mathfrak{L}_{CTL}$	if Π is one of the operators AU or EU and $x, y \in \mathfrak{L}_{CTL}$

The semantics of temporal formulas are defined with respect to a Kripke structure that contains a set of states *State*, a mapping *Label* that takes a state to the set of atomic propositions that label the state and a transition relation $Trans \subseteq State \times State$.

That a state s_0 in a Kripke structure M satisfies the temporal formula γ is henceforth written as $M, s_0 \models \gamma$. An atomic proposition p is true in a state exactly when it is in the labelling set of the state; the semantics of compound formulas are defined recursively:

$M, s_0 \models p$	iff $p \in AP$ and $p \in Label(s_0)$
$M, s_0 \models \neg\phi$	iff not $M, s_0 \models \phi$
$M, s_0 \models \phi \wedge \psi$	iff $M, s_0 \models \phi$ and $M, s_0 \models \psi$
$M, s_0 \models AX\,\phi$	iff for all states t such that $Trans(s_0, t)$ $M, t \models \phi$
$M, s_0 \models EX\,\phi$	iff for some state t such that $Trans(s_0, t)$ $M, t \models \phi$
$M, s_0 \models AU[\phi, \psi]$	iff for all paths $(s_0, s_1, ...)$, $\exists i \geq 0.\ M, s_i \models \psi \ \wedge\ \forall j.(0 \leq j < i \rightarrow\ M, s_i \models \phi)$
$M, s_0 \models EU[\phi, \psi]$	iff for some path $(s_0, s_1, ...)$, $\exists i \geq 0.\ M, s_i \models \psi \ \wedge\ \forall j.(0 \leq j < i \rightarrow\ M, s_i \models \phi)$

Disjunction, implication and the remaining temporal operators are regarded as abbreviations:

$\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$	
$\phi \rightarrow \psi \equiv \neg(\phi \wedge \neg\psi)$	
$AF\,\phi \equiv AU[\top, \phi]$	"ϕ is true in the future along all paths from s_0"
$EF\,\phi \equiv EU[\top, \phi]$	"ϕ is true in the future along some path from s_0"
$AG\,\phi \equiv \neg EF(\neg\phi)$	"ϕ holds in all future states along all paths from s_0"
$EG\,\phi \equiv \neg AF(\neg\phi)$	"ϕ holds in all future states along some path from s_0"

4.4.2 Computing Models of CTL Formulas

We are interested in determining the set of models $\overline{\phi}$ of a temporal formula ϕ relative to a particular Kripke structure M:

$$\overline{\phi} = \{s\ : M, s \models \phi\}$$

The set of models for an atomic proposition is directly computable; the connectives correspond to set operations

$$\begin{aligned} \overline{\neg\phi} &= State \setminus \overline{\phi} \\ \overline{\phi \vee \psi} &= \overline{\phi} \cup \overline{\psi} \\ \overline{\phi \wedge \psi} &= \overline{\phi} \cap \overline{\psi} \\ \overline{\phi \rightarrow \psi} &= (State \setminus \overline{\phi}) \cup \overline{\psi} \end{aligned}$$

Given the models for a formula ϕ, the set of models for $AX\,\phi$ and $EX\,\phi$ can be constructed by inspecting neighbouring states

$$\begin{aligned} \overline{EX\,\phi} &= \{x : \exists s.Trans(x, s) \wedge s \in \overline{\phi}\} \\ \overline{AX\,\phi} &= \{x : \forall s.Trans(x, s) \rightarrow s \in \overline{\phi}\} \end{aligned}$$

All of the remaining temporal operators have well-known fixpoint characterisations [35]:

$$\begin{aligned}
\overline{AF\,\gamma} &= \mu Y.\gamma \vee AX\ Y \\
\overline{EF\,\gamma} &= \mu Y.\gamma \vee EX\ Y \\
\overline{AG\,\gamma} &= \nu Y.\gamma \wedge AX\ Y \\
\overline{EG\,\gamma} &= \nu Y.\gamma \wedge EX\ Y \\
\overline{AU[\gamma_1\,,\gamma_2]} &= \mu Y.\gamma_2 \vee (\gamma_1 \wedge AX\ Y) \\
\overline{EU[\gamma_1\,,\gamma_2]} &= \mu Y.\gamma_2 \vee (\gamma_1 \wedge EX\ Y)
\end{aligned}$$

The fixpoint operators $\mu Y.\rho(Y)$ and $\nu Y.\rho(Y)$ takes a *temporal transition formula* $\rho(Y)$ to a set of states. A temporal transition formula can be thought of as a temporal formula schema, i.e. a formula that is parameterised with a variable ranging over other formulas. The denotation of a temporal transition formula is hence a set transformer: If Y is the only free variable in $\rho(Y)$, then

$$\overline{\rho} : \overline{\phi} \longmapsto \overline{\rho[\phi/Y]}$$

The fixpoint operators themselves are implicitly defined by

$\mu Y.\rho(Y) = S$	iff S is the least set satisfying the equation $\overline{\rho}(S) = S$
$\nu Y.\rho(Y) = S$	iff S is the largest set satisfying the equation $\overline{\rho}(S) = S$

All temporal transition formulas in the fixpoint characterisations denote monotone set transformers [35]; the equations are therefore well-defined.

Symbolic Model Checking. The basic algorithms for finding models for CTL formulas are uncomplicated; the tricky part is to make them scale up to industrial size systems. A major obstacle is that the algorithms will require space linear in the number of states if sets are represented by a straightforward enumeration. This means that systems with several hundreds of variables will have a slight chance of being verifiable. Symbolic model checking tries to get around this problem by instead representing sets by a necessary and sufficient criterion for being a member of the set. The choice of criterion and how it is encoded has great impact on the class of verifiable systems.

4.5 Requirements on a Representation for Sets of States

Primitive Model Checking Operations. A representation for sets of states must support all the operations that are needed to compute the set of models of arbitrary CTL formulas. The description of CTL model checking indicates that the following operations, that we refer to as *primitive model checking operations*, are necessary:

1. Atomic proposition model-finding. $m_{atom}(p) = \overline{p}$ for $p \in AP$
2. Compound formula model construction
   - Unary connective. $m_\neg : \overline{\phi} \longmapsto \overline{\neg\phi}$
   - Binary connectives. $m_\# : (\overline{\phi}, \overline{\psi}) \longmapsto \overline{\phi \# \psi}$ for # arbitrary propositional connective
3. Transitions operations
   - $m_{EX} : \overline{\phi} \longmapsto \overline{EX\ \phi}$
   - $m_{AX} : \overline{\phi} \longmapsto \overline{AX\ \phi}$
4. Fixpoint operations. m_μ, m_ν

Example 1 *Given that* msg *and* safe *are atomic propositions, the fixpoint characterisation of* AG *implies that the set of models of the temporal formula* msg $\rightarrow$ AG(safe) *is computed by*

$$m_\rightarrow(m_{\text{atom}}(\text{msg}), m_\nu(\lambda x.m_\wedge(m_{\text{atom}}(\text{safe}), m_{AX}(x))))$$

Efficiency Requirements. A representation for sets of states must also satisfy a number of other requirements in order to make model checking computationally feasible.

Let us use the term "suitable" to mean "has a representation that should not be too expensive to compute nor be too large for the sets of states that arise". Assuming $p \in AP$, it is clear that a representation for sets of states must have the following characteristics relative to a particular verification problem:

1. $m_{atom}(p)$ is suitable
2. $m_\neg, m_\#, m_{AX}$ and m_{EX} preserve suitability
3. Set equality can be decided efficiently to allow fixpoint stabilisation checking.
4. Set membership can be decided efficiently.

Different representations therefore have different suitability for each problem.

4.6 Encodings Sets of States in Propositional Logic

As the Kripke model M is only labelled with finitely many propositions, we can enumerate AP as $\{p_i : i < |AP|\}$ and in each state define

$$val_n^s = \begin{cases} p_n & \text{if } M, s \models p_n \\ \neg p_n & \text{otherwise} \end{cases}$$

We presume that each state is uniquely labelled (it is simple to modify the encodings to suit structures where this is not the case). This conveniently allows us to identify each state s with the set of literals $\{val^s_n : n < |AP|\}$; sets of states S are represented by a propositional logic formula ϕ that has AP as propositions and fulfils that

$$s \models \phi \ \text{ iff } \ s \in S \tag{4.1}$$

Set equality and set membership testing corresponds to tautology checking.

We assume a syntactic set transforming operator $next(S)$ that takes a set of propositional logic formulas and "primes" every propositional variable. Binary relations on states $R \subseteq State \times State$ are defined by propositional logic formulas σ for which it holds that their propositions come from $AP \cup next(AP)$ and that

$$s_1 \cup next(s_2) \models \sigma \ \text{ iff } \ R(s_1, s_2) \tag{4.2}$$

When we say that a set S or relation R is "defined by ρ", we mean that ρ is a formula that fulfils equation 4.1 or 4.2.

Given that the Kripke model M has a transition relation defined by τ, the primitive model checking operations for a propositional logic set representation have very simple definitions:

$$\begin{aligned} m_{atom}(p) &= p \\ m_{\neg}(\phi) &= \neg\phi \\ m_{\#}(\phi,\psi) &= \phi\#\psi \\ m_{EX}(\phi) &= \exists t.\tau(s,t) \wedge \phi(t) \\ m_{AX}(\phi) &= \forall t.\tau(s,t) \rightarrow \phi(t) \\ m_{\mu}(f) &= f^{\omega}(\bot) \\ m_{\nu}(f) &= f^{\omega}(\top) \end{aligned}$$

The next state operations m_{AX} and m_{EX} need to compute propositional logic formulas Θ that is the result of quantifying over a state s in a defining formula. This translates to nested boolean quantifications over the atomic propositions $p_0, p_1 \ldots$ in AP

$$\begin{aligned} \exists s.\phi(s) &\equiv \exists p_0.\exists p_1 \ldots \exists p_n.\phi \\ \forall s.\phi(s) &\equiv \forall p_0.\forall p_1 \ldots \forall p_n.\phi \end{aligned}$$

These QBF formulas can in turn be mapped to pure propositional logic formulas as described in section 4.3. A quantification of a state variable is therefore equivalent to $|AP|$ boolean quantifier eliminations on the defining formulas.

Both the fixpoint operations m_{μ} and m_{ν} take a formula transformer f as an argument, and use it to compute the fixpoint $f^{\omega}(\rho)$ of a sequence of formulas

ϕ_n with $\phi_0 = \rho$ and $\phi_{n+1} = f(\phi_n)$. The equality under consideration here is logical equivalence, which means that

$$f^\omega(\rho) = \phi_k \text{ iff } k \text{ is the smallest number such that } \models \phi_{k+1} \leftrightarrow \phi_k$$

All sets that are generated during the model checking can therefore be encoded as formulas. There are two essentially different ways of representing sets symbolically, both bearing simple relations to the defining formulas. Each of these representations have special strengths and weaknesses.

Semantic Representation. A simple representation for the defined sets is to encode the truth table of the propositional logic formulas. The encoding must be clever as the size of a truth table is exponential in the number of atomic propositions; representation size can therefore be lowered significantly by opting for a decision graph rather than a table.

Binary decision diagrams(BDDs) [16] are examples of such an encoding that also demand canonicity in order to turn equivalence checking of represented formulas into an $O(1)$ operation. The canonicity requirement consists of fixing an ordering of the variables; which particular ordering that is chosen has a large impact on the representation size.

BDDs have many good characteristics, and often allow model checking of state spaces that are surprisingly large. A design with 100 boolean variables (2^{100} states) has a good chance of being within the reach of the technology. However, it is not hard to find problems where the BDD representations grow too large even for relatively small circuits; for example, multipliers of greater width than 16 bits seem impossible to verify monolithically with BDDs.

Direct Formula Representation. Canonicity makes equivalence checking cheap, but can cause representations to explode. For certain systems non-canonical representations are therefore preferable.

An obvious, non-canonical representation for the sets defined by propositional logic formulas are the formulas themselves. Basing the representation on the actual formulas means that we choose a syntactic theorem-proving inspired approach to set encoding. There are many possible benefits to this view:

- Many sets with enormous BDD representations for any variable ordering have tractable formula representations
- Variable ordering is unnecessary
- The number of variables is less critical than for BDDs as tautology checkers routinely handle formulas with thousands of variables
- We can draw from the experience of the propositional logic theorem proving community

The naive implementations of the primitive model checking operations will however be space inefficient as

$$|m_{\#}(\phi, \psi)| \approx |\phi| + |\psi|$$
$$|m_{AX,EX}(\phi)| \approx |\phi| \cdot 2^{|AP|}$$

Special algorithms that select sufficiently small formula representatives are clearly needed.

It is also necessary to be able to decide set equality efficiently; any set can be represented by infinitely many equivalent formulas which means that set equality checking must be done by propositional logic theorem proving. This is a coNP complete problem, so it is unlikely that equality can be efficiently decided for any two sets. However, the choice of theorem prover and the particular formulas will govern the tractability of each case.

To summarise, we will at a bare minimum need the following operations for the implementation of practical propositional-logic based model checking:

- A tautology checker that is efficient for many of the formulas that arise in practice.
- A mechanism for minimising formulas.

4.7 Model Checking using Stålmarck's Method

Stålmarck's method [92] is a proof procedure for propositional logic, that has been industrially successful. It is remarkably proficient at tautology checking for many very large formulas that arise in practice.

We will in this section present how Stålmarck's method can be used to implement the necessary operations on the formula representation efficiently. Only the parts of the proof system that are necessary for understanding the algorithms will be presented; readers interested in further information are referred to Sheeran and Stålmarck's tutorial [87].

4.7.1 Formula Representation

Stålmarck's method represents a propositional logic formula as a set of *triples* and a set of *literal identifiers.* Each triple $i_n : r_1 \# r_2$ is formed from an identifier, a connective and two references; each literal identifier $i_n : l$ is formed from an identifier and a negated or unnegated formula variable. References are negated or unnegated identifiers.

The triple set that represents a given formula can be seen as the abstract syntax tree (AST) of the formula, with an identifier allocated to each internal node and formula literal; each subformula is hence denoted by a unique identifier. The AST root i_{top} is called the *top identifier*.

Given that we denote the set of triples that correspond to a given formula $Triple$ and the set of literals Lit, the set of entities under considerations in the proof is $Entity = Triple \cup Lit \cup Bool$ (special use is made of the two propositional constants $\bot, \top$). Each entity corresponds to a truth constant or a subformula. The size ordering on subformulas induce an ordering on entities; this allows definition of $min(e_1, e_2)$ and $max(e_1, e_2)$.

4.7.2 Equivalence Checking

At any stage of a proof, each $e \in Entity$ is in a unique equivalence class $[e]$ which can be considered to be the set of entities that at the moment are assumed to have the same truth value as e. We refer to the equivalence classes and the set of entities under consideration as the *system*. A singular system is a system where $[e] = \{e\}$ for all entities e; the representing system for ϕ, $repr(\phi)$, is the singular system that results from triplifying ϕ according to some reasonable strategy.

All proofs proceed by refutation; the proof procedure tries to derive a contradiction from an representing system which has been augmented by merging $[i_{top}]$ and $[\bot]$.

Example 1 *The system that is used to prove the formula $\neg a \wedge b \rightarrow b \vee c$ contains the set of entities*

$$\{i_1 : \neg a,\ i_2 : b,\ i_3 : c,\ i_4 : i_1 \wedge i_2,\ i_5 : i_2 \vee i_3,\ i_6 : i_4 \rightarrow i_5,\ \top,\ \bot\}$$

and the following equivalence classes:

$$\{\{i_1\},\ \{i_2\},\ \{i_3\},\ \{i_4\},\ \{i_5\},\ \{i_6, \bot\},\ \{\top\}\}$$

Inference rules are applied to the given system in order to propagate semantic information; every successful rule application merges at least two equivalence classes. A proof has been reached when $[\bot] = [\top]$; a system that satisfies this condition is called *explicitly contradictory*.

The inference rules comes in two varieties:

- Simple rules, that use information on the equivalence classes of i_i, i_j and i_k for a triple $i_i : i_j \# i_k$ to derive new information according to the semantics of the connective $\#$. As an example, if $[i_j] = [\top]$ and $\#$ is the disjunction operator, one of the simple rules will merge $[i_i]$ and $[\top]$.
- The Dilemma rule, that case splits over the truth value of a triple and adds the information that was derivable under both the assumption that the triple was false and under the assumption that the triple was true to the original system.

Exhaustive application of simple rules to a set of triples until no new equivalence classes can be merged is called *0-saturation*. This operation is linear in the number of entities, and is therefore very cheap.

1-saturation is the process of in turn applying the Dilemma rule to each entity in a set with an internal 0-saturation in each branch. Only one assumption has been made at all times; hence the name 1-saturation. Analogously (n+1)-saturation can be defined in terms of n-saturation.

Every tautology can be shown to have a degree of hardness, a minimum saturation level that is needed to arrive at an explicitly contradictory system from the augmented representing system. The complexity of n-saturation is exponential in n. However, a very large class of formulas that arise in formal verification turn out to have a hardness degree of 0 or 1. The power of Stålmarck's method is that the proof search is done in such a way that a proof for a formula with a low degree of hardness is found quickly.

4.7.3 Formula Minimisation

So, Stålmarck's method is in practice very efficient when it comes to finding proofs for large formulas. But can the proof procedure also do formula minimisation?

A completed k-saturation without any initial assumptions on the truth value of i_{top} results in a system where the equivalence classes contain entities that must have the same truth value in all interpretations. This information can be used to simplify the system by removing entities.

We define two entities e_1 and e_2 in a system sys to be k-discoverable as equivalent, denoted $equiv(k, sys, e_1, e_2)$, if k-saturation without assumptions results in a system that not is explicitly contradictory and that also fulfils that $\{e1, e2\} \subseteq eq$ for some equivalence class eq. We also assume the existence of an operation $subst(sys, e_1, e_2)$ that overwrites all the systems references to $max(e_1, e_2)$ with references to $min(e_1, e_2)$.

We refer to the following minimisation algorithm as *k-reduce*:

1. Find all pairs of k-equivalent entities in the system.
2. If the system was discovered to be explicitly contradictory during the search for k-equivalent entities, the system is equivalent to $\perp$. Stop.
3. Otherwise use the *subst* operator to iteratively reduce the system using the k-equivalent entity pairs (e_i, e_j) in an order that respects decreasing absolute value of $|e_i| - |e_j|$. Remove unconnected triples and literals, and then stop.

The resulting equivalent system will be smaller if at least one of the k-equivalent pairs contained entities of differing size. It is furthermore clear that the propositions of the resulting formula are a subset of the propositions of the original formula.

By choosing an appropriate notion of equivalence in step 1, we choose how much work we want to spend on compacting the representation. 0-saturation is a linear algorithm, which makes propagation equivalences easy to find. When the resulting compaction not is enough, the algorithms that use higher degrees of saturation can be applied; they have higher complexity but can find more removable subformulas. 1-saturation is for example guaranteed to find all shared subformulas [50].

4.7.4 The Connective Operators $m_{\#}$, $m_{\neg}$

It is sufficient to consider the binary connectives ($\neg\phi \equiv \phi \rightarrow \bot$ and $\bot$ can be encoded as a system only containing $\bot$).

We refer to the following algorithm that takes two given systems S_1 and S_2 and builds a composite system under an operator # as the *k-fuse* algorithm:

1. k-reduce S_1 and S_2
2. Build a new system S with the following characteristics
   - *Lit* contains the literals that occur in the formulas represented by S_1 and S_2.
   - *Triple* is the union of the sets of triples in S_1 and S_2 after triple identifiers have been changed to preserve uniqueness and literal references have been adjusted to be consistent with *Lit*.
   - The top triple of S is $i_{top} : i_1 \# \ i_2$, where i_1 and i_2 are the triples that correspond to the old top triples of S_1 and S_2.
   - $[e] = \{e\}$ for all entities e in S.

4.7.5 Boolean Quantification for m_{AX},m_{EX}

We implement boolean quantification by using the identity $\Pi p.\phi \equiv \phi[\top/p] \ @ \ \phi[\bot/p]$ for Π quantifier and @ either the disjunction and conjunction operator (see section 4.3). The substitution operations is simple to express using *subst*; the operator application is implemented by k-fusing.

We refer to the following algorithm as *k-quantify*:

1. Make two copies, sys_1 and sys_2, of the current formula system.
2. Construct $res_1 = subst(sys_1, \top, p)$ and $res_2 = subst(sys_2, \bot, p)$
3. k-fuse res_1 and res_2 under @

Our main concern is to stop the representation from growing too much as each boolean quantification without minimisation potentially doubles the formula size. The use of k-fuse will guarantee k-minimisation, and the systems will be in good position for reductions as at least two formula leaves are propositional constants.

4.7.6 Discussion

Efficiency. The presented algorithms will work well when the representation size can be kept within bounds using low degrees of saturation, and the formulas that arise during a verification are easy so that fixpoint termination can be determined.

However, if it is too hard to detect that a fixpoint has been reached, the model checker can just continue iterating and building new formulas (logically equivalent to the old); paradoxically the hardness degree of a formula can sometimes be reduced by building another, more redundant representation. Hardness of a formula can also be decreased by adding special subformulas that preserves the semantics [85]. Which formulas to add seems to be hard to deduce in general, but some heuristic could be possible as we are interested in formulas that arise in a particular way.

One indication that formula hardness not should be a general problem is the fact that experiments with bounded model checking using Stålmarck's method (see below) have generated formulas that are tractable even though the investigated systems have been very hard to verify with traditional methods.

The algorithms that have been presented in this section are primitive and can be refined and optimised. Minimisations can for example be done more often than the algorithms suggest (we have here minimised at least once in each primitive operation), or only when strictly needed. Algorithm complexity will however always be dominated by the cost of k-saturating, an algorithm that is $O(n^{2k+1})$ for n formula size; the remaining parts of the presented algorithms are linear.

Bounded Model Checking. Embedding model checking in propositional logic is not a novel idea; a very promising approach to linear time temporal logic (LTL) model checking is that taken by Ed Clarke and his colleagues [6] in their work on bounded model checking.

The main idea of bounded model checking (BMC) is to generate a single propositional logic formula that defines the set of length k paths that satisfies the temporal formula according to a suitably modified semantics. A test for PROP satisfiability is therefore enough to prove temporal satisfaction for bounded length paths. It turns out that there is always a finite k such that satisfiability for length k paths is necessary and sufficient to infer that there exists an infinite path that satisfies the temporal formula. A troublesome aspect of the bounded model checking approach is that a bound on k must be determined to guarantee that bounded satisfiability is equivalent to general satisfiability.

We, on the other hand, generate a sequence of formulas ϕ_k that analogously defines the set of states that satisfies a CTL temporal formula for k steps forward. The logical fixpoint of this sequence defines the set of states that satisfies the temporal property with respect to the standard semantics. Rather than producing a single, monolithic formula, however, we have the opportunity of minimising and using domain specific knowledge to guide the process inbetween each step. We can also avoid the issue of finding bounds on k since theorem proving is used to detect that a fixpoint has been reached.

Bounded model checking is closely related to our approach. We have however arrived at a similar solution by considering how sets rather than paths can be represented using logical formulas, which among other things means that we could avoid having to prove that a bound k really exists (we have only changed the representation of sets).

The experimental data presented for bounded model checking is encouraging, and suggests that embeddings into propositional logic can be very good for systems that are hard for BDDs. We believe that the fact that our approach can avoid reasoning about bounds, and that we have the possibility to do internal minimisation when the formulas grow too much could be very useful.

4.8 Other Logics

Up to now, we have been considering a representation that encodes a set as a propositional logic formula. But we are not forced to use propositional logic; any other logic of equal or greater strength is a possible candidate.

For stronger logics it is actually often possible to embed a model checking problem as a single formula that directly expresses the temporal property according to the normal formula semantics. We maintain that there are many benefits to solving a number of smaller subproblems generated by fixpoint calculations rather than approaching the problem monolithically and relinquishing all control to the theorem prover. If a fixpoint approach is taken, the model checking process is divided up into many steps; inbetween each step the process can be guided and the formulas transformed.

We will briefly illustrate the benefits of some other logics by discussing how efficiency could be improved by using QBF, and by indicating how FOL could allow model checking of more complex state spaces.

4.8.1 Quantified Boolean Formulas

Set representation in QBF might at first not seem any different from set representation in PROP as there exists a simple translation between the two logics.

The translation is however superpolynomial, so we can gain efficiency by allowing boolean quantifiers at the formula level throughout the process: For example, we can sidestep having to immediately fold out the quantifiers, which in some cases avoids generation of huge formulas.

Consider a formula $\phi \leftrightarrow \phi$, where ϕ contains a large number of boolean quantifiers. Syntactic comparison on the QBF level alone is enough to deduce that the formula is a tautology. However, if we are first obliged to transform the QBF formula into an equivalent PROP formula, a large amount of simplification would be necessary to keep the formula size under control.

QBF-level reasoning can also be used to move quantifiers around before a mapping to PROP is done. One example of this would be to push in quantifiers as far as possible by an anti-prenex transformation, or to reduce quantifiers in a different order than inside out. In this way existing PROP provers can be used as back-ends for QBF theorem proving.

We do not know of any dedicated QBF theorem provers, but any FOL prover that has special support for quantifications over finite domains is also capable of QBF level reasoning: Given that Π is a quantifier and that x not is free in ϕ, the QBF formula $\Pi p.\phi$ can be mapped to an equivalent finite domain FOL formula $\Pi x \in \{0,1\}.\phi[(x=1)/p]$.

4.8.2 A Simple Use of First Order Logic

First order logic, in contrast to propositional logic, is expressible enough to define many interesting infinite sets of states. Given that $k \in \mathbb{Z}$ and i_x is an element of a finite set of integer variables IVar, this makes it possible to model check Kripke structures labelled with both ordinary atomic propositions and statements of the form $i_x = k$. We define an extension to CTL, called iCTL, based on such extended Kripke structures:

In each state s, each $p \in AP$ is true or false and each $i \in \text{IVar}$ has an integer value. The atoms of iCTL, AP', are

$$AP \ \cup \ \{i_n = k : n < |\text{IVar}|, k \in \mathbb{Z}\} \ \cup \ \{i_n = i_m : n, m < |\text{IVar}|\}$$

Compound iCTL formula semantics are defined by a straight forward extension of the CTL semantics.

Given an extended Kripke structure M, we define in each state s

$$ival_n^s = (i_n = k) \ \text{ iff } \ M, s \models i_n = k$$

For convenience we again assume that the states are uniquely labelled, and identify the state s with the set

$$\{val_n^s : n < |\text{AP}|\} \cup \{ival_n^s : n < |\text{IVar}|\}$$

Under the assumption that Γ is a standard theory of first order arithmetic, recursive sets of states S are now representable by first order logic formulas ϕ that have propositions from AP', variables from IVar and for which it holds that

$$s \in S \text{ iff } \quad s, \Gamma \models \phi$$

Recursive binary relations can analogously be represented with formulas, just as in section 4.6.

No new operations are needed for iCTL model checking; m_{atom}, $m_{\neg}$, $m_{\#}$, m_{AX}, m_{EX}, m_{μ} and m_{ν} suffice. All of the operations, with the exception of the transition operators, can be defined in the same spirit as for CTL.

The problem with the transition operators is now that quantification over a state will result in quantifiers that both range over propositions and variables. Boolean quantifiers can be handled as before but the quantifiers that range over integer variables are not expandable as their domains are infinite. However, as we will use a first order logic theorem prover, these quantifiers can be left in to be removed by unification.

To implement iCTL in practice, both a minimiser and theorem prover for FOL is needed. One candidate for such a system would be the FOL extension to Stålmarck's method [64], with the algorithms of section 4.7 lifted to first order logic.

4.8.3 And More....

There is no need to stop at simple equalities as we did in the case of iCTL. Once the step has been taken to first order logic for set representation, any operations that are needed can be axiomatised.

Special logics such as monadic first order logic, Presburger arithmetic and PROP + finite domain arithmetic can also be the foundations for model checking tools with different characteristics, applicable to yet other kinds of systems. The efficiency of the underlying theorem prover and minimiser is really the only limiting factor.

4.9 Conclusions

Many different trade-offs can be made between representation size and the complexity of primitive model checking operations. Table 4.1 contains a number of approaches to verification of temporal logic specifications together with the complexity of operations relative to the size of the generated problem(s). Each of the verification methods has its own time/space complexity characteristic, and is therefore preferable for certain systems. For example, enumeration-based model checking has very simple primitive model checking operations but requires a linear amount of memory in the size of the represented sets. BDD

Verification approach	Complexity of operations
Enumeration	Polynomial
BDD fixpoint calculations	Variable ordering NP-complete
BMC	Single NP/coNP-complete problem
PROP fixpoint calculations	coNP-complete subproblems
QBF fixpoint calculations	PSPACE-complete subproblems
FOL fixpoint calculations	Possibly undecidable subproblems
Temporal logic theorem proving	Single PSPACE-complete problem

Table 4.1: Some approaches to verification of temporal logic specifications

based symbolic model checking require operations of medium complexity, but can represent many sets compactly.

The aim of this paper has been to present a new family of model checking algorithms with different tradeoffs, and to demonstrate some benefits of a theorem proving approach to an efficient method of temporal logic theorem proving.

Our next step will be to implement the algorithms we have described inside Prover, a commercial tool based on Stålmarcks method, and investigate what systems the approach is suitable for. We are eager to compare results with bounded model checking and investigate how the generalisations to other logics work out in practice.

Chapter 5

Symbolic Reachability Analysis Based on SAT-Solvers

The introduction of symbolic model checking using Binary Decision Diagrams (BDDs) has led to a substantial extension of the class of systems that can be algorithmically verified. Although BDDs have played a crucial role in this success, they have some well-known drawbacks, such as requiring an externally supplied variable ordering and causing space blowups in certain applications. In a parallel development, SAT-solving procedures, such as Stålmarck's method or the Davis-Putnam procedure, have been used successfully in verifying very large industrial systems. These efforts have recently attracted the attention of the model checking community resulting in the notion of bounded model checking. In this paper, we show how to adapt standard algorithms for symbolic reachability analysis to work with SAT-solvers. The key element of our contribution is the combination of an algorithm that removes quantifiers over propositional variables and a simple representation that allows reuse of subformulas. The result will in principle allow many existing BDD-based algorithms to work with SAT-solvers. We show that even with our relatively simple techniques it is possible to verify systems that are known to be hard for BDD-based model checkers.

This chapter was written together with Parosh Aziz Abdulla and Niklas Eén. It was published at the International Conference on Tools and Constructions for the Analysis of Systems 2000 [2].

5.1 Introduction

In recent years *model checking* [21, 79] has been widely used for algorithmic verification of finite-state systems such as hardware circuits and communication protocols. In model checking, the specification of the system is formulated as a temporal logical formula, while the implementation is described as a finite-state transition system. Early model-checking algorithms suffered from *state explosion*, as the size of the state space grows exponentially with the number of components in the system. One way to reduce state explosion is to use *symbolic model checking* [18, 67], where the transition relation is coded symbolically as a boolean expression, rather than explicitly as the edges of a graph. Symbolic model checking achieved its major breakthrough after the introduction of *Binary Decision Diagrams* (BDDs) [16] as a data structure for representing boolean expressions in the model checking procedure. An important property of BDDs is that they are canonical. This allows for substantial sub-expression sharing, often resulting in a compact representation. In addition, canonicity implies that satisfiability and validity of boolean expressions can be checked in constant time. However, the restrictions imposed by canonicity can in some cases lead to a space blowup, making memory a bottleneck in the application of BDD-based algorithms. There are examples of functions, for example multiplication, which do not allow sub-exponential BDD representations. Furthermore, the size of a BDD is dependent on the variable ordering which in many cases is hard to optimize, both automatically and by hand. BDD-based methods can typically handle systems with hundreds of boolean variables.

A related approach is to use satisfiability solvers, such as implementations of Stålmarck's method [92] and the Davis-Putnam procedure [100]. These methods have already been used successfully for verifying industrial systems [87, 14, 15, 93, 42]. SAT-solvers enjoy several properties which make them attractive as a complement to BDDs in symbolic model checking. For instance, their performance is less sensitive to the size of the formulas, and they can in some cases handle propositional formulas with thousands of variables. Furthermore, SAT-solvers do not suffer from space explosion, and do not require an external variable ordering to be supplied. Finally, satisfiability solving is an NP-complete problem, whereas BDD-construction solves a #P-complete problem [74] as it is possible to determine the number of models of a BDD in polynomial time. #P-complete problems are widely believed to be harder than NP-complete problems.

The aim of this work is to exploit the strength of SAT-solving procedures in order to increase the class of systems amenable to verification via the traditional symbolic methods. We consider modifications of two standard algorithms—forward and backward reachability analysis—where formulas are used to characterize sets of reachable states [9]. In these algorithms we replace BDDs by satisfiability checkers such as the Prover implementation of Stålmarck's method [92] or Sato [100]. We also use a data structure which we call *Reduced Boolean Circuits* (RBCs) to represent formulas. RBCs avoid unnecessarily large representations through the reuse of subformulas, and allow for efficient storage and manipulation of formulas. The only operation of the reachability algorithms that does not carry over straightforwardly to this representation is quantification over propositional variables. Therefore, we provide a simple procedure for the removal of quantifiers, which gives adequate performance for the examples we have tried so far.

We have implemented a tool FIXIT [32] based on our approach, and carried out a number of experiments. The performance of the tool indicates that even though we use simple techniques, our method can perform well in comparison to existing ones.

Related Work. *Bounded Model Checking* (BMC) [5, 6, 7] is the first approach in the literature to perform model checking using SAT-solvers. To check reachability, the BMC procedure searches for counterexamples (paths to undesirable states) by "unrolling" the transition relation k steps. The unrolling is described by a (quantifier-free) formula which characterizes the set of feasible paths through the transition relation with lengths smaller than or equal to k. The search can be terminated when the value of k is equal to the *diameter* of the system—the maximum length of all shortest path between states in the system. Although the diameter can be specified by a logical formula, its satisfiability is usually hard to check, making BMC incomplete in practice. Furthermore, for "deep" transition systems, formulas characterizing the set of reachable states may be much smaller than those characterizing witness paths. Since our method is based on encodings of sets of states, it may in some cases cope with systems which BMC fails to analyze as it generates formulas that are too large.

Our representation of formulas is closely related to *Binary Expression Diagrams* (BEDs) [3, 54]. In fact there are straightforward linear space translations back and forth between the representations. Consequently, RBCs share the good properties of BEDs, such as being exponentially more succinct than BDDs [3]. The main difference between our approach and the use of BEDs is the way in which satisfiability checking and existential quantification is handled. In [3], satisfiability of BEDs is checked through a translation to equivalent BDDs. Although many simplifications are performed at the BED level, converting to BDDs during a fixpoint iteration could cause degeneration into a standard BDD-based fixpoint iteration. In contrast, we check satisfiability by mapping RBCs back to formulas which are then fed to external SAT-solvers. In fact, the use of SAT-solvers can also be applied to BEDs, but this does not seem to have been explored so far. Furthermore, in the BED approach, existential quantification is either handled by introducing explicit quantification vertices, or by a special transformation that rewrites the representation into a form where naive expansion can be applied. We use a similar algorithm that also applies an extra inlining rule. The inlining rule is particularly effective in the case of backward reachability analysis, as it is always applicable to the generated formulas. To our knowledge, no results have been reported in the literature on applications

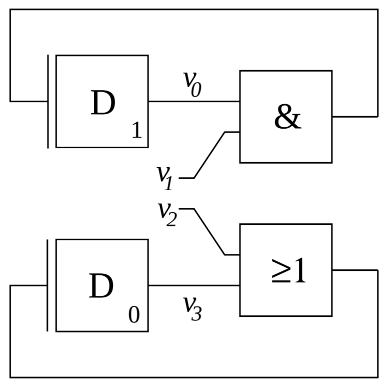

Figure 5.1: A simple circuit built from combinational gates and delays.

of BEDs in symbolic model checking. We would like to emphasize that we view RBCs as a relatively simple representation of formulas, and not as a major contribution of this work.

5.2 Preliminaries

We verify systems described as synchronous circuits constructed from elementary combinational gates and unit delays—a simple, yet popular, model of computation. The unit delays are controlled by a global clock, and we place no restriction on the inputs to a circuit. The environment is free to behave in any fashion.

We define the *state-holding elements* of a circuit to be the primary inputs and the contents of the delays, and define a *valuation* to be an assignment of boolean values to the state-holding elements. The behaviour of a circuit is modelled as a state-transition graph where (1) each valuation is a state; (2) the initial states comprise all states that agree with the initial values of the delays; and (3) there is a transition between two states if the circuit can move between the source state and the destination state in one clock cycle.

We construct a symbolic encoding of the transition graph in the standard manner. We assign every state-holding element a propositional state variable v_i, and make two copies of the set of state variables, $s = \{v_0, v_1, \ldots, v_k\}$ and $s' = \{v'_0, v'_1, \ldots, v'_k\}$. Given a circuit we can now generate two *characteristic formulas*. The first of the characteristic formulas, $Init(s) = \bigwedge_i v_i \leftrightarrow \phi_i$, defines the initial values of the state-holding elements. The second characteristic formula, $Tr(s, s') = \bigwedge_i v'_i \leftrightarrow \psi_i(s)$, defines the next-state values of state-holding elements in terms of the current-state values.

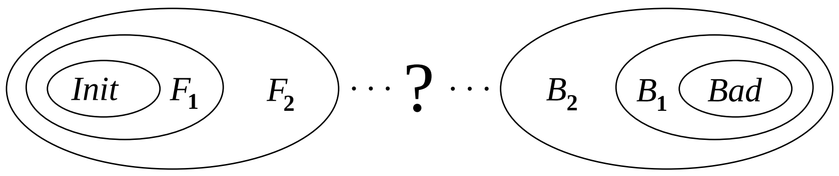

Figure 5.2: The intuition behind the reachability algorithms.

Example 1 The following formulas characterize the circuit in Figure 5.1:

$$\begin{aligned} Init &= (v_0 \leftrightarrow \top) \wedge (v_3 \leftrightarrow \bot) \\ Tr &= (v_0' \leftrightarrow (v_0 \wedge v_1)) \;\wedge\; (v_3' \leftrightarrow (v_2 \vee v_3)) \end{aligned}$$

□

We investigate the underlying state-transition graph by applying operations at the formula level. In doing so we make use of the following three facts. First, the relation between any points in a given circuit can be expressed as a propositional formula over the state-holding variables. Second, we can represent any set S of transition-graph states by a formula that is satisfied exactly by the states in S. Third, we can lift all standard set-level operations to operations on formulas (for example, set inclusion corresponds to formula-level implication and set nonemptiness checking to satisfiability solving, respectively).

5.3 Reachability Analysis

Given the characteristic formulas of a circuit and a formula $Bad(s)$, we define the *reachability problem* as that of checking whether it is possible to reach a state that satisfies $Bad(s)$ from an initial state. As an example, in the case of the circuit in Figure 5.1, we might be interested in whether the circuit could reach a state where the two delay elements output the same value (or equivalently, where the formula $v_0 \leftrightarrow v_3$ is satisfiable). We adapt two standard algorithms for performing reachability analysis. In *forward reachability* we compute a sequence of formulas $F_i(s)$ that characterize the set of states that the initial states can reach in i steps:

$$\begin{aligned} F_0(s) &= Init \\ F_{i+1}(s') &= toProp(\exists s.\ Tr(s,s') \wedge F_i(s))) \end{aligned}$$

Each computation of F_{i+1} gives rise to a *Quantified Boolean Formula* (QBF), which we translate back to a pure propositional formula using an operation *toProp* (defined in Section 5.5). We terminate the sequence generation if either (1) $F_n(s) \wedge Bad(s)$ is satisfiable: this means that a bad state is reachable; hence we answer the reachability problem positively; or (2) $\bigvee_{k=0}^{n} F_k(s) \rightarrow \bigvee_{k=0}^{n-1} F_k(s)$

holds: this implies that we have reached a fixpoint without encountering a bad state; consequently the answer to the reachability question is negative.

In *backward reachability* we instead compute a sequence of formulas $B_i(s)$ that characterize the set of states that can reach a bad state in i steps:

$$\begin{aligned} B_0(s) &= Bad \\ B_{i+1}(s) &= \mathit{toProp}(\exists s'.\, Tr(s,s') \wedge B_i(s'))) \end{aligned}$$

In a similar manner to forward reachability, we terminate the sequence generation if either (1) $B_n(s) \wedge Init(s)$ is satisfiable, or (2) $\bigvee_{k=0}^{n} B_k(s) \rightarrow \bigvee_{k=0}^{n-1} B_k(s)$ holds.

Figure 5.2 shows the intuition behind the algorithms. We remark that the two reachability methods can be combined by alternating between the computation of F_{i+1} and B_{i+1}. The generation can be terminated when either a fixpoint is reached in some direction, or when F_n and B_n intersect. However, we do not make use of hybrid analyses in this paper.

We need to address three nontrivial issues in an implementation of the adapted reachability algorithms. First, we must avoid the generation of unnecessarily large formula characterizations of the sets F_i and B_i—formulas are not a canonical representation. Second, we must define the operation *toProp* in such a way that it translates quantified boolean formulas to propositional logic without needlessly generating exponential results. Third, we must interface efficiently to external satisfiability solvers. The remainder of the paper explains our solutions, and evaluates the resulting reachability checker.

5.4 Representation of Formulas

Let **Bool** denote the set of booleans; **Vars** denote the set of propositional variables, including a special variable $\top$ for the constant *true*; and **Op** denote the set $\{\leftrightarrow, \wedge\}$.

We introduce the representation *Boolean Circuit* (BC) for propositional formulas. A BC is a directed acyclic graph, $(\mathbf{V}, \mathbf{E})$. The vertices $\mathbf{V}$ are partitioned into internal nodes, $\mathbf{V_I}$, and leaves, $\mathbf{V_L}$. The vertices and edges are given attributes as follows:

- Each internal vertex $v \in \mathbf{V_I}$ has three attributes: A binary operator $op(v) \in \mathbf{Op}$, and two edges $\mathit{left}(v), \mathit{right}(v) \in \mathbf{E}$.
- Each leaf $v \in \mathbf{V_L}$ has one attribute: $var(v) \in \mathbf{Vars}$.
- Each edge $e \in \mathbf{E}$ has two attributes: $sign(e) \in \mathbf{Bool}$ and $target(e) \in \mathbf{V}$.

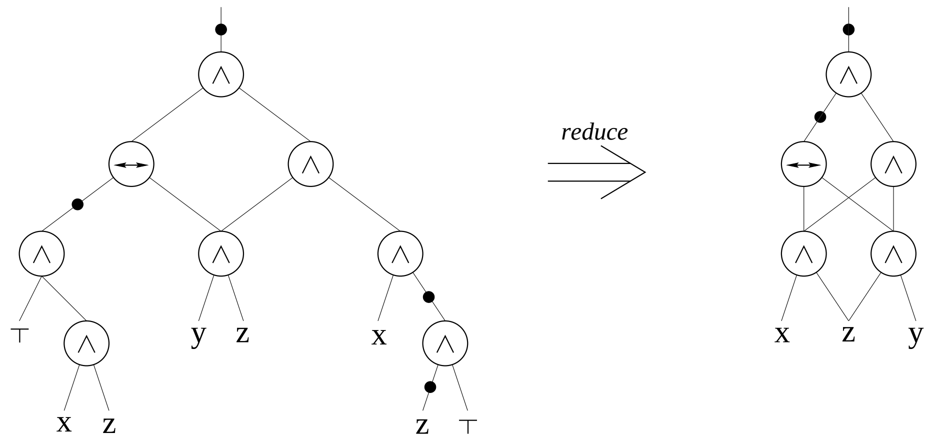

Figure 5.3: A non-reduced *Boolean Circuit* and its reduced form.

We observe that negation is coded into the edges of the graph, by the *sign* attribute. Furthermore, we identify *edges* with *subformulas*. In particular, the whole formula is identified with a special top-edge having no source vertex. The interpretation of an edge as a formula is given by the standard semantics of $\wedge$, $\leftrightarrow$ and $\neg$ by viewing the graph as a parse tree (with some common sub-expressions shared). Although $\wedge$ and $\neg$ are functionally complete, we choose to include $\leftrightarrow$ in the representation as it would otherwise require three binary connectives to express. Figure 5.3 shows an example of a BC.

A *Reduced Boolean Circuit* (RBC) is a BC satisfying the following properties:

1. All common subformulas are shared so that no two vertices have identical attributes.
2. The constant $\top$ never occurs in an RBC, except for the single-vertex RBCs representing *true* or *false*.
3. The children of an internal vertex are syntactically distinct, $\mathit{left}(v) \neq \mathit{right}(v)$.
4. If $\mathit{op}(v) = \leftrightarrow$ then the edges to the children of v are unsigned.
5. For all vertices v, $\mathit{left}(v) < \mathit{right}(v)$, for some total order $<$ on BCs.

The purpose of these constraints is to identify as many equivalent formulas as possible, and thereby increase the amount of subformula sharing. For this reason we allow only one representation of $\neg(\phi \leftrightarrow \psi) \iff (\neg\phi \leftrightarrow \psi)$ (in 4 above), and $(\phi \wedge \psi) \iff (\psi \wedge \phi)$ (in 5 above).

The RBCs are created in an implicit environment, where all existing subformulas are tabulated. We use the environment to assure property (1). Figure 5.4 shows

```
reduce(AND, left ∈ RBC, right ∈ RBC)            reduce(EQUIV, left ∈ RBC, right ∈ RBC)
    if   (left = right)   return left               if   (left = right)   return ⊤
    elif (left = ¬right)  return ⊥                  elif (left = ¬right)  return ⊥
    elif (left = ⊤)       return right              elif (left = ⊤)       return right
    elif (right = ⊤)      return left               elif (left = ⊥)       return ¬right
    elif (left = ⊥)       return ⊥                  elif (right = ⊤)      return left
    elif (right = ⊥)      return ⊥                  elif (right = ⊥)      return ¬left
    else                  return NIL                else                  return NIL


mk_Comp(op ∈ Op, left ∈ RBC, right ∈ RBC, sign ∈ Bool)
    result := reduce(op, left, right)
    if (result ≠ NIL)
        return id(result, sign)     – id returns result or ¬result depending on sign

    if (right < left)
        (left, right) := (right, left)     – Swap the values of left and right

    if (op = EQUIV)
        sign  := sign xor sign(left) xor sign(right)
        left  := unsigned(left)
        right := unsigned(right)

    result := lookup(RBC_env, (op, left, right))     – Look for vertex in environment
    if (result = NIL)
        result := insert(RBC_env, (op, left, right))
    return id(result, sign)
```

Figure 5.4: Pseudo-code for creating a composite RBC from two existing RBCs.

the only non-trivial constructor for RBCs, *mk_Comp*, which creates a composite RBC from two existing RBCs (we use $x \in \text{Vars}(\phi)$ to denote that x is a variable occurring in the formula ϕ). It should be noted that the above properties only takes constant time to maintain in *mk_Comp*.

5.5 Quantification

In the reachability algorithms we make use of the operation *toProp* to translate QBF formulas into equivalent propositional formulas. We reduce the translation of a set of existential quantifiers to the iterated removal of a single quantifier after we have chosen a quantification order. In the current implementation an arbitrary order is used, but we are evaluating more refined approaches.

Figure 5.5 presents the quantification algorithm of our implementation. By definition we have:

$$\exists x \,.\, \phi(x) \iff \phi(\bot) \vee \phi(\top) \qquad (*)$$

The definition can be used to naively resolve the quantifiers, but this may yield an exponential blowup in representation size. To try to avoid this, we use the following well-known identities (applied from left to right) whenever possible:

Inlining:

$$\exists x \;.\; (x \leftrightarrow \psi) \wedge \phi(x) \quad\Longleftrightarrow\quad \phi(\psi) \qquad \text{(where } x \notin \mathrm{Vars}(\psi))$$

Scope Reduction:

$$\exists x \;.\; \phi(x) \wedge \psi \quad\Longleftrightarrow\quad (\exists x.\phi(x)) \wedge \psi \qquad \text{(where } x \notin \mathrm{Vars}(\psi))$$
$$\exists x \;.\; \phi(x) \vee \psi(x) \quad\Longleftrightarrow\quad (\exists x.\phi(x)) \vee (\exists x.\psi(x))$$

When applicable, *inlining* is an effective method of resolving quantifiers as it immediately removes all occurrences of the quantified variable x. The applicability of the transformation relies on the fact that the formulas occurring in reachability often have a structure that matches the rule. This is particularly true for backward reachability as the transition relation is a conjunction of next state variables defined in terms of current state variables $\bigwedge_i v_i' \leftrightarrow \psi_i(s)$.

The first step of the inlining algorithm temporarily changes the representation of the top-level conjunction. From the binary encoding of the RBC, we extract an equivalent set representation $\bigwedge\{\phi_0, \phi_1, \dots, \phi_n\}$. If the set contains one or more elements of the form $x \leftrightarrow \psi$, the smallest such element is removed from the set and its right-hand side ψ is substituted for x in the remaining elements. The set is then re-encoded as an RBC.

If inlining is not applicable to the formula (and variable) at hand, the translator tries to apply the *scope reduction* rules as far as possible. This may result in a quantifier being pushed through an OR (represented as negated AND), in which case inlining may again be possible.

For subformulas where the scope can no longer be reduced, and where inlining is not applicable, we resort to *naive quantification* (*). Reducing the scope as much as possible before doing this will help prevent blowups. Sometimes the quantifiers can be pushed all the way to the leaves of the RBC, where they can be eliminated.

Throughout the quantification procedure, we may encounter the same subproblem more than once due to shared subformulas. For this reason we keep a table of the results obtained from all previously processed subformulas.

5.6 Satisfiability

Given an RBC, we want to decide whether there exists a satisfying assignment for the corresponding formula by applying an external SAT-solver. The naive translation—unfold the graph to a tree and encode the tree as a formula—has the drawback of removing sharing. We therefore use a mapping where each internal node in the representation is allocated a fresh variable. This variable is used in place of the subformula that corresponds to the internal node. The

```
– Global variable processed tabulates the results of the performed quantifications.

quant_naive($\phi \in \mathbf{RBC}$, $x \in \mathbf{Vars}$)
        result = $subst(\phi, x, \bot) \vee subst(\phi, x, \top)$
        insert(processed, $\phi$, $x$, result)
        return result

quant_reduceScope($\phi \in \mathbf{RBC}$, $x \in \mathbf{Vars}$)
        if ($x \notin \mathrm{Vars}(\phi)$) return $\phi$
        if ($\phi = x$)          return $\top$

        result := lookup(processed, $\phi$, $x$)
        if (result $\neq$ NIL)
            return result

        – In the following $\phi$ must be composite and contain $x$:
        if ($\phi_{op}$ = EQUIV)
            result := quant_naive($\phi$, $x$)
        elif (not $\phi_{sign}$)     – Operator AND, unsigned
            if   ($x \notin \mathrm{Vars}(\phi_{left})$)  result := $\phi_{left} \wedge$ quant_reduceScope($\phi_{right}$, $x$)
            elif ($x \notin \mathrm{Vars}(\phi_{right})$) result := quant_reduceScope($\phi_{left}$, $x$) $\wedge\ \phi_{right}$
            else                      result := quant_naive($\phi$, $x$)
        else    – Operator AND, signed ("OR")
            result := quant_inline($\neg\phi_{left}$, $x$) $\vee$ quant_inline($\neg\phi_{right}$, $x$)

        insert(processed, $\phi$, $x$, result)
        return result

quant_inline($\phi \in \mathbf{RBC}$, $x \in \mathbf{Vars}$)    – "Main"
        $C$ := collectConjuncts($\phi$)      –  Merge all binary ANDs at the top of $\phi$ into a
                                                "big" conceptual conjunction (returned as a set).
        $\psi$ := findDef($C$, $x$)           –  Return the smallest formula $\psi$ such that $(x \leftrightarrow \psi)$
                                                is a member of $C$.

        if ($\psi \neq$ NIL)
            $C'$ := $C \setminus (x \leftrightarrow \psi)$                      – Remove definition from $C$.
            return subst(makeConj($C'$), $x$, $\psi$)         – makeConj builds an RBC.
        else
            return quant_reduceScope($\phi$, $x$)
```

Figure 5.5: Pseudo-code for performing existential quantification over one variable. By ϕ_{left} we denote *left*(*target*(ϕ)) etc. We use $\wedge$, $\vee$ as abbreviations for calls to *mk_Comp*.

generated formula is the conjunction of all the definitions of internal nodes and the literal that defines the top edge.

Example 2 The right-hand RBC in Figure 5.3 is mapped to the following formula in which the i_k variables define internal RBC nodes:

$$\begin{aligned}
&(i_0 \leftrightarrow \neg i_1 \land\ i_2)\\
\land\ &(i_1 \leftrightarrow i_3 \leftrightarrow i_4)\\
\land\ &(i_2 \leftrightarrow i_3 \land i_4)\\
\land\ &(i_3 \leftrightarrow x \land z)\\
\land\ &(i_4 \leftrightarrow z \land y)\\
\land\ &\neg i_0
\end{aligned}$$

□

A formula resulting from the outlined translation is *not* equivalent to the original formula without sharing, but it will be satisfiable if and only if the original formula is satisfiable. Models for the original formula are obtained by discarding the values of internal variables.

5.7 Experimental Results

We have implemented a tool FIXIT [32] for performing symbolic reachability analysis based on the ideas presented in this paper. The tool has a *fixpoint mode* in which it can perform both forward and backward reachability analysis, and an *unroll mode* where it searches for counterexamples in a similar manner to the BMC procedure. We have carried out preliminary experiments on three benchmarks: a *multiplier* and a *barrel shifter* (both from the BMC distribution), and a *swapper* (defined by the authors). The first two benchmarks are known to be hard for BDD-based methods.

In all the experiments, PROVER outperforms SATO, so we only present measurements made using PROVER. Furthermore, we only present time consumption. Memory consumption is much smaller than for BDD-based systems. Garbage collection has not yet been implemented in FIXIT, but the amount of simultaneously referenced memory peaks at about 5-6 MB in our experiments. We also know that the memory requirements of PROVER are relatively low (worst case quadratic in the formula size). The test results for FIXIT are compared with results obtained from VIS release 1.3, BMC version 1.0f and CADENCE SMV release 09-01-99.

The Multiplier. The example models a standard 16×16 bit shift-and-add multiplier, with an output result of 32 bits. Each output bit is individually verified against the C6288 combinational multiplier of the ISCAS'85 benchmarks by checking that we cannot reach a state where the computation of the shift-and-add multiplier is completed, but where the selected result bit is not consistent with the corresponding output bit of the combinational circuit.

Bit	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	VIS sec	SMV sec
0	0.8	2.0	0.7	1.0	5.3	41.4
1	0.9	2.3	0.7	1.4	5.4	41.3
2	1.1	3.0	0.8	2.0	5.3	42.5
3	1.8	3.9	0.9	4.0	5.5	42.6
4	3.0	6.1	1.2	8.2	6.2	[>450 MB]
5	7.2	9.9	1.8	19.9	10.2	–
6	24.3	21.5	3.8	66.7	32.9	–
7	100.0	61.9	11.8	304.6	153.5	–
8	492.8	224.7	45.2	1733.7	[>450 MB]	–
9	2350.6	862.6	197.8	9970.8	–	–
10	11927.5	3271.0	862.8	54096.8	–	–
11	60824.6	13494.3	3838.0	–	–	–
12	–	50000.0	16425.8	–	–	–

Table 5.1: Experimental results for the multiplier.

Table 5.1 presents the results for the multiplier. The SAT-based methods outperform both VIS and SMV. The unroll mode is a constant factor more efficient than the fixpoint mode. However, we were unable to prove the diameter of the system by the diameter formula generated by BMC, which means that the verification performed by the unroll method (and BMC) should be considered partial.

The Barrel Shifter. The barrel shifter rotates the contents of a register file R with one position in each step. The system also contains a fixed register file R_0, related to R in the following way: if two registers from R and R_0 have the same contents, then their neighbours also have the same contents. We constrain the initial states to have this property, and the objective is to prove that it holds throughout the reachable part of the state space. The width of the registers is $\log|R|$ bits, and we let the BMC tool prove that the diameter of the circuit is $|R|$.

Table 5.2 presents the results for the barrel shifter. No results are presented for VIS due to difficulties in describing the extra constraint on the initial state in the VIS input format.

The backward reachability mode of FixIt outperforms SMV and BMC on this example. The reason for this is that the set of bad states is closed under the pre-image function, and hence FixIt terminates after only one iteration. SMV is unable to build the BDDs characterising the circuits for larger problem instances. The BMC tool has to unfold the system all the way up to the diameter, producing very large formulas; in fact, the version of BMC that we used could not generate formulas for larger instances than size 17 (a size 17 formula is 2.2 MB large). The oscillating timing data for the SAT-based tools reflects the heuristic nature of the underlying SAT-solver.

$\lvert R \rvert$	$^{\text{FixIt}}$Fwd	$^{\text{FixIt}}$Bwd	$^{\text{FixIt}}$Unroll	BMC	Diam	SMV
	sec	sec	sec	sec	sec	sec
2	1.7	0.1	0.1	0.0	0.0	0.0
3	2.3	0.1	0.1	0.0	0.0	0.1
4	3.0	0.1	0.2	0.0	0.0	0.1
5	42.4	0.2	0.3	0.1	0.1	44.2
6	848.9	0.2	0.5	0.3	0.1	[>450 MB]
7	5506.6	0.4	0.5	0.4	0.2	–
8	[>3 h]	0.5	1.0	1.2	0.3	–
9	–	0.8	1.6	2.4	0.6	–
10	–	1.1	2.3	8.6	0.8	–
11	–	1.5	2.3	3.3	1.1	–
12	–	2.3	4.1	25.6	1.5	–
13	–	2.6	3.9	7.1	2.0	–
14	–	3.2	7.8	80.1	2.6	–
15	–	3.7	8.6	75.1	3.5	–
16	–	4.3	12.1	150.0	4.4	–
17	–	6.7	11.0	34.6	7.9	–
18	–	8.7	30.5	?	?	–
19	–	9.2	15.6	?	?	–
20	–	13.5	49.1	?	?	–
⋮						
30	–	51.4	452.1	?	?	–
⋮						
40	–	230.5	2294.7	?	?	–
⋮						
50	–	501.5	8763.3	?	?	–

Table 5.2: Experimental results for the barrel shifter.

The Swapper. N nodes, each capable of storing a single bit, are connected linearly:

$$\boxed{1}-\boxed{2}-\boxed{3}-\boxed{4}-\cdots-\boxed{N}$$

At each clock-cycle (at most) one pair of adjacent nodes may swap their values. From this setting we ask whether the single final state in which exactly the first $\lfloor N/2 \rfloor$ nodes are set to 1 is reachable from the single initial state in which exactly the last $\lfloor N/2 \rfloor$ nodes are set to 1. Table 5.3 shows the result of verifying this property.

Both VIS and SMV handle the example easily. FixIt can handle sizes up to 14, but does not scale up as well as VIS and SMV, as the representations get too large. This illustrates the importance of maintaining a compact representation during deep reachability problems; something that is currently not done by FixIt. However, BMC does even worse, even though the problem is a strict search for an existing counterexample—something BMC is generally good at. This shows that fixpoint methods can be superior both for proving unreachability and detecting counterexamples for certain classes of systems.

N	$^{\text{FixIt}}$Fwd	$^{\text{FixIt}}$Bwd	$^{\text{FixIt}}$Unroll	BMC	VIS	SMV
	sec	sec	sec	sec	sec	sec
3	0.2	0.2	0.2	0.0	0.3	0.0
4	0.3	0.3	0.2	0.0	0.3	0.0
5	0.6	0.5	0.3	0.1	0.3	0.0
6	0.9	1.5	1.8	7.2	0.4	0.1
7	1.7	3.7	131.2	989.5	0.4	0.1
8	3.8	10.4	[>2 h]	[>2 h]	0.4	0.1
9	9.7	58.9	–	–	0.4	0.1
10	27.7	187.1	–	–	0.4	0.1
11	74.1	779.2	–	–	0.5	0.2
12	238.8	4643.2	–	–	0.6	0.2
13	726.8	[>2 h]	–	–	0.7	0.3
14	2685.7	–	–	–	0.7	0.4
15	[>2 h]	–	–	–	0.7	0.6
⋮						
20	–	–	–	–	1.6	7.9
⋮						
25	–	–	–	–	3.3	53.0
⋮						
30	–	–	–	–	15.1	263.0
⋮						
35	–	–	–	–	39.1	929.6
⋮						
40	–	–	–	–	89.9	2944.3

Table 5.3: Experimental results for the swapper.

5.8 Conclusions and Future Work

We have described an alternative approach to standard BDD-based symbolic model checking which we think can serve as a useful complement to existing techniques. We view our main contribution as showing that with relatively simple means it is possible to modify traditional algorithms for symbolic reachability analysis so that they work with SAT-procedures instead of BDDs. The resulting method gives surprisingly good results on some known hard problems.

SAT-solvers have several properties which make us believe that SAT-based model checking will become an interesting complement to BDD-based techniques. For example, in a proof system like Stålmarck's method, formula size does not play a decisive role in the hardness of satisfiability checking. This is particularly interesting since industrial applications often give rise to formulas which are extremely large in size, but not necessarily hard to prove.

There are several directions for future work. We are currently surveying simplification methods that can be used to maintain compact representations. One promising approach [3] is to improve the local reduction rules to span over multiple levels of the RBC graphs. We are also interested in exploiting the structure of big conjunctions and disjunctions, and in simplifying formulas using algorithms based on Stålmarck's notion of formula saturation [9]. As for the representation itself, we are considering adding *if-then-else* and substitution nodes [54]. Other ongoing work includes experiments with heuristics for choosing good quantification orderings.

In the longer term, we will continue to work on conversions of BDD-based algorithms. For example, we have already implemented a prototype model checker for general (fair) CTL formulas. Also, employing traditional BDD-based model checking techniques such as front simplification and approximate analysis are very likely to improve the efficiency of SAT-based model checking significantly.

Many important questions related to SAT-based model checking remain to be answered. For example, how should the user choose between bounded and fixpoint-based model checking? How can SAT-based approaches be combined with standard approaches to model checking?

Acknowledgements

The implementation of FixIt was done as a Master's thesis at Prover Technology, Stockholm. Thanks to Purushothaman Iyer, Bengt Jonsson, Gordon Pace, Mary Sheeran and Gunnar Stålmarck for giving valuable feedback on earlier drafts.

This research was partially supported by TFR, the ASTEC competence center for advanced software technology, and the ARTES network for real-time research and graduate education in Sweden.

Chapter 6

SAT-based Verification without State Space Traversal

Binary Decision Diagrams (BDDs) have dominated the area of symbolic model checking for the past decade. Recently, the use of satisfiability (SAT) solvers has emerged as an interesting complement to BDDs. SAT-based methods are capable of coping with some of the systems that BDDs are unable to handle.

The most challenging problem that has to be solved in order to adapt standard symbolic model checking to SAT-solvers is the boolean quantification necessary for traversing the state space. A possible approach to extending the applicability of SAT-based model checkers is therefore to reduce the amount of traversal.

In this paper, we investigate a BDD-based verification algorithm due to van Eijk. Van Eijk's algorithm tries to compute information that is sufficient to prove a given safety property directly. When this is not possible, the computed information can be used to reduce the amount of traversal needed by standard model checking algorithms. We convert van Eijk's algorithm to use a SAT-solver instead of BDDs. We also make a number of improvements to the original algorithm, such as combining it with recently developed variants of induction. The result is a collection of substantially strengthened and complete verification methods that do not require state space traversal.

This chapter was written together with Koen Claessen. It was published at the International Conference on Formal Methods in Computer Aided Design 2000 [10].

6.1 Introduction

Symbolic model checking based on satisfiability (SAT) solvers [6, 2, 99, 86] has recently emerged as an interesting complement to model checking with Binary Decision Diagrams (BDDs) [16]. There are a number of systems which are not suited to be effectively verified using BDD-based model checkers, but can be verified using SAT-based methods. The use of SAT-solvers rather than BDDs also has advantages such as freeing the user from providing good variable orderings, and making the number of variables in the system less of a bottleneck. However, the boolean quantification that is necessary for computing characterisations for sets of predecessors (and successors) of states can sometimes lead to excessively large formulas in SAT adaptions of standard model checking algorithms.

In the hope of alleviating these problems, we investigate a BDD-based algorithm due to van Eijk [33] that attempts to verify safety properties of circuits without performing state-space traversal. The main idea behind the algorithm is to use induction to cheaply compute points in the circuit that always have the same value (or always have opposite values) in the reachable state space. This information sometimes directly implies the safety properties. If such a direct proof is not possible, the computed information can be used to decrease the number of necessary fixpoint iterations in backwards reachability algorithms. Van Eijk [33] has used the algorithm to directly prove equivalence between the original circuits and synthesised and optimised versions of 24 of the 26 circuits in the ISCAS'89 benchmark suite.

We are specifically interested in using van Eijk's algorithm to prove safety properties of circuits that are hard to represent using BDDs. Also, when a direct proof is not possible, we want to use the computed information to reduce the amount of state space traversal in exact SAT-based model checking methods as this could decrease the amount of necessary quantification drastically. As a consequence, we want to find alternatives to the use of BDDs in the original analysis. Van Eijk's algorithm also has the drawback of always computing the *largest* possible set of equivalences, even when this is not needed for the verification of the particular safety property at hand. In some cases this can become too costly; we would therefore like to be able to control how much work we put into finding equivalences.

We solve the two problems by converting the algorithm to use propositional formulas to represent points in the circuit, and by applying Stålmarck's saturation algorithm [92, 87] rather than BDDs for discovering equivalences between points.

The resulting algorithm is generalised in three ways. First, we make the algorithm complete by changing the induction scheme that is used in the method to some recently developed stronger variants of induction [86]. Second, we modify the algorithm to also discover implications between points in the circuit. Third, we demonstrate that van Eijk's algorithm can be viewed as an approximate forwards reachability analysis, and use this insight to construct the dual approximate backwards reachability algorithm and a mutual improvement algorithm.

The information that is computed by the resulting algorithms can in principle be used together with any BDD- or SAT-based model checking method. We show some benchmarks that demonstrate that the methods on their own can be very powerful tools for checking safety properties. For example, we use the algorithms to verify a non-trivial industrial example that previously has been out of reach for the SAT-based model checker FixIt [2].

6.2 Van Eijk's Method

In this section, we describe van Eijk's method [33]. In the original paper it is presented as a method for equivalence checking of sequential synchronous circuits. However, while using the method we have observed that it can work well also for general safety property verification.

Basic idea. The idea behind van Eijk's algorithm is to find the points in the circuit which have the same value (or have opposite values) in all reachable states. This information can then be used to either directly prove the safety property or to strengthen other verification methods.

The information is represented as an equivalence relation over the points of the circuit and their negations. The algorithm computes such an equivalence relation by means of a fixed point iteration. It starts with the equivalence relation that necessarily holds between the points in the initial state. Then it improves the relation by assuming that the equivalences hold at one time instance and deriving the subset of these equivalences that must hold in the next time instance. After a number of consecutive improvements, a fixed point is reached. The resulting equivalence relation is satisfied by the initial states, and is moreover preserved by any circuit transition. Therefore, it must hold in all reachable states.

Before we give a more precise description of van Eijk's algorithm, we first introduce some definitions.

Formulas. We describe the systems we are dealing with using propositional logic formulas. These are syntactic objects, built from variables like x and y, boolean values 1 and 0, and connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$. We say that a formula is *valid* if and only if it evaluates to 1 for all variable assignments under the usual interpretation of the connectives.

State machines. We represent sequential synchronous circuits as state machines in the standard way [22], where the set of states is the set of boolean valuations of a vector s of variables; one variable for each input and internal latch. As we do not restrict the input part of the states, these state machines are non-deterministic. The standard representation also guarantees that every state has at least one outgoing transition.

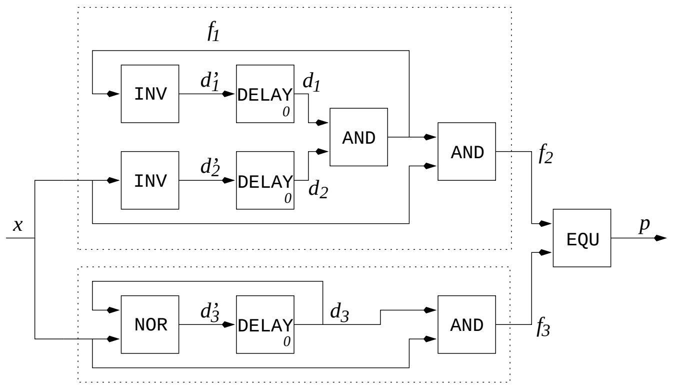

Figure 6.1: An example circuit

We characterise the set of initial states and the transition relation of the state machine by the propositional logic formulas $\mathit{Init}(s)$ and $\mathit{Trans}(s, s')$, respectively. In other words, $\mathit{Init}(s)$ is satisfied exactly by the initial states, and $\mathit{Trans}(s, s')$ is satisfied precisely when there is a transition between the states s and s'. The safety property of the system we want to verify is represented by the formula $\mathit{Prop}(s)$.

Example 3 Assume that we want to decide whether the two subcircuits in Figure 6.1 are equivalent. This amounts to checking whether the signal p is always true. Let us construct the necessary formulas. There are four state variables—one for every input and one for every delay component—so $s = (x, d_1, d_2, d_3)$. Since the delay components have an initial value of 0, the formula for the initial states becomes:

$$\mathit{Init}(x, d_1, d_2, d_3) \quad = \quad \neg d_1 \wedge \neg d_2 \wedge \neg d_3$$

Looking at the logic contained in the circuit, we can write down the formula for the transition relation:

$$\begin{aligned} &\mathit{Trans}(x, d_1, d_2, d_3, x', d_1', d_2', d_3') \quad = \\ &\qquad (d_1' \Leftrightarrow \neg(d_1 \wedge d_2)) \;\wedge\; (d_2' \Leftrightarrow \neg x) \;\wedge\; (d_3' \Leftrightarrow \neg(x \vee d_3)) \end{aligned}$$

Lastly, we define the formula for the property p:

$$\mathit{Prop}(x, d_1, d_2, d_3) \quad = \quad (d_1 \wedge d_2 \wedge x) \;\Leftrightarrow\; (d_3 \wedge x)$$

□

Signals. Given the formulas that characterise a state machine, we define the set Signals that models the points in the circuit. The elements of Signals are functions taking state variable vectors to formulas instantiated with these variables. Specifically, for every subformula $f(s)$ of the system formulas $\mathit{Trans}(s, s')$ and $\mathit{Prop}(s)$, such that $f(s)$ is not dependent on any of the variables of s', we add the corresponding functions f and $\neg f$ to the set. Moreover, we also add the constant signals tt and ff, for which $\mathsf{tt}(s) = 1$ and $\mathsf{ff}(s) = 0$.

> **Example 4** f_1 is a signal in Figure 6.1 with the definition $f_1(x, d_1, d_2, d_3) = d_1 \wedge d_2$. The negated signal $\neg f$ has the definition $\neg f_1(x, d_1, d_2, d_3) = \neg(d_1 \wedge d_2)$. □

Signal correspondence. For a given equivalence relation $\equiv$ over the set $\mathsf{Signals}$, we define the *signal correspondence condition*, denoted by $\mathsf{Holds}(\equiv, s)$, as follows:

$$\mathsf{Holds}(\equiv, s) \;=\; \bigwedge_{f \equiv g} f(s) \Leftrightarrow g(s).$$

This means that the correspondence condition for an equivalence relation is satisfied by a state when all the signals that are equivalent have the same value in that state. We define a *signal correspondence relation* as an equivalence relation whose correspondence condition holds in all reachable states.[1]

Algorithm. In order to find a signal correspondence relation, van Eijk's algorithm computes a sequence of equivalence relations $\equiv_i$, each being a better overapproximation of the desired relation. The sequence stops when an n is found such that $\equiv_n$ is equal to $\equiv_{n+1}$.

The first approximation $\equiv_0$ is the equivalence relation that holds in the initial states. We can define it as follows; for all f and g, $f \equiv_0 g$ if and only if the following formula is valid:

$$\mathit{Init}(s_1) \quad \Rightarrow \quad f(s_1) \Leftrightarrow g(s_1).$$

This means that two signals are equivalent precisely if they must have the same value in all initial states. The original algorithm computes $\equiv_0$ by constructing a BDD for every signal, and pairwise comparing these BDDs under the assumption that the BDD for $\mathit{Init}(s)$ holds.

The other approximations $\equiv_{n+1}$ for $n \geq 0$ are subsets of $\equiv_n$. We can define them as follows; for all f and g, $f \equiv_{n+1} g$ if and only if $f \equiv_n g$ and the following formula is valid:

$$\mathsf{Holds}(\equiv_n, s_1) \wedge \mathit{Trans}(s_1, s_2) \quad \Rightarrow \quad f(s_2) \Leftrightarrow g(s_2).$$

This means that two signals are equivalent in the new relation, when (1) they were equivalent in the old relation and (2) they have the same value in the next state if the old relation holds in the current state. The original algorithm computes $\equiv_{n+1}$ by pairwise comparison of the BDDs for the signals related by $\equiv_n$ under the assumption that the BDD for $\equiv_{n+1}$ holds.

[1] Note that this is a slight generalisation of van Eijk's original definition [33].

```
1.  $\equiv_1, \equiv_2$ := $\emptyset, \emptyset$ ;
2.  -- compute first approximation
3.  for every $f, g$ in Signals do
4.      form := $Init(s_1) \Rightarrow (f(s_1) \Leftrightarrow g(s_1))$ ;
5.      if (validBdd(form)) then
6.          set $f \equiv_2 g$ ;
7.  -- iterate until a fixed point is reached
8.  while ($\equiv_1 \neq \equiv_2$) do
9.      $\equiv_1, \equiv_2$ := $\equiv_2, \emptyset$ ;
10.     for every $f, g$ in Signals such that $f \equiv_1 g$ do
11.         form := $\mathsf{Holds}(\equiv_1, s_1) \wedge Trans(s_1, s_2) \Rightarrow (f(s_2) \Leftrightarrow g(s_2))$ ;
12.         if (validBdd(form)) then
13.             set $f \equiv_2 g$ ;
14. return $\equiv_1$ ;
```

Figure 6.2: Van Eijk's algorithm

The construction of approximations $\equiv_i$ has the shape of an *inductive* argument; it has a base case and a step that is iterated until it is provable. The final signal correspondence relation therefore holds in all reachable states.

For a schematic overview of the algorithm, see Figure 6.2. At lines 5 and 12, we use the function validBdd that checks if a formula is valid by building its BDD. At lines 6 and 13, we use **set** to modify an equivalence relation by merging the equivalence classes for f and g.

Example 3 (ctd.). The signal correspondence relation found by the algorithm for Example 3 looks as follows:

$$\{\ldots, (f_1, d_3), (f_2, f_3), (p, \mathsf{tt}), \ldots\}$$

From this information it follows immediately that the property p is always true. □

Remarks. The signal correspondence relation found by the algorithm sometimes implies the safety property directly. If this is not the case, then we can strengthen the transition formula $Trans(s, s')$ to a new formula $Trans(s, s') \wedge \mathsf{Holds}(\equiv, s) \wedge \mathsf{Holds}(\equiv, s')$. This is legal as we only are interested in transitions in the reachable state space. The new transition formula relates fewer states, and can consequently reduce the number of fixpoint iterations in conventional model checking methods.

Van Eijk's original paper presents a number of improvements of the basic method, such as *retiming* techniques that enlarge the set Signals so that the equivalence relation can contain more information, and *random simulation* that aims to reduce the number of pairwise comparisons by computing a better initial approximation $\equiv_0$. We will not discuss these techniques here, but refer to the original paper [33].

6.3 Stålmarck's Method Instead of BDDs

Van Eijk's method has a number of disadvantages. First of all, sometimes it is impossible to complete the analysis as some signals in the circuit can not be represented succinctly as BDDs. Second, the algorithm always finds the largest equivalence relation, which can be unnecessarily costly for proving the property. Third, the equivalences are computed by pairwise comparisons of signals, which means we have to build a quadratic number of BDDs. We will now focus on trying to solve these problems by using a SAT method instead of BDDs.

Stålmarck's method. Stålmarck's *saturation method* [92, 87] is a patented algorithm that is used for satisfiability checking. The method has been successfully applied in an wide range of industrial formal verification applications. The algorithm takes a set of formulas $\{p_1, \ldots, p_n\}$ as input, and produces an equivalence relation over the negated and unnegated subformulas of all p_i. Two subformulas are equivalent according to the resulting relation only when this is a logical consequence of assuming that all formulas p_i are true. The algorithm computes the relation by carefully propagating information according to the structure of the formulas.

The saturation algorithm is parameterised by a natural number k, the *saturation level*, which controls the complexity of the propagation procedure. The worst-case time complexity of the algorithm is $O(n^{2k+1})$ in the size n of the formulas, so that for a given k, the algorithm runs in polynomial time and space. For any specific k, there are formulas for which not all possible equivalences are found, but for every formula there is a k such that the algorithm finds all equivalences. A fortunate property is that this k is surprisingly low (usually 1 or 2) for many practical applications, even for extremely large formulas.

The advantage of having control over the saturation level is that the user can make a trade-off between the running time and the amount of information that is found. A disadvantage is that it is not always clear what k to choose in order to find enough information. In contrast, finding equivalences using BDDs results in discovering either all information, or no information at all due to excessive time and space usage.

Modification of van Eijk's method. We now adapt van Eijk's algorithm to use Stålmarck's method.

To compute the initial approximation $\equiv_0$, we use the saturation procedure to compute equivalence information between positive and negative subformulas of $\mathit{Init}(s_1)$ and $\mathsf{Holds}(\mathsf{Id}, s_1)$ under the assumption that both of the these formulas are true. Here, Id denotes the identity equivalence relation on signals, relating f to g if and only if $f = g$. Note that $\mathsf{Holds}(\mathsf{Id}, s_1)$ is a valid formula, so assuming that it is true adds no real information; we just add it to the system to ensure that all subformulas that correspond to signals are present in the resulting equivalence relation. We then use the resulting information to generate the equivalence relation $\equiv_0$ on signals.

```
1.  $\equiv_1$       $:=$ $\emptyset$ ;
2.  -- *compute first approximation*
3.  system  $:=$ $\{Init(s_1), \mathsf{Holds}(\mathsf{Id}, s_1)\}$ ;
4.  $\equiv_2$       $:=$ STÅLMARCK(system) / $s_1$ ;
5.  -- *iterate until a fixed point is reached*
6.  **while** $(\equiv_1 \neq \equiv_2)$ **do**
7.      $\equiv_1$       $:=$ $\equiv_2$ ;
8.      system  $:=$ $\{\mathsf{Holds}(\equiv_1, s_1), Trans(s_1, s_2), \mathsf{Holds}(\mathsf{Id}, s_2)\}$ ;
9.      $\equiv$        $:=$ STÅLMARCK(system) ;
10.     $\equiv_2$       $:=$ $\equiv_1 \cap (\equiv / s_2)$ ;
11. **return** $\equiv_1$ ;
```

Figure 6.3: Van Eijk's algorithm using Stålmarck's method

To improve a relation $\equiv_n$, we run the saturation procedure on a set of formulas that contains $\mathsf{Holds}(\equiv_1, s_1)$, $Trans(s_1, s_2)$, and $\mathsf{Holds}(\mathsf{Id}, s_2)$. Again, we need the formula $\mathsf{Holds}(\mathsf{Id}, s_2)$ to ensure that all subformulas that correspond to signals are present. From the result we extract $\equiv_{n+1}$ by looking at the equivalences between subformulas depending on s_2, and taking the intersection with the original equivalence relation $\equiv_n$. The intersection of two equivalence relations relates two signals if both original relations relate them.

For a schematic overview of our algorithm, see Figure 6.3. The notation $\equiv / s_1$, occuring at lines 4 and 10, turns an equivalence relation $\equiv$ on formulas into an equivalence relation on signals, by relating two signals f and g if and only if their instantiated formulas $f(s_1)$ and $g(s_1)$ are related by $\equiv$.

In our modified algorithm, we have explicit control over the running time complexity of each iteration step; each step is guaranteed to take polynomial time and space. As a consequence, we do not have to worry about a possible exponential space blowup, as in the case of building BDDs. However, having this explicit control also means that we do not always compute the *largest* relation, since the saturation algorithm is possibly incomplete depending on what k we have chosen. In many cases it turns out that even for small k, the equivalence relation we compute using Stålmarck's algorithm is still large enough to decide if the property is true or not, or to considerably reduce the number of subsequent model checking iterations.

Signal implications. Finding equivalences between signals is a rather arbitrary choice. We could just as well try to find other information about signals that is easy to compute. For example, we can compute *implications* between signals.

An implication $f \Rightarrow g$ occurs between two signals f and g, if g must be true whenever f is true. The implications over the set of signals are interesting as they can capture *all* binary relations. The reason for this is that any formula that contains two variables can be characterised by a finite number of implications between these variables, their negations, and constants.

The presented algorithm can easily be extended to find implications between the computed equivalence classes of signals. Note that implications *within* equivalence classes do not give any information, since we know that every point in an equivalence class implies every other point in the same class. Our approach for generating the implications is simple: To begin with, we generate all the equivalence classes that hold over the reachable state space using the algorithm in Figure 6.3. From each equivalence class we take a representative signal. Finally, we run a modified version of the algorithm in Figure 6.3, that uses induction to find implications rather than equivalences between the representatives.

6.4 Induction

In order to further improve our adaptions of van Eijk's method, we start by investigating another safety property verification method: *induction* [86].

Simple induction. The idea behind simple induction is to attempt to decide whether all reachable states of the described system make the formula $\mathit{Prop}(s)$ true by proving that the property holds at the initial states, and proving that if it holds in a certain state, it also holds in the next state. The inductive proof is expressible in propositional logic using the following two formulas:

$$\begin{aligned} \mathit{Init}(s_1) &\Rightarrow \mathit{Prop}(s_1) \\ \mathit{Prop}(s_1) \land \mathit{Trans}(s_1, s_2) &\Rightarrow \mathit{Prop}(s_2) \end{aligned}$$

If the first formula is valid, we know that all the initial states of the system make the property true. If the second formula is valid, we know that any time a state makes the property true, all states reachable in one step from that state also make the property true. We can thus infer that all reachable states are safe.

Simple induction is not a complete proof technique for safety properties; it is easy to construct a system whose reachable states all make $\mathit{Prop}(s)$ true, but for which the inductive proof fails. Just take a safe system and change it by adding two unreachable states s_1 and s_2 in such a way that there is a transition between s_1 and s_2, the formula $\mathit{Prop}(s_1)$ is true, and $\mathit{Prop}(s_2)$ is false. This system can not give a provable induction step even though all reachable states satisfy the property. A stronger proof scheme is needed for completeness.

Induction with depth. In the step case of simple induction we prove that the property holds in the current state, assuming that it holds in the previous state. One way to strengthen the induction step is to instead assume that the property holds in the previous n consecutive states. Correspondingly, the base case becomes more demanding.

Let $\mathit{Trans}^*(s_1, \ldots, s_n)$ be the formula that expresses that $s_1, \ldots, s_n$ are states on a path $s_1, \ldots, s_n$, and let $\mathit{Prop}^*(s_1, \ldots, s_n)$ be the formula that expresses that Prop is true in all of the states $s_1, \ldots, s_n$. *Induction with depth* n amounts to proving that the following formulas are valid:

$$\begin{aligned} \mathit{Init}(s_1) \wedge \mathit{Trans}^*(s_1, \ldots, s_n) &\Rightarrow \mathit{Prop}^*(s_1, \ldots, s_n) \\ \mathit{Prop}^*(s_1, \ldots, s_n) \wedge \mathit{Trans}^*(s_1, \ldots, s_{n+1}) &\Rightarrow \mathit{Prop}(s_{n+1}) \end{aligned}$$

The modified base case expresses that all states on a path with n states starting in the initial states make the property true. The step says that if $s_1 \ldots s_{n+1}$ is a path where $s_1 \ldots s_n$ all make *Prop* true, then s_{n+1} also makes *Prop* true. We henceforth refer to n as the *induction depth*. Note that induction with depth 1 is just simple induction.

Unique states induction. The induction scheme with depth discovers paths to any state s in the reachable state space that makes $\mathit{Prop}(s)$ false: A path with n states starting from the initial states and ending in a bad state is a counterexample to base cases of depth n or larger. As we are verifying finite systems, some depth n is therefore sufficient to discover all bugs.

Unfortunately the scheme is still not complete; it is possible to construct a safe system where the induction step fails for any depth n. Just take any safe system and change it by adding two unreachable states s_1 and s_2, so that the property holds in s_1, s_1 can both reach itself and s_2, and the property fails in s_2. Then there exist a counterexample for every depth n that loops $n-1$ times in s_1, and then visits s_2.

However, a state that is reachable from the initial states must be reachable by at least one path that only contains *unique* states. Therefore, we can add a formula $\mathsf{Uniq}(s_1, \ldots, s_n)$ to the induction step that expresses that $s_1, \ldots, s_n$ are different from each other. This restriction on the shape of considered paths makes it impossible to generate counterexamples of arbitrary length from loops in the unreachable state space. The induction step now becomes:

$$\mathit{Prop}^*(s_1, \ldots, s_n) \wedge \mathit{Trans}^*(s_1, \ldots, s_{n+1}) \wedge \mathsf{Uniq}(s_1, \ldots, s_{n+1}) \quad \Rightarrow \quad \mathit{Prop}(s_{n+1})$$

The result is a complete scheme for verifying safety properties of finite systems: If there are paths in the unreachable state space that falsely make the induction step unprovable, they are ruled out from consideration by some induction depth n. However, a major problem is that this n can be extremely large for some verification problems, and that it is often difficult to predict what n is needed.

6.5 Stronger Induction in van Eijk's Method

We will now make use of the insight into induction methods we gained in the previous section. The underlying proof method that van Eijk's algorithm uses to find equivalences that always hold in the reachable state space is simple induction. Recall that we have demonstrated that this proof technique is too weak to prove all properties that hold globally in the reachable states. Consequently

```
1.  $\equiv_1$      := $\emptyset$ ;
2.  -- compute first approximation
3.  system  := $\{Init(s_1), Trans^*(s_1,\ldots,s_n), \mathsf{Holds}^*(\mathsf{Id}, s_1,\ldots,s_n)\}$ ;
4.  $\equiv$        := STÅLMARCK(system) ;
5.  $\equiv_2$      := $(\equiv / s_1) \cap \ldots \cap (\equiv / s_n)$ ;
6.  -- iterate until a fixed point is reached
7.  while $(\equiv_1 \neq \equiv_2)$ do
8.      $\equiv_1$      := $\equiv_2$ ;
9.      system  := $\{\mathsf{Holds}^*(\equiv_1, s_1,\ldots,s_n), Trans^*(s_1,\ldots,s_{n+1}),$
                                   $\mathsf{Holds}(\mathsf{Id}, s_{n+1}), \mathsf{Uniq}(s_1,\ldots,s_{n+1})\}$ ;
10.     $\equiv$        := STÅLMARCK(system) ;
11.     $(\equiv_2)$    := $\equiv_1 \cap (\equiv / s_{n+1})$ ;
12. return $\equiv_1$ ;
```

Figure 6.4: The adaption of the algorithm for depth n unique states induction

there are circuits that contain useful equivalences that van Eijk's original algorithm misses due to the incompleteness of its underlying proof method.

Generalisation to completeness. We can make van Eijk's original algorithm complete by modifying our implementation to use unique states induction with depth n rather than simple induction. In the base case of the algorithm, we compute an equivalence relation on signals that hold in the first n states on paths from the initial states. In the algorithm step, we assume that our most recently computed signal equivalence relation holds in the first n of $n+1$ consecutive unique states, and derive the subset of the signal equivalences that necessarily holds in state $n+1$.

For a detailed description of the resulting algorithm, see Figure 6.4. We use the notation $\mathsf{Holds}^*(\equiv, s_1, \ldots, s_n)$, occuring at lines 3 and 9, as a shorthand for $\mathsf{Holds}(\equiv, s_1) \wedge \cdots \wedge \mathsf{Holds}(\equiv, s_n)$. The algorithm for finding implications between signals is modified in an analogous way.

We can now discover all equivalences that hold globally in the state space. In particular, if a safety property holds in all reachable states, there exists a saturation level and an induction depth that is sufficient to discover that the corresponding signal is equivalent to the true signal.

As an additional benefit, the possibility to adjust both the saturation level and the induction depth allows a high degree of control over how much work is spent on discovering equivalences. We can now increase the number of equivalences that can be discovered for a fixed saturation level by increasing the induction depth; this can be useful as an increase in saturation level means a big change in the time complexity of the algorithm.

We note that the idea of using stronger induction not is restricted to our SAT-based version of van Eijk's algorithm; the original BDD-based algorithm can also be made complete by stronger induction.

```
1. $n, S_0 := 0, \textsc{init}$ ;
2. loop
3.     $S_{n+1} := \textsc{postImage}(S_n) \cup S_n$ ;
4.     $n := n + 1$ ;
5. until $(S_{n+1} = S_n)$ ;
6. return $S_{n+1}$ ;
```

Figure 6.5: A standard forwards reachability algorithm

6.6 Approximations

In this section we show that van Eijk's algorithm is an *approximative forwards reachability* analysis. We then use this insight to derive an analogous backwards approximative analysis, and combine the two algorithms into a mutual improvement algorithm.

The forwards reachability view. Figure 6.5 shows the shape of a standard forwards reachability analysis, where we use INIT to denote the set of initial states, and the operation POSTIMAGE to compute postimages (the postimage of a set of states S is the set of states that can be reached from S in one transition). We now demonstrate that van Eijk's algorithm performs such a forwards analysis approximatively, in the sense that it is a variant of the standard analysis where INIT has been replaced with an overapproximation, and the exact operations $\cup$ and POSTIMAGE have been replaced by overapproximative operators.

Van Eijk's algorithm computes a sequence of relations $\equiv_i$. Each of the corresponding formulas $\mathsf{Holds}(\equiv_i, s)$ can be seen as the characterisation of a set of states A_i.

In the base case, the algorithm computes the binary relation $\equiv_0$ that holds between points in all the initial states. The formula $\mathsf{Holds}(\equiv_0, s)$ will therefore be valid for every state s that makes $\mathit{Init}(s)$ valid, and possibly for some other states. A_0 is consequently a superset of the initial states.

In the step, the algorithm computes the subrelation $\equiv_{n+1}$ of $\equiv_n$ that provably holds after a transition under the assumption that $\equiv_n$ holds before the transition. Every state s that is reachable in one transition from a state in A_n therefore makes $\mathsf{Holds}(\equiv_{n+1}, s)$ valid. But $\equiv_{n+1}$ is a subrelation of $\equiv_n$, so every state s in A_n also satisfies $\mathsf{Holds}(\equiv_{n+1}, s)$. Consequently, the step operation corresponds to computing A_{n+1} as the overapproximative union of A_n and the overapproximative postimage of A_n.

Finally, the algorithm checks whether $\equiv_{n+1}$ is the same relation as $\equiv_n$. This corresponds to checking whether $A_{n+1} = A_n$. If this is the case, the algorithm terminates, otherwise the step is iterated.

Approximative backward analysis. It is well known that forwards reachability analysis has a dual analysis called *backwards* reachability analysis [22]. We can perform the backward analysis using the forwards algorithm by modifying the characterisation of the underlying system in the following way:

$$\begin{aligned} \mathit{Init}'(s) &= \neg\,\mathit{Prop}(s) \\ \mathit{Trans}'(s,s') &= \mathit{Trans}(s',s) \\ \mathit{Prop}'(s) &= \neg\,\mathit{Init}(s) \end{aligned}$$

The result of the computation is the set of states that are backwards reachable from the bad states—the states where the property does not hold.

We can use the system transformation together with any of our variants of van Eijk's algorithm. In particular, we can compute a relation $\equiv$ that characterises an overapproximation of the states that are backwards reachable from the states that make $\mathit{Prop}(s)$ false. Analogously to the forwards algorithm, the system is safe if $\mathsf{Holds}(\equiv, s)$ implies the safety property, which in this case corresponds to that no initial state is in the overapproximation of the set that can be backwards reached from the bad states. Also, if we do intersect the initial states, we can still use the approximation to constrain the transition relation in order to reduce the number of necessary iterations of an exact forwards reachability algorithm.

Mutual improvement. The new approximate backward algorithms can be very useful on their own. However, there exists a general way of enhancing approximative reachability analyses that improves matters further [45].

The idea is to first generate the overapproximation of the reachable states. If the corresponding set has an empty intersection with the bad states, we are done. If it has an nonempty intersection, there are two possible reasons: Either the system is unsafe, or the approximation is too coarse. Regardless of which is the case, we know that the only possible bad states we can reach are those that are contained in the intersection. We can therefore take the intersection to be our new bad states.

But now we can apply approximate backwards reachability analysis from the new bad states. If we do not intersect the initial states, the system must be safe. If we do, we can take the intersection to be the new initial states and restart the whole process. The algorithm terminates if we generate the same overapproximations twice, as this implies that no further improvement is possible.

The resulting mutually improved overapproximations are always at least as good as the original overapproximations, and they can sometimes be substantially better as we demonstrate in the next section.

6.7 Experimental Results

In this section, we present a number of experiments we have done using a prototype implementation of our variants of van Eijk's algorithm. We compare our results against the results of three other methods. The first two are reachability analysis and unique states induction, as implemented in the SAT-based model checking workbench FixIt [2]. The third method is BDD-based model checking, as implemented in the verification tool VIS version 1.3. In the experiments with VIS we have used dynamic variable reordering and experimented with different partitionings.[2] All running times are measured on a 296 MHz Ultrasparc-II with 512 MB memory. The results are displayed in Table 6.1.

The motivation for the choice of benchmarks is as follows. We have chosen one industrial example, one example that is difficult to represent with BDDs, and one example that belongs to the easy category for BDDs.

The Lalita example. The Lalita example is an industrial telecommunications example from Lucent Technologies that was one of the motivations for the research presented in this paper. We received the example as a challenge from Prover Technology, a Swedish formal verification company. It was given to us as a black-box problem; we had no information about the structure of the system. The design was already known to be within reach of BDD technology, but not all of the properties were possible to verify using unique states induction. When we attempted to verify the design using SAT-based reachability analysis, the representations became excessively large due to the computation of pre- and postimages.

The design contains 178 latches. The problem comes with thirteen safety properties that should be verified; we present the four most interesting properties: the two properties that were most difficult for VIS (2 and 7), one of the two properties that are hard for induction (11), and the property that was hardest for our methods (10).

All of the properties except property 10 and 11 can be done using SAT-based induction. However, we can verify or refute every property except property 10 directly using our forwards equivalence algorithm. Property 10 is verified using one iteration of mutual improvement of the computed equivalences. As the table demonstrates our analyses are a factor 10 to 100 faster than BDD-based verification in VIS.

The butterfly circuits. This family of benchmarks arose when we were designing sorting circuits for implementation on an FPGA. The problem is to decide whether a butterfly network containing reconfigurable sequential components is equivalent to an optimised version where the components have been shifted around. When we attempted to verify the circuits we discovered that standard algorithms could not handle circuits of any reasonable size. In particular, BDD-based methods did not work because the BDDs representing the circuits became too large.

The model checking algorithms in VIS are unable to verify larger networks than size 4 in a reasonable amount of time and space. SAT-based reachability analysis and induction are also unable to cope with larger instances of the circuits. However, the forwards equivalence algorithm handles all the sizes we

[2]We have also tried approximate model checking in VIS, but there appears to be a bug in the implementation which makes it unsound.

Property	FixIt Reach.	FixIt Induct.	VIS	Our Method
Lalita, nr. 2	0.3	0.2	219.9	2.3[a]
7	41.8	0.2	207.3	2.2[a]
10	[>15min]	[>15min]	86.6	9.7[b]
11	[>15min]	[>15min]	199.3	2.1[a]
Butterfly, size 2	0.1	1.0	0.3	0.1[a]
4	16.6	[>15min]	2.0	0.1[a]
16	[>15min]	—	[>15min]	1.4[a]
64	—	—	—	37.4[a]
Arbiter	2.5	[>15min]	2.1	76.9[c]

[a] with equivalences, [b] with mutual improvement, [c] with implications

Table 6.1: Experimental results (times are in seconds).

have tried in less than 40 seconds.

The arbiter. This example is a simple benchmark from the VIS distribution. The arbiter controls three clients that compete for bus access. We verify the property of mutual exclusion.

The problem is easy both for BDDs and SAT-based symbolic reachability analysis, but can not be done using unique states induction. The example clearly demonstrates that finding implications between equivalence classes can be much stronger than only computing equivalences: Our equivalence based analysis alone is unable to verify the design in a reasonable amount of time, but we can verify the design in 77 seconds by computing implications between the equivalence classes.

6.8 Related Work

The first approach in the literature to apply SAT-based techniques to model checking was Bounded Model Checking [6]. Bounded model checking of safety properties corresponds to searching for bugs by attempting to prove the base case only of induction with depth. Certain bugs that are hard to find using BDD-based model checking can be found very quickly in this way. In order to effectively also prove safety of systems, standard symbolic reachability analysis was adapted to use SAT-solvers [2] which resulted in the analysis implemented in FixIt. Currently, interesting work is being done on combinations of SAT-solvers and BDDs for model checking [99].

The idea to use approximate analyses to generate semantic information from systems originally comes from the field of program analysis. Many different such analyses can be seen as abstract interpretations [26]. In particular, Halbwachs et al. [45, 48] have used abstract interpretation techniques to generate linear constraints between arithmetic variables that always holds in the reachable state space of synchronous programs and timed automata. This information is used both for compilation purposes and for verification. The same techniques are used for generating strengthenings in the STeP system [65] that is targeted towards deductive verification of reactive programs.

The main differences between our work and the work on synchronous programs and STeP, is (1) that the analyses we present here are specially designed for generating information about *gate level circuits* rather than programs, (2) that we focus specifically on simple relations between boolean signals, and (3) that we use Stålmarck's saturation method as a possibly incomplete but fast method for generating the relations. Also, we generate information while keeping in mind that we can apply an exact analysis later, and we have consequently optimised the algorithms for quickly generating information. In the case of synchronous program verification and STeP, a precise analysis is not even possible in general as infinite state systems are addressed.

Dill and Govindaraju [41] have developed a method for performing BDD-based approximate symbolic model checking based on overlapping projections. Their idea is to alleviate BDD blow-up by representing an overapproximation of a set of states S as a vector of BDDs, where each individual BDD characterises the relation in S between some subset of the state variables. The conjunction of the BDDs represents an overapproximation of the underlying set. The main difference between our approach and theirs, is that we consider some particular relations between *all* pairs of signals, while they consider all relations between a number of subsets of state variables. Also, the user of Dill and Govindaraju's method must manually choose the subsets of state variables to build BDDs for, whereas our methods are fully automatic.

6.9 Conclusions and Future Work

We have taken an existing BDD-based verification method which finds equivalent points in a circuit, and adapted it to use Stålmarck's method instead of BDDs. Then, we strengthened the resulting algorithm by combining it with recently developed induction techniques. We also discussed how the algorithm can be improved by computing implications rather than simple equivalences between points. Lastly, we observed that the algorithm can be transformed into a mutual improvement approximative reachability analysis.

The resulting collection of new algorithms can be seen as SAT-based improvements of van Eijk's original algorithm, where we use stronger inductive methods. Viewed from this angle, we have made van Eijk's method complete and provided a more fine-grained tuning between the time used and the information found.

Viewing our work in a different way, we can say that we have improved an inductive method by using van Eijk's algorithm to find equivalences. In some cases, such as the butterfly examples (see Section 6.7), unique states induction needs an exponentially larger induction depth than our improved analyses.

We believe the proposed methods work well for several reasons. First of all, van Eijk's original idea of finding equivalences of points in the circuit makes it very hard for properties to "hide" deep down in the logic of a circuit. Comparing this with problems occurring with methods that only look at state variables (such as conventional model checking methods) or methods that only look at the outputs (such as inductive methods) clearly shows that this is a desirable thing to do.

Second, the use of Stålmarck's saturation algorithm forms a natural fit with van Eijk's original algorithm. The possibility of controlling the saturation level pays off especially in systems where it is hard to find *all* equivalences, but sufficient to find some. Stålmarck's algorithm is also rather robust in the number of variables used in formulas.

Third, inductive methods perform well because they do not need any iteration or complicated quantification. Unfortunately, when we prove partial properties of systems, or when we prove properties about systems with a lot of logic between the latches and the property, induction performs poorly because the induction hypothesis is not strong enough to establish the inductive step. In this case, finding equivalence or implication information is just the right thing to do, because it strengthens the induction hypothesis, and provides direct information not only about the latches, but about all points in the circuit.

For future work, we would like to investigate other signal relations than equivalences and implications. General relations over three variables is a candidate, although it is not clear how to represent the found information. Furthermore, we are interested in extending the proposed algorithms to work with other properties than just safety properties. Lastly, we would like to extend the presented ideas to the verification of safety properties of synchronous reactive systems; for example, systems implemented in the programming language Lustre [46]. In order to do this, we need to add support for integer arithmetic and to investigate how Halbwachs's ideas [45] can be combined with our analyses.

Acknowledgements

Many thanks to Niklas Sörensson who implemented a large part of the algorithms and benchmarks, as well as to Niklas Eén, who helped with running some of the benchmarks. We also thank Byron Cook, Koen van Eijk, Carl-Johan Lillieroth, Gordon Pace, Mary Sheeran, and Satnam Singh for their useful comments on earlier drafts of this paper.

Chapter 7

Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers

We describe the techniques we have used to search for bugs in the memory subsystem of a next-generation Alpha microprocessor. Our approach is based on two model checking methods that use satisfiability (SAT) solvers rather than binary decision diagrams (BDDs).

We show that the first method, bounded model checking, can reduce the verification runtime from days to minutes on real, deep, microprocessor bugs when compared to a state-of-the-art BDD-based model checker. We also present experimental results showing that the second method, a version of symbolic trajectory evaluation that uses SAT-solvers instead of BDDs, can find as deep bugs, with even shorter runtimes. The tradeoff is that we have to spend more time writing specifications.

Finally, we present our experience with the two SAT-solvers that we have used, and give guidelines for applying a combination of bounded model checking and symbolic trajectory evaluation to industrial strength verification.

The bugs we have found are significantly more complex than those previously found with methods based on SAT-solvers.

This chapter was written together with Tim Leonard and Abdel Mokkedem. It will be published at the International Conference on Computer Aided Verification 2001 [12].

7.1 Introduction

Getting microprocessors right is a hard problem, with harsh punishments for failure. With current design methods, hundreds to thousands of bugs must be found and removed during the design of a new processor, and there are heavy economic incentives to get most of them out before first silicon.

Current designs are so complex that simulation-based methods are no longer adequate. Most companies in the industry, including at least AMD, Compaq, HP, IBM, Intel, Motorola, and Sun, have therefore investigated formal verification. Their choices of methods, tools, and application area have varied, as has their level of success.

One of the areas we have concentrated on at Compaq is property verification for our microprocessor designs. Among other things, we have investigated the use of symbolic model checking [67] to find *Register Transfer Level* (RTL) bugs in a next-generation Alpha processor. Our goal in this work has been to find bugs, rather than to prove their absence, since there are many bugs to find in a design under development.

Our initial experiments with symbolic model checking convinced us that the capacity limits of many model checkers prevent us from finding bugs cost effectively. The best model checker we could find, an experimental version of Cadence SMV [68], needs several hours to days to check simple properties of heavily reduced components. As a consequence, we have also looked at model checking using satisfiability (SAT) solvers [6, 2, 99]. These methods have shown real promise, especially for finding bugs, when compared to BDD-based model checkers like SMV.

In this paper, we describe how we have applied two SAT-based verification techniques to find bugs in the memory subsystem of the Alpha chip. The first technique, bounded model checking (BMC) [6], has previously been applied to industrial verification, but not for finding bugs of length anywhere near what we will describe. The second of these techniques, symbolic trajectory evaluation (STE) [81], has previously not been used together with SAT-solvers at all.

We compare the performance of SAT-based bounded model checking to state-of-the-art BDD-based model checking, and present results showing the usefulness of SAT-based STE. Our experiences are very positive: the use of SAT-based methods has reduced the time for finding certain bugs from days to a few minutes. We also compare the performance, when finding bugs in real designs, of the two SAT-solvers we have used: GRASP [89], and Prover Technology's PROVER [87] proof engine. Finally, we present guidelines for applying a combination of BMC and SAT-based STE to microprocessor bug finding.

Related Work. *Bounded model checking* [6] (BMC) was invented by Biere and coworkers as a method for using SAT-solvers to do model checking. BMC has previously been applied to bug finding for Power PC chips [7]. To our knowledge, BMC is the only SAT-based model checking method that has been used in realistic microprocessor verification.

In the Power PC verification, the authors did not model the environment of the designs under analysis. BMC quickly found short counterexamples to the properties being verified, but they were false failures due to illegal input sequences. BMC did well at this compared to BDD-based model checking, but the results said little about whether BMC could find real bugs, which are generally much deeper. We, on the other hand, present the results of searching for, and finding, real, deep bugs. One of our important contributions is therefore that we demonstrate that BMC together with cutting edge SAT-solvers has the capacity to find realistic bugs in industrial designs.

Symbolic trajectory evaluation (STE) is a model checking method invented by Seger and Bryant [81] that consists of an interesting mix of abstract interpretation and symbolic evaluation. STE is in industrial use, primarily for data path and memory verification, at companies including Intel [1] and Motorola. Up to now, STE has always been implemented using BDDs; the use of SAT-solvers to do STE has not been reported previously in the literature. Moreover, we apply symbolic trajectory evaluation to verification at the synchronous gate level—a fairly high level of abstraction for STE, which has previously been used predominantly at the transistor level.

There are other ways of doing SAT-based model checking than the ones that we discuss in this paper. We refer readers interested in these alternative approaches to [2, 99, 43, 86, 10].

The paper is organised as follows. In sections 7.3 and 7.4 , we give brief introductions to BMC and STE. We then describe the component that we have focused on, the merge buffer, and the process we have used to analyse it. After that we go on to describe the actual use of the verification tools and the results. Finally, we give guidelines for using a combination of BMC and STE for heavy-duty industrial verification.

7.2 Preliminaries

In this paper, we will search for counterexamples to properties of synchronous gate-level hardware. Such circuits can be viewed as finite transition systems, where the states are value assignments to a vector $s = (s.\mathit{0}, \dots , s.n)$ of boolean variables called the system's *state variables* [22]. The transition system for a given circuit can be represented as two propositional logic formulas [2]:

$\mathit{Init}(s)$	Initial states formula
$\mathit{Trans}(s, s')$	Transition relation formula

The first formula, *Init*, is a formula that characterises the initial states by evaluating to true exactly for the assignments to the state variables that are initial states. The second formula, *Trans*, evaluates to true for s and s' precisely when there is a transition from the state assigned to s to the state assigned to s'.

Our analyses take as inputs the formulas *Init* and *Trans* together with a description of a property to check. Such a property might for example be "a store instruction to an IO address is never discarded." The aim of the analyses is then to generate a trace, if one exists, where an IO store is thrown away.

In the case of BMC, we will specifically focus on detecting failures of *safety properties*. Informally, safety properties are properties of the form "in every reachable state of the system, the property P holds."

7.3 Bounded Model Checking

Bounded model checking tries to find bugs in a system by constructing a formula that is satisfiable precisely if there exists a length N or shorter trace violating a property given by the user. The BMC procedure feeds this formula to an external SAT-solver, and uses the returned assignment (if any) to extract a *failure trace*.

The bound N is given by the user, and will affect both the size of the generated formulas, and the length of the failure trace that can be detected. A negative answer from the SAT-solver for a given N does not mean that the whole system is safe, only that there are no failure traces of length N or shorter. BMC is thus used to find bugs, rather than to prove their absence.

We assume that the safety property we are interested in has been encoded as a propositional logic formula $\mathit{Prop}(s)$ that will evaluate to true exactly for the states fulfilling the property. Given the bound N, and the formulas $\mathit{Init}(s)$, $\mathit{Trans}(s, s')$, and $\mathit{Prop}(s)$, the BMC procedure constructs the following formula, which characterises failure traces of length N or shorter:

$$\begin{aligned}
&\mathit{Init}(s_1)\ \wedge \\
&\mathit{Trans}(s_1, s_2) \wedge \ldots \wedge \mathit{Trans}(s_{N-1}, s_N)\ \wedge \\
&(\neg\, \mathit{Prop}(s_1) \vee \ldots \vee \neg\, \mathit{Prop}(s_N))
\end{aligned}$$

If the SAT-solver returns an assignment to the state variables in $s_1 \ldots s_N$ that makes this formula true, then there exists an initial state s_1 in the system, from which we can reach another state s_k ($k \in \{1 \ldots N\}$) where the property fails. The BMC procedure can thus extract a failure trace from the assignment.

7.4 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation is a form of model checking using four-valued rather than two-valued logic.

A symbolic trajectory evaluator takes $\mathit{Trans}(s, s')$ as input together with a so called *trajectory assertion* of the form $\mathit{Ant} \Rightarrow \mathit{Cons}$. The antecedent and consequent, Ant and Cons, are lists of equal length, in each of which the ith entry says something about the system's state variables at time i.

As an example, assume that we have a circuit whose state variable $s.o$ contains the delayed **or** of $s.a$ and $s.b$. The following trajectory assertion specifies this circuit completely:

$$[\texttt{node } s.a \texttt{ is } x \texttt{ and node } s.b \texttt{ is } y, \langle\cdot\rangle] \Rightarrow [\langle\cdot\rangle, \texttt{node } s.o \texttt{ is } x \vee y]$$

Here $\langle\cdot\rangle$ means "no requirements on the state variables", so the assertion can be read, "if we assume that $s.a$ and $s.b$ contain the values x and y at some time i and we make no assumptions about any state variables at time $i + 1$, that implies that $s.o$ contains the logical **or** of x and y at time $i + 1$."

The job of the trajectory evaluator is to compute a boolean expression *ok* that evaluates to true precisely for the assignments to x and y that make the trajectory assertion true. In order to cope with unknown and overspecified values, the trajectory evaluator uses a four-valued logic to represent the contents of the state variables over time. (A state variable can for example get an overspecified value by being required to be true and false at the same time.) In addition to the values *True* and *False*, the four-valued logic therefore contains the values X (unknown), and $\top$ (overspecified).

When *ok* has been computed, the evaluator uses an external SAT-solver to check whether there exists an assignment that makes *ok* evaluate to false. If there is such an assignment, there is a trace of the system that is consistent with the antecedent but violates the consequent. The trajectory evaluator then instantiates the trajectory assertion with the falsifying values, and constructs a failure trace that is given back to the user.

7.5 The Merge Buffer

Alpha processors, like most state-of-the-art microprocessors, have a very hierarchical structure. A processor is divided into a handful of so called *boxes*, each responsible for dealing with a particular aspect of instruction execution. For example, the IBox handles instruction fetch, and the MBox executes memory-reference instructions. Each box is further divided into a handful of parts that we will call subboxes.

The subbox that is the focus of our attention in this paper is the *merge buffer*, an important component of the MBox for a next-generation Alpha chip. We chose

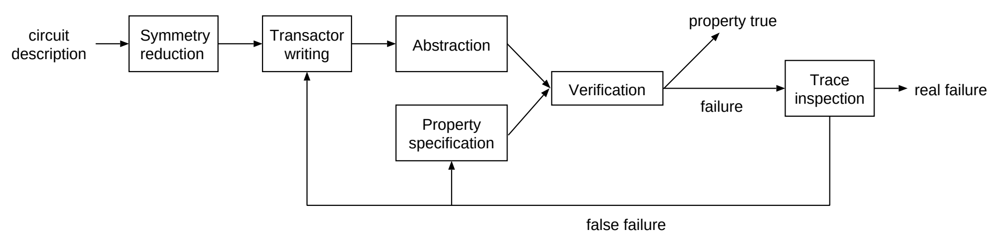

Figure 7.1: Our verification flow

the merge buffer as it is one of the most complex subboxes in the processor. Our hope is that if we can cost-effectively find bugs in this component, then we can use the same methods on most other subboxes.

The function of the merge buffer is to receive requests to write into memory, and to reduce the traffic on the memory bus by merging stores to the same physical address. In order to do the merging correctly, the merge buffer communicates with four other subboxes: (1) the *store queue*, where store instructions are saved until they are written out of the merge buffer; (2) the *load queue*, where load instructions are stored until they have received results from memory; (3) the *CBox*, which deals with the cache coherence protocol; and (4) the *backend tag module*.

The merge buffer is essentially a large buffer with a very complex policy for reading in entries, merging stores, and writing out stores to the memory. It has about 14 400 latches, 400 primary inputs, and 15 pipeline stages. The pipeline has complex feedback that prevents us from retiming away latches.

7.6 Analysis Cycle

In Figure 7.1 we show the analysis cycle that we have used to locate bugs in the merge buffer.

We start off with the original RTL description of the circuit. As the full-size merge buffer contains more than ten thousand latches—too much state to be feasible to verify using standard model checking technology—we need to reduce the size of the model. The idea is to remove portions of the state in the circuit in ways that do not alter the circuit behaviour with respect to the properties of interest. The most important reductions are *symmetry reductions* [56], which we use to reduce the number of buffer entries, address bits per entry, data bytes per entry, and bits per data byte.

We do not mind if some of our reductions do not preserve all possible properties of the circuit, as long as we can find problems in the reduced circuit that also are present in the full size circuit. The reason for this is that we are interested in finding bugs, as opposed to proving correctness. We are thus permitted to do ad-hoc reductions that are formally incorrect, but that preserve most of the interesting behaviour of the circuit.

After the reductions, the merge buffer has about 40 primary inputs. When the merge buffer is in use, these inputs will be connected to the four subboxes with which the merge buffer communicates. If we leave them unrestricted, the verification will be done under the assumption that any inputs can occur at any times. However, in order to function correctly, the merge buffer relies on assumptions about the behaviour of its environment. We therefore have to restrict the input to the merge buffer by adding *transactor* state machines that react to the verification environment and drive some of the inputs so that we rule out input behaviours that could not arise in real use.

We then abstract the resulting circuit in two ways. First, we use an RTL compiler to optimise the circuit by performing transformations like constant propagation and common subexpression elimination. The reduced merge buffer now has about 1800 latches and 10 free primary inputs. We then do a final abstraction step that removes redundant latches, and replaces groups of transparent latches that we cannot model synchronously with standard flip-flops. The final model has about 600 state nodes in the cone of most properties.

The end result of the reductions and abstractions is the model that we give to the verification tools. However, before we can do that, we need to write down the property of interest in a format that the tool we want to use accepts. Given the model and the property, the verification tool then either produces a failure trace, or tells us that the property is true (which has little meaning as we have performed ad-hoc reductions).

A lot of design knowledge is needed to decipher a failure trace; a property can fail for more than one reason. First of all, we might have made a specification mistake that causes the tool to diagnose an intended behaviour of the system as a failure. In this case we need to modify the property. Second, the trace might be a trace that the real system could not exhibit, because it has arisen due to the merge buffer's environment providing input signals that cannot occur in real-life. In this case we need to go back and modify the transactors so that we disallow this behaviour, and re-abstract the resulting model. Third, we might have found a real bug.

7.7 Verification

In this section, we describe our experiences of applying BDD-based symbolic model checking, BMC, and STE to the merge buffer. The areas of the merge buffer that we target have previously been well explored with simulation-based verification.

Failure length	SMV sec	CAPTAIN PROVE BMC sec	GRASP BMC sec
25	62 280	85	25
26	32 940	19	19
34	11 290	586	272
38	18 600	39	101
53	54 360	1 995	[>10000 s]
56	44 640	2 337	[>10000 s]
76	27 130	619	6 150
144	44 550	10 820	[>10000 s]

Table 7.1: Comparison between bounded model checking and SMV.

7.7.1 BDD-based Symbolic Model Checking

SMV was the first BDD-based tool that we evaluated that showed some promise for checking non-trivial merge buffer properties. (We have evaluated several.) However, most of the interesting merge-buffer properties contain about 600 latches in the cone of influence, and BDD-based model checking of state machines containing more than a couple of hundred latches is highly non-trivial. In order to find bugs using SMV, we therefore have to decrease the size of the cone by setting a subset of the 10 free primary inputs to specific values during the run. These values restrict the part of the state space that we explore using the model checker.

In order to get better performance out of SMV, we have ported it to the 64-bit Alpha architecture. This allows us the benefits of performing the model checking runs on a high performance server with 8 GB of main memory. We have also removed some bottlenecks in the implementation and augmented the standard variable reordering heuristics with two special purpose tactics.

In spite of the improvements to SMV, each property still takes several hours to explore on the server. We have found many bugs this way, but it is slow.

7.7.2 Bounded Model Checking

The first alternative to BDD-based model checking that we have investigated is bounded model checking, as implemented in the SAT-based model checking workbench FIXIT [2].

One of the SAT-solvers that we wanted to use together with FIXIT, PROVER [87], is currently not available for the Alpha architecture. We have therefore done all of our BMC runs on a 32-bit PC. The performance of the BMC analysis is still remarkable. Even though we are not using a high-performance processor with many gigabytes of memory, we can find failures in a fraction of the time needed by SMV. In Table 7.1 we compare the runtimes of BMC, running on a 450 MHz 32-bit PC, to SMV, running on a 700 MHz 64-bit Alpha.

The first column of BMC runtimes is obtained using CAPTAIN PROVE, a command-line tool based on Prover Technology's proof engine PROVER [87]. CAPTAIN PROVE uses PROVER's application programming interface [78] to search for models using *strategies*. A simple such strategy, which we will refer to as the *timed strategy*, looks as follows:

```
sat 1 time 3600.
back level 5 [ sat 1 time 30. ].
```

The timed strategy first does a preprocessing step called *1-saturation* [87] for 3600 seconds. This analysis tries to find information restricting the search space we have to traverse for a model. The 1-saturation is then followed by the actual search, *backtracking*. At every fifth level of the search tree, the SAT-solver is instructed to do 30 seconds of additional 1-saturation.

The use of strategies allows us to control the search for assignments. We use different choices of strategies for different bounds N. When N is less than 40, we use the default strategy of 1-saturation without a time limit followed by normal backtracking. For N larger than 40, we use the timed strategy with different values for the initial 1-saturation. For example, for length 60 traces we normally need 1000 seconds of initial saturation, whereas for traces over 100 cycles long we use 10 000 or 20 000 seconds of initial saturation.

As can be seen from Table 7.1, BMC using CAPTAIN PROVE detects the failures significantly faster than SMV. In some cases it reduces the runtime for finding a bug from a day to a couple of minutes. The lengths of failures that are detected ranges from 25 cycles up to well over a hundred cycles.

The second column of BMC runtimes is obtained using GRASP [89]. In our previous experience, this is the highest capacity public domain SAT-solver. As can be seen in the table, CAPTAIN PROVE and GRASP both work well for short failures. For longer failures, CAPTAIN PROVE outperforms GRASP. (Please note that the reason for the [>10000 s] table entries is that GRASP automatically terminates after 10 000 seconds; we did not cut it off.)

7.7.3 SAT-based Symbolic Trajectory Evaluation

The second alternative to BDD-based model checking that we have investigated is a SAT-based version of symbolic trajectory evaluation that we have implemented in FIXIT.

The advantage of using STE instead of BMC is that we are not forced to give symbolic values to each time-instance of a state variable. Instead we can choose to give concrete values to some state variables, or leave them to contain X. This potentially permits us to do much deeper exploration of the state-space than we can do using BMC, while preserving the short run times.

However, in order to take full advantage of this increased flexibility, we have to spend more time coming up with a good specification that judiciously gives concrete and symbolic values to the right variables.

For example, if we do not give concrete or symbolic values to some of the state variables, they are initialised to contain the unknown value X. This value often propagates, since it may be impossible to draw conclusions about the outputs of a gate with an unknown input. We might also have forgotten to assign a value to a primary input at an important time. When a property fails because of such underspecification, we have to make the specification more detailed by introducing symbolic or concrete values. A given STE specification will thus often have to go through several iterations of revision.

Failure length	CAPTAIN PROVE	GRASP
	sec	sec
77	7.7	33.3
77	7.7	34.2
112	10.8	51.9
123	11.7	51.9

Table 7.2: Runtimes for detecting failures using symbolic trajectory evaluation.

In Table 7.2, we present the runtimes needed to find four bugs in the merge buffer using STE. The times to do the actual detections are short, but we had to spend a lot of time developing the specifications. Luckily, the turnaround time for discovering that an assertion is underspecified is a few seconds at most, which means that the specification work is very interactive.

The table shows a clear difference between the performance of STE using GRASP and CAPTAIN PROVE. However, the actual runtimes are very low in both cases. For the purpose of using SAT-based STE to locate bugs in the merge buffer, we can clearly make do with a public domain SAT solver.

7.8 A Proposal for a Methodology

From the previous section, it is clear that BDD-based model checking, BMC, and STE have very different characteristics. Based on the experiences we have had while locating design errors in the merge buffer, we have the following suggestion for a methodology:

- Start the analysis of a new subbox with bounded model checking.
- Initially test a new property with a small bound, so that the check only takes a few seconds. This will catch low-hanging fruits, and alert us to simple problems with inputs that are not properly constrained.
- Remove false counterexamples by modifying the transactors or the property, as appropriate.

- Start looking for long failures of the property. Choose a small set of bounds, ranging from medium long up to very challenging, and check each of them using the timed Captain Prove strategy. Use longer and longer saturation times.

- Use STE to quickly check that the problem is fixed whenever the designers have corrected a bug found using BMC. Also abstract the failure trace by making some of the inputs or control signals symbolic. This allows quick checking for failures that are similar to the original failure.

- When the BMC checks start taking more than half an hour or so, start working in parallel on using STE to find the bug.

- If neither BMC nor STE seems to find any failures, try SMV or move on to another property.

7.9 Conclusions

In this paper, we have presented the techniques that we have used to find bugs in a crucial component of a microprocessor in design. Our approach is based on bounded model checking and a SAT-based version of symbolic trajectory evaluation that we have developed.

Our experimental results demonstrate that it is possible for BMC to outperform state-of-the-art BDD-based symbolic model checking by two orders of magnitude, even when we look for bugs in deeply pipelined industrial components. None of the bugs described here has been a false counterexample. As a result, their complexity in terms of the length of minimum failure traces has been significantly larger than previously have been found using SAT-based techniques.

We have had less time to evaluate the use of SAT-based STE, but it seems clear that it is a very attractive bug-finding method. We have used STE to find bugs as deep as the ones we have been able to find using BMC, with negligible runtimes. However, this does not come for free; we have decreased the tool's runtime by spending more time developing specifications.

We have also presented a comparison of the performance of Captain Prove and GRASP for BMC and STE, and suggested a methodology for SAT-based industrial bug finding.

We believe that the approach we have presented here can be cost effective, and that the techniques we have used will become vital instruments in the standard verification toolbox. During the two months when the work that is presented in this paper was done, we improved the SAT-based framework FixIt significantly and removed many bottlenecks that we had not encountered on academic examples. The dramatic decrease in runtimes that we achieved in this short time makes us believe that there is a large potential for further improvement.

Acknowledgements

Many thanks to Gunnar Andersson, Luis Baptista, Arne Borälv, and João Marques Silva, who gave advice on running the SAT-solvers. We would also like to thank Gabriel Bischoff, John Matthews and Mary Sheeran for their useful comments on earlier drafts of this paper. Finally, Per Bjesse thanks Compaq's Alpha Development group for hosting him during the autumn of 2000.

Chapter 8

SAT-based Model Checking: A Tutorial and Overview

Symbolic model checking is one of the most widely used methods for verifying finite state systems such as hardware components and certain protocols. The key factor that made this possible was the realisation that Binary Decision Diagrams (BDDs) could be used to encode the system and the sets of states that are generated during the verification process in a way that often is very compact. BDDs are extremely powerful datastructures. Unfortunately, many useful systems are inherently hard to verify using BDDs.

Recently, there has been a lot of interest in the application of satisfiability (SAT) solvers to design verification. The satisfiability solvers of today are very mature, and have been successfully applied in many domains. Interestingly, SAT solvers seem to have strong points that complement those of BDDs. As a consequence, a number of methods for doing symbolic model checking based on satisfiability solvers instead of BDDs have been proposed in the research literature. These methods have been used to verify systems that are too hard for BDD-based model checkers.

In this paper, we explain three of these previously presented methods in a tutorial style, make some comments about how they are related, and consider how they can be implemented. We also outline some future research directions that we believe will be important. Our overall aim is to make the presentation as easy to follow as possible, and we therefore focus on the verification of safety properties of synchronous hardware.

This chapter was written together with Mary Sheeran and Gunnar Stålmarck. It is submitted for publication [13].

8.1 Introduction

Model checking [21, 79] is an automatic method for verifying systems that can be modelled as finite state machines. Such systems include hardware components and communication protocols. The idea behind model checking is to determine whether or not a property of a system holds by searching through the state space of a model of the system. As a consequence, the first model checking algorithms had a complexity that was proportional to the size of the state space of the investigated systems. Although model checking was considered interesting in industry, the direct dependence between the computation time and the size of the state space made verification of realistic systems impossible. If this seems surprising, consider that a synchronous hardware component containing n latches has a state space containing 2^n states when it is modelled in the standard way.

Due to this limitation, it was not until the introduction of *symbolic* model checking algorithms [18, 67] that model checking started to be used in industrial design projects. In symbolic model checking, the state machine is encoded symbolically, rather than in terms of explicit states. This is done by letting every state be represented by a vector of boolean values, and encoding sets of such states by symbolic expressions. Relations between states are encoded by viewing a relation as a set of pairs of states. Of course, for symbolic model checking to be viable, the model checking algorithms must be based on symbolic expressions that can represent very large sets and relations efficiently.

The data structure that revolutionised model checking by allowing efficient representation of the sets of states generated during verification, was the Binary Decision Diagram (BDD) [16]. BDDs are canonical graph structures that can encode many sets and relations compactly. The symbolic algorithms that resulted from combining traditional model checking algorithms with BDDs turned out to be able to handle many industrial systems. Success stories include the verification of a pipelined ALU with 10^{20} states [18], the verification of the Future Bus+ protocol [20], and the verification of a one million gate ASIC that implemented part of a cache coherence protocol [34].

BDDs work extremely well for many systems. Unfortunately, they still have drawbacks. First, there are some hardware systems, such as multiplier circuits, that BDDs cannot represent in subexponential space. Second, BDDs need the user to provide a variable ordering that can make the difference between being able to build the BDD in realistic space or not. Third, BDDs can rarely cope with more than a few hundred state variables. As a consequence of these drawbacks, the formal verification community has been interested in discovering alternatives to BDDs that can cope with systems that cannot be represented succinctly with BBDs under any variable order, or that contain too many state variables.

In parallel with the development of symbolic model checking, other researchers concentrated on verification based on satisfiability (SAT) solvers [93, 14, 15, 42, 87]. Satisfiability solvers are algorithms that decide whether a propositional logic formula has a satisfying assignment or not; many verification problems can be cast in terms of the satisfiability (or unsatisfiability) of formulas. SAT-solvers have some interesting characteristics that seem to complement BDDs well. Propositional logic formulas can represent any gate level netlist in polynomial space, and in addition, SAT solvers do not normally need an externally provided variable ordering. Finally, SAT solvers have been used to verify systems containing thousands of variables.

Recently, the use of SAT solvers instead of BDDs in symbolic model checking has received much attention. This approach has also generated good results for some of the systems that are known to be hard to verify using BDDs.

In this paper, we present a self-contained tutorial on the underlying ideas behind three of these methods for SAT-based model checking [6, 2, 86]. We also try to explain similarities and differences between the methods, and present some research directions that we think will be interesting.

Our aim is to focus exclusively on explaining the methods. We have therefore decided not to discuss specific SAT solvers. This means that a detailed analysis of the performance of the methods we describe is beyond the scope of this paper. For experimental evidence of the usefulness of the techniques, we therefore refer the reader to the original papers. We also restrict our discussion to the verification of safety properties of synchronous hardware; this is representative for how the methods have been used in practice up to now. Readers interested in learning more about the theory of BDD-based symbolic model checking are advised to read Clarke, Grumberg, and Peled's excellent textbook [22].

8.2 Preliminaries

In the following we will model and verify circuits using propositional logic. Propositional logic formulas are made up from (1) variables such as x and y, (2) the connectives $\neg$, $\wedge$, $\vee$, $\leftrightarrow$, and $\rightarrow$, and (3) parentheses. The formulas may also refer to the values *True* and *False*.

Example 5 (Some propositional logic formulas) The following are three examples of propositional logic formulas.

1. $a \leftrightarrow (a \vee b)$
2. $(a \wedge b) \rightarrow a$
3. $a \wedge \neg a$

The first formula states that a has the same value as the logical or of a and b, the second formula states that a and b implies a, and the third formula states that a and the negation of a both hold. $\square$

An *assignment* for a formula is a binding of each variable in the formula to true or false. For a given assignment, the whole formula will thus evaluate to either true or false. An assignment that makes a formula true is called a *satisfying assignment*, or equivalently, a *model* of the formula.

The job of a SAT-solver is to take a formula, and compute a satisfying assignment if one exists. Formulas that have no satisfying assignment are called *unsatisfiable*, or *contradictory*. Formulas that only have satisfying assignments are called *tautologies*. We can use a SAT-solver to check whether a given formula is a tautology by checking whether the negation of the formula is unsatisfiable. This process is often referred to as *proving* the formula.

Example 6 (Status of the formulas in Example 5) Consider the formulas in Example 5. The first of them is satisfied when a is true and b is false. The second formula is satisfied by all assignments to a and b, so it is a tautology. The third formula is unsatisfiable. □

8.3 Modelling Systems in Propositional Logic

To be able to apply the system analysis methods described in this paper, we must find a way to model systems using propositional logic. We restrict our attention to finite state systems that are also purely boolean. To model such a system, we use propositional formulas to represent (1) the initial values of the state variables and (2) the transition function, which maps an input and an old state to a a new state.

We must produce the formulas for the initial states and the transition from descriptions of circuits or programs in standard design formats. To give some intuition about such translations, we consider the problem of how to generate formulas from a circuit in the form of a schematic diagram.

We will be concerned with circuit *states* and how they change. Think of the state of a circuit as being the values of its primary inputs and the contents of all of its delay elements. The possible initial values of the state are given by the initial values of the delay elements. The transition function expresses the new contents of the delay elements in terms of the old values and the primary inputs.

Example 7 (A simple toggle circuit, shown in Figure 8.1) The circuit contains two components, an inverter and a unit delay element (or latch) whose initial value is *True*.

The delay element is assumed to be clocked once per cycle, at which point it delivers its stored value on its output and reads a new value from its input into the (single bit) memory. So, this circuit outputs *True*, *False*, *True*, *False*, and so on ad inifinitum. The circuit has one state variable, a (shown in the diagram), and this is also the output. To express the behaviour of the circuit as

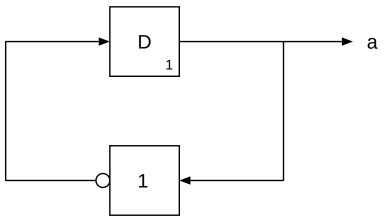

Figure 8.1: A simple toggle circuit called toggle$_1$

a formula, we must first characterise the initial value of a, which is represented by the propositional variable a_0. This is given by the initial value of the delay element, which is *True*. Thus,

$$\mathit{Init}_1(a_0) \quad = \quad a_0 \leftrightarrow \mathit{True}$$

which could also have been written

$$\mathit{Init}_1(a_0) \quad = \quad a_0$$

Next, we must express the new value of a after a cycle (which we will call a_{n+1}) in terms of its value before the cycle (a_n). The new value is given by the *input* to the delay element, and so is the negation of the old value (the output of the delay element) because of the presence of the inverter. So we find that for an arbitrary transition

$$T_1(a_n, a_{n+1}) \quad = \quad a_{n+1} \leftrightarrow \neg a_n$$

□

Example 8 (Changing the initialisation) If we change the circuit so that the initial value of the delay element is instead *False*, then we must change the initialisation to

$$\mathit{Init}_2(a_0) \quad = \quad \neg a_0$$

but the transition part remains unchanged: $T_2(a_n, a_{n+1}) = T_1(a_{n+1}, a_n)$. What is the behaviour of the resulting circuit, which we call toggle$_2$? □

Example 9 (Another toggle circuit, toggle$_3$**, shown in Figure 8.2)** This circuit contains two delay elements, called b and c, so the state is represented by the vector $\langle b, c\rangle$. The contributions of the delay elements to initialisation are *False* and *True* respectively, so

$$\mathit{Init}_3(\langle b_0, c_0\rangle) \quad = \quad \neg b_0 \wedge c_0$$

For the transition function, we name the inputs of the delay elements b_{n+1} and c_{n+1} respectively, and the outputs b_n and c_n, and write down the relationships

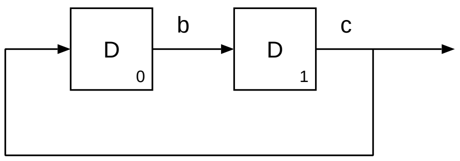

Figure 8.2: Another toggle circuit, toggle$_3$

implied by the rest of the circuit (the two wires joining the delay elements). The upper wire gives $c_{n+1} \leftrightarrow b_n$, while the lower one gives $b_{n+1} \leftrightarrow c_n$, so

$$T_3(\langle b_n, c_n\rangle, \langle b_{n+1}, c_{n+1}\rangle) \quad = \quad c_{n+1} \leftrightarrow b_n \ \wedge \ b_{n+1} \leftrightarrow c_n$$

This equation expresses the relationship between the old state $\langle b_n, c_n\rangle$ and the new one $\langle b_{n+1}, c_{n+1}\rangle$.

□

In larger circuits, it would become tedious to list all the elements of the state vectors, so a standard notation has been introduced. We use s to represent the vector of state variables, in this case $\langle b, c\rangle$, and s' to represent a primed version, in this case $\langle b', c'\rangle$. Then, we use s to represent the initial state in the equation for initialisation, and we use s for the old state and s' for the new in the transition formula. For toggle$_3$, $s = \langle b, c\rangle$ and

$$\begin{aligned} \mathit{Init}_3(s) \quad &= \quad \neg b \wedge c \\ T_3(s, s') \quad &= \quad c' \leftrightarrow b \ \wedge \ b' \leftrightarrow c \end{aligned}$$

None of the circuits considered so far had any inputs. Our final toggle circuit has a single input.

Example 10 (Yet another toggle, shown in Figure 8.3) When the input is *True*, the circuit toggles as before, but when it is *False* the delay element is instead set to *False*.

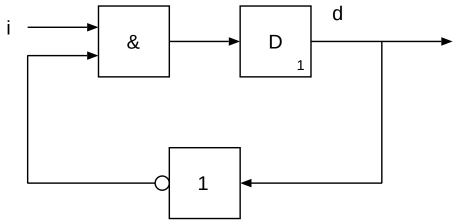

Figure 8.3: A toggle circuit with input i, *toggle*$_4$

For this circuit, the state s is the vector $\langle i, d\rangle$. The initialisation constrains d to be true, but does not constrain i.

$$Init_4(s) \quad = \quad d$$

So in fact there are two initial states, one for each of the possible values of i, that is $\langle \mathit{False}, \mathit{True}\rangle$ and $\langle \mathit{True}, \mathit{True}\rangle$.

For the transition formula, we give the new value of the delay element, d', in terms of the old value, d, and the primary input, i.

$$T_4(s, s') \quad = \quad d' \leftrightarrow (i \wedge \neg d)$$

□

In the translation that we have presented, the initialisation characterises the set of states that are compatible with the initial values of the delay elements. The transition formula characterises the possible moves from one state to another. To describe a run of a circuit, we first define what it means for a sequence of states to be a path through the transition graph.

$$Path[s_0, \ldots, s_k] \quad \equiv \quad \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

A run of a circuit with transition formula T then corresponds to a path starting in an initial state:

$$R_k \quad = \quad Init(s_0) \;\wedge\; Path[s_0, \ldots, s_k]$$

Example 11 (A run of length 4 of the circuit toggle_4**)** Any sequence of states s_0, s_1, s_2, s_3, s_4 that makes R_4 true when we use the initial states formula $Init_4$ and the transition formula T_4 is a possible trace for toggle_4. A simple such trace is $\langle \mathit{True}, \mathit{True}\rangle$, $\langle \mathit{True}, \mathit{False}\rangle$, $\langle \mathit{False}, \mathit{True}\rangle$, $\langle \mathit{False}, \mathit{False}\rangle$, $\langle \mathit{False}, \mathit{False}\rangle$. This sequence demonstrates that all of the possible states are in fact reachable from the initial states in this circuit. This is not always the case. □

Example 12 (The state transition diagram for toggle_3**)** The circuit toggle_3, shown in figure 8.2, also has two bits of state. But not all combinations of those two bits are reachable, starting at the initial state. Let us take the formulas $Init_3$ and T_3, and from them construct a *state transition diagram.*

A state consists of a pair of boolean variables, b and c, so there are four possible states, which we picture as circles with the $\langle b, c\rangle$ values inside. The initial state is $\langle \mathit{False}, \mathit{True}\rangle$, from $Init_4$, and we mark that with an incoming arrow. To decide what the state transitions should be (the arrows between states), we look for *models* of T_4. For example, setting s to $\langle \mathit{False}, \mathit{True}\rangle$ and s' to $\langle \mathit{True}, \mathit{False}\rangle$ makes $T_4(s, s')$ true. So, we draw an arrow from the state $\langle \mathit{False}, \mathit{True}\rangle$ to the

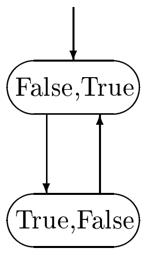

Figure 8.4: State diagram for toggle$_3$, reachable states only

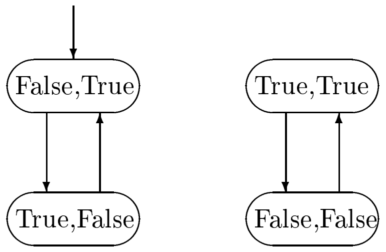

Figure 8.5: State diagram for toggle$_3$, reachable and unreachable states

state $\langle \mathit{True}, \mathit{False}\rangle$. Similarly, we find that there is an arrow back in the other direction. This gives us a diagram that captures the behaviour of toggle$_3$, as shown in Figure 8.4.

Figure 8.4 shows only the *reachable states*, those that can be reached via the transition relation from the initial state. Setting s to $\langle \mathit{True}, \mathit{True}\rangle$ and s' to $\langle \mathit{False}, \mathit{False}\rangle$ also makes $T_4(s, s')$ true. So we get an arrow from $\langle \mathit{True}, \mathit{True}\rangle$ to $\langle \mathit{False}, \mathit{False}\rangle$, and we find that there is also an arrow back in the other direction. This part of the state space is entirely *unreachable*. It is also possible to have unreachable states that have transitions to reachable states, but that is not the case in this example. Figure 8.5 shows the complete state transition diagram for toggle$_3$ whose circuit diagram is shown in Figure 8.2. □

As an exercise, you might like to construct the state transition diagram for toggle$_4$.

Later, when we consider algorithms for analysing the reachable states of a system, we will have to take account of the fact that the transition relation may also contain unreachable states.

8.3.1 Expressing Properties of Systems in Propositional Logic

We have seen how to model circuits using formulas. How do we express *properties* of our circuits or transition functions? We consider properties of *states*. You can think of a property and the set of states satisfying the property as being interchangeable. Such a property can be represented by a boolean formula, just as we represented the property of being the initial state by a formula (see $Init_1$ to $Init_4$). For example, if the state contains two boolean variables, d and e, then the property that they must be different is expressed by the formula $\neg(d \leftrightarrow e)$, and two of the four possible states obey the property. For toggle$_3$, whose state diagram we have just constructed, the two reachable states obey this property, but the unreachable ones do not.

If a particular valuation of the state variables is a model of a formula encoding a property P, then we say that the corresponding state satisfies P. Equivalently, we sometimes say that the state is a P-state. In the remainder of this paper, we adopt the view that a property and the set of states satisfying the property are interchangeable.

The most important kind of property that we want to prove is of the form P *is true of all reachable states.* Such properties are called *safety* properties, as they assert that all reachable states are good states, or equivalently, that nothing bad will ever happen in any run of the circuit starting from the initial states. Safety properties arise naturally when we compare two sequential circuits for equivalence. We can cast this problem as building a composite circuit containing both circuits as well as the combinational logic to check that corresponding outputs are equal. The two circuits are equivalent exactly when the output of the checking circuit is *True* in all reachable states.

We can also generalise this way of checking circuit equivalence to the notion of expressing properties as *observer circuits.* In that case, we have one copy of the circuit being analysed, as well as a checking circuit that keeps tabs on a particular property of the circuit that we would like to check. In general, this observer circuit will have as input both the inputs and the outputs of the circuit. It outputs *True* as long as the property holds, and *False* if it is violated. The important point, though, is that the checking circuit can be sequential (it can contain state-holding elements), so it can express quite complicated properties involving sequences of circuit steps. In this way, many useful and practically interesting circuit analyses can be reduced to the kind of safety property checking that we describe here. This idea for coding properties, which is called *synchronous observers*, is used, for example, in the verification of programs written in the synchronous dataflow language Lustre [47, 63].

An alternative to synchronous observers is to express properties in a special *temporal logic* designed for expressing properties over time. This has the disadvantage that the user needs to learn a new language, but has the advantage that a rich logic can be used to allow a larger range of properties to be expressed.

8.4 Using SAT-solvers to Verify Reachable State Invariants

We will now assume that we have been given a system modelled as shown in the previous section, and a property P. Our task is to decide whether P is a *reachable states invariant* for the system; that is, whether P holds in all states reachable from the initial states. We will call a system P-*safe* when P is a reachable states invariant of the system.

8.4.1 Reachability Analysis

Ordinary induction is a simple verification approach that can be implemented using propositional logic theorem proving. The idea behind induction is to reduce the verification that all reachable states satisfy P to a proof that the initial states satisfy P and a proof that all immediate successors of P-states satisfy P:

$$\begin{array}{rl} \mathit{Init}(s) \rightarrow P(s) & \qquad \text{Base case} \\ P(s) \wedge T(s,s') \rightarrow P(s') & \qquad \text{Step} \end{array}$$

Both proofs can be carried out using a SAT-solver. Induction is a *sound* rule for deciding P-safety: P must hold in every reachable state if both proof conditions are provable.

Example 13 (Proving circuit equivalence by induction) We use induction to prove that the outputs of toggle$_1$ and toggle$_2$ are always different. This is done by constructing a composite transition function where the state s is the pair of bits $\langle a, b\rangle$, one from toggle$_1$ and one from toggle$_2$

$$\begin{array}{rcl} \mathit{Init}_5(s) & = & a \wedge \neg b \\ T_5(s,s') & = & a' \leftrightarrow \neg a \;\wedge\; b' \leftrightarrow \neg b \end{array}$$

The required property is then $P(s) = \neg(a \leftrightarrow b)$, which leads us to the following proof conditions:

$$\begin{array}{rl} a \wedge \neg b \;\rightarrow\; \neg(a \leftrightarrow b) & \qquad \text{Base case} \\ \neg(a \leftrightarrow b) \wedge (a' \leftrightarrow \neg a \;\wedge\; b' \leftrightarrow \neg b) \;\rightarrow\; \neg(a' \leftrightarrow b') & \qquad \text{Step} \end{array}$$

The base case and the step are easily proven to be tautologies. □

The soundness of induction guarantees that if we can prove a system correct using the proof rule, then the system is indeed correct. However, standard induction is too weak to be *complete*; there are systems that are P-safe, but that *cannot* be proven correct using the scheme. The induction step cannot be proven in a system that contains an unreachable P-state with an immediate non-P successor.

Example 14 (A failed inductive proof) We attempt to show that toggle$_1$ (Figure 8.1) and toggle$_3$ (Figure 8.2) are equivalent. This time the state consists of three bits: $s = \langle a, b, c\rangle$. The resulting transition function is

$$\begin{aligned} Init_6(s) &= a \wedge \neg b \wedge c \\ T_6(s, s') &= a' \leftrightarrow \neg a \;\wedge\; b' \leftrightarrow c \;\wedge\; c' \leftrightarrow b \end{aligned}$$

The required property is $P(s) = a \leftrightarrow c$. This time, however, the base case goes through but the step does not. Looking at the transition function, we can see that knowing that $a \leftrightarrow c$ before the transition does not allow us to conclude that $a' \leftrightarrow c'$ after it. If b is equivalent to a, then we find that c' and a' must be different from each other, so that $P(s')$ is false. Thus, it is possible to go from a state satisfying the property to one not satisfying it. But this happens only in the unreachable part of the state space: states in which a, b and c all have the same value can not be reached from the initial states. However, it is enough to scupper our inductive proof, despite the fact that the two circuits are in fact equivalent in their reachable behaviour. □

Fortunately, a simple extension of induction circumvents the incompleteness. We refer to this extension as *generalised induction.* The generalised rule requires the user to supply a formula *Inv* that both implies P and allows proofs of the base case and the step:

$$\begin{aligned} Init(s) &\rightarrow Inv(s) && \text{Base case} && (1) \\ Inv(s) \wedge T(s, s') &\rightarrow Inv(s') && \text{Step} && (2) \\ Inv(s) &\rightarrow P(s) && \text{Strengthening} && (3) \end{aligned}$$

The extended rule can easily be seen to be sound: *Inv* holds in the reachable state space if the base case and the step are provable, and the proof of strengthening guarantees that this is a sufficient condition for P-safety. The strengthened rule is also complete as we can always take *Inv* to be a formula *Reachable* that characterises the reachable states. To see that this choice of *Inv* gives a complete rule, observe that (1) the initial states are reachable, (2) successors of reachable states are also reachable and (3) the reachable states in a P-safe system satisfy P by definition. All of the proof conditions are thus provable. The existence of formulas that characterise the reachable states is guaranteed by the finiteness of the system; every state in a finite system can be encoded as a formula that characterises the state (and no others), so an arbitrary finite set can be described using a finite disjunction.

Example 15 (Characterising the reachable states of toggle$_3$**)** Consider the state space of toggle$_3$ in Figure 8.5. The reachable state space is the set of states $\{\langle \mathit{False}, \mathit{True}\rangle, \langle \mathit{True}, \mathit{False}\rangle\}$. The first of these states is encoded by the formula $\neg b \wedge c$, and the second state is encoded by $b \wedge \neg c$. The reachable states are therefore characterised by the formula $(\neg b \wedge c) \vee (b \wedge \neg c)$. □

The formula *Reachable* has an interesting property: Every strengthening that can be used for generalised induction must contain the reachable states as a subset. The reason for this is that the proof rule requires that the strengthening include the initial states, and that the strengthening be closed under the transition relation. As a consequence, the property *Reachable* is the *least* possible strengthening.

Dually, there is a *largest* strengthening that satisfies all the conditions: the set of P-invariant states, $PInv$. This is the set containing all the states from which only P-states can be reached. The P-invariant states form the *largest* strengthening, as all properties P' that fulfil the proof conditions can only contain P-invariant states. If this were not the case it would be possible to reach a non-P state by finitely many transitions from a state in P', and this is impossible due to (2) and (3).

The set of P-invariant states serves as a strengthening as all initial states in a P-safe system are P-invariant by the base case (equation (1)), P-invariant states by definition only lead to other P-invariant states, by the step (equation (2)), and P-invariant states are trivially P-states, by equation (3) .

So, generalised induction is a simple, yet sound and complete, method for proving P-safety of finite systems. This rule allows us to reduce the real safety verification problem to the computation of a strengthening. In particular, we can reduce the problem to computing either the reachable states or the P-invariant states.

Let us consider how to produce the formula *Reachable*, which is perhaps the most obvious of the strengthenings. It makes sense to build up a formula characterising the reachable states iteratively, starting with the formula that characterises the initial states and then gradually adding new states. We will do this by mimicking a breadth first construction of increasing subsets R_i of the reachable states:

$$\begin{aligned} R_0 &= \mathit{Init} \\ R_{n+1} &= \mathit{Init} \cup \{s \mid \exists r.\ r \in R_n \wedge T(r,s)\} \end{aligned}$$

Each set R_i is the set of states reachable in i or fewer steps from the initial states; R_0 is the set of initial states, and each R_{n+1} is the union of the initial states and the set of states reachable in one transition from R_n. As the sequence of sets generated is monotonically increasing in the sense that $R_0 \subseteq R_1 \subseteq R_2 \ldots$, and there are only finitely many states in the state space, the sequence must eventually stabilise. Thus, for some i, it is the case that $R_i = R_{i+1}$. This fixpoint is the set of all reachable states.

Example 16 (Generating the set of reachable states of toggle_3**)** Let us generate the reachable states of toggle_3. R_0 is the set of initial states, so $R_0 = \{\langle \mathit{False}, \mathit{True} \rangle\}$. R_1 is the union of the initial states and the set of states that can be reached from R_0 in one step. The only state that can be reached from R_0

in one step is $\langle \mathit{True}, \mathit{False}\rangle$, so $R_1 = \{\langle \mathit{False}, \mathit{True}\rangle, \langle \mathit{True}, \mathit{False}\rangle\}$. As $R_0 \neq R_1$, the construction has not stabilised, so we must compute R_2. However, $R_2 = R_1$ as the states that can be reached from R_1 in one transition are already part of R_1. We have thus arrived at the reachable states. $\square$

We now know how to construct the *set* of reachable states. However, we really want to generate a sequence of *formulas*, $\phi_i(s)$, that characterises the sets R_i without actually building any sets. Remember that systems containing 10^{20} reachable states are not uncommon.

The set of initial states is represented by *Init* so the construction of $\phi_0(s)$ is trivial. However, the construction of a formula that characterises R_{n+1} is more involved. Given the characteristic formula of R_n, we need to generate the characteristic formula of $\mathit{Init} \cup \{s \mid \exists r.\ r \in R_n \wedge T(r,s)\}$. The set union is easy to mimic using disjunction of characteristic formulas. But the formula characterising the set $\{s \mid \exists r.\ r \in R_n \wedge T(r,s)\}$ is $\exists r.\ \phi_n(r) \wedge T(r,s)$, where the quantification over r corresponds to an existential quantifier for each boolean variable in the vector r. This formula is what is known as a *quantified boolean formula* (QBF). Fortunately, such a formula can be translated into propositional logic, and we call the translation method $\mathit{Reduce}(\cdot)$. Such a translation method must exist as a quantified boolean variable p can be removed using the transformation $\exists p.\phi(p) \Rightarrow \phi(\mathit{True}) \vee \phi(\mathit{False})$. The characteristic formulas ϕ_i of the sets R_i are generated using the following schema:

$$\begin{aligned}\phi_0(s) &= \mathit{Init}(s)\\ \phi_{n+1}(s) &= \mathit{Init}(s) \vee \mathit{Reduce}(\exists r.\ \phi_n(r) \wedge T(r,s))\end{aligned}$$

As there exists an i for which $R_i = R_{i+1}$, it holds for this i that ϕ_i is provably equivalent to ϕ_{i+1}. We can therefore choose *Reachable* to be this ϕ_i, and determine whether a system is P-safe by attempting to prove the strengthening condition $\mathit{Reachable}(s) \rightarrow P(s)$. (The reachable states fulfil the base case and step condition by definition). A successful proof allows us to conclude that the system is P-safe, and P-safe systems must conversely have provable base cases.

We can also optimise the method further. Recall that each constructed formula ϕ_j characterises a subset of the states that are reachable from the initial states. Every time we construct a new formula ϕ_j, we can check whether there is some non-P state in the underlying set by attempting to satisfy the formula $\phi_j(s) \wedge \neg P(s)$. If we succeed, the system is obviously not P-safe: reachability in j or fewer steps trivially implies reachability. The resulting verification algorithm that incorporates this optimisation is called *forwards reachability analysis* (Algorithm 1). In the presentation of the algorithm, we make use of two operations, $Sat(\cdot)$ and $Taut(\cdot)$. These two operations take a formula as argument and return true when the formula is satisfiable or is a tautology, respectively.

Just as the formula *Reachable* can be constructed by a fixpoint iteration, it is possible to generate the property *PInv* by a fixpoint construction on another sequence of sets S_n. Each S_i is the set of states from which all paths of length

Algorithm 1 Check if system is P-safe by forwards reachability analysis

```
$\phi(s) = \phi_{old}(s) = Init(s)$
**repeat**
  **if** $Sat(\phi(s) \wedge \neg P(s))$ **then**
    **return** False
  **end if**

  $\phi_{old}(s) = \phi(s)$
  $\phi(s) = Init(s) \vee Reduce(\exists r.\phi_{old}(r) \wedge T(r,s))$
**until** $Taut(\phi(s) \leftrightarrow \phi_{old}(s))$
**return** $Taut(\phi(s) \rightarrow P(s))$
```

i contain only P-states.

$$\begin{aligned} S_0 &= P \\ S_{n+1} &= P \cap \{s \mid \forall t.\ T(s,t) \rightarrow t \in S_n\} \end{aligned}$$

In the formation of S_{n+1}, we first compute the set of states that have all successors in S_n, and then remove all non-P states. Our construction is correct as all paths from S_{n+1} then start in a P-state and immediately go on into S_n. This ensures that the length-n suffixes of paths contain only P-states. The sequence is monotonically decreasing as the starting points of paths of length i must be a superset of the starting points of paths of length j if $i < j$. As the number of states in the system is finite, the construction has a fixpoint which must be the property $PInv$.

Once the formula characterising the P-invariant states has been computed, we know by the fixpoint properties of the construction that we only have to check whether the generalised induction base case $Init(s) \rightarrow PInv$ (s) is provable. Furthermore, if a characteristic formula ψ_k for the set S_k makes the formula $\neg\psi_k(s) \wedge Init(s)$ satisfiable, there is an initial state from which a non-P state can be reached by a path of length k. Then, the fixpoint construction can be terminated as the system is clearly not P-safe.

Traditionally, the complement of $PInv$ is constructed, rather than $PInv$ itself. This is done by a fixpoint construction on the complemented sequence S'_i:

$$\begin{aligned} S'_0 &= S \setminus P \\ S'_{n+1} &= (S \setminus P) \cup \{s \mid \exists t.\ T(s,t) \wedge t \in S'_n\} \end{aligned}$$

S is the entire set of states, both reachable and unreachable, and the $\setminus$ operator is set subtraction. The set S'_k contains the states that can reach a $\neg P$-state by a path of length k or shorter. This sequence of sets must be monotonically increasing as the sequence of their complements is monotonically decreasing. When we reach a fixpoint, we have the set of states that can reach a $\neg P$-state by a path of arbitrary length. Just as in forwards reachability, we actually perform this computation using formulas rather than explicit sets. The resulting verification method (Algorithm 2) is called *backwards reachability analysis*; we compute the set of states that are backwards reachable from the states violating P, rather than the set of states forwards reachable from *Init*.

Algorithm 2 Check if system is P-safe by backwards reachability analysis

```
$\psi(s) = \psi_{old}(s) = \neg P(s)$
**repeat**
  **if** $Sat(\psi(s) \wedge Init(s))$ **then**
    **return** False
  **end if**

  $\psi_{old}(s) = \psi(s)$
  $\psi(s) = \neg P(s) \vee Reduce(\exists t.\ \psi_{old}(t) \wedge T(s,t))$
**until** $Taut(\psi(s) \leftrightarrow \psi_{old}(s))$
**return** $Taut(Init(s) \rightarrow \neg\psi(s))$
```

8.4.2 Bounded Model Checking

Recall that the simple induction scheme presented in section 8.4.1 was incomplete, as the unreachable part of the state space could contain bad sequences of states. Our solution was to adopt generalised induction, a proof rule that relied on the invention of a new property that implied P and removed the troublesome part of the unreachable states space from consideration. This is one way to solve the problem. However, there is a simple, alternative method that avoids the invention of such a predicate altogether, and still circumvents problems with the unreachable state space.

Assume, to begin with, that we are willing to settle for a method that verifies that all length k paths that originate in the initial states contain only P states. We refer to such paths as k-bounded P-paths. As we are no longer interested in paths of arbitrary length, we can "unwind" the inductive argument into a single formula, that we refer to as Safe_k:

$$Init(s_0) \wedge Path[s_0, \ldots, s_k] \rightarrow \bigwedge_{i=0}^{k} P(s_i) \tag{8.1}$$

The formula Safe_k is provable if and only if all paths of length bounded by k from the initial states are P-paths; we elegantly avoid problems with unreachable states as we explicitly model paths that start in the initial states. P-safety verification methods that are based on proving formulas that code bounded paths are special instances of *Bounded Model Checking* [6].

Example 17 (Bounded model checking of toggle_3**)** Consider the circuit toggle_3 (Figure 8.2). If we are interested in the property that the two bits of state are always equal, then Safe_1 becomes

$$(\neg b_0 \wedge c_0) \wedge (c_1 \leftrightarrow b_0 \wedge b_1 \leftrightarrow c_0) \rightarrow (b_0 \leftrightarrow c_0 \wedge b_1 \leftrightarrow c_1)$$

□

If the formula Safe_k is falsifiable (as is the case in Example 17), the theorem prover will return a countermodel that encodes a counterexample of length bounded by k. This information is very useful for debugging purposes. Many bugs are detectable by comparably short counterexamples, so a relatively low value of k can often be sufficient to detect the presence of errors. However, it would be even more satisfying to know that there exists some choice of k that we can use to detect the *absence* of counterexamples.

Let us find a sufficient k. Given two states s and t in the system S, we know that if there exists one or more paths from s to t, at least one of these paths is a shortest path. As we are verifying systems with finitely many states, and a shortest path visits an arbitrary state at most once, the length of such shortest paths is bounded by the number of states $|S|$. But if we are to reach a state at all, we must reach it through some shortest path. Therefore, if we have determined that there are no counterexamples of length $|S|$ or shorter, there cannot be any paths at all to non-P states from the initial states. Consequently, $k = |S|$ is a sufficient bound to decide P-safety by the provability of Safe_k.

Real systems often contain millions of states, so choosing k to be equal to $|S|$ is seldom practical. Fortunately, we can derive a much better bound. All shortest paths have finite length, and there are only finitely many pairs of states. We can therefore construct the set containing the lengths of shortest-paths between all states. This set contains finitely many natural numbers, so it must have a maximum element d. We refer to this element as the *system diameter*. The system diameter bounds the length of shortest paths; a proof of Safe_d therefore implies P-safety. For many classes of systems, d is an exponentially better choice of k than $|S|$.

Example 18 (Possible values of k for toggle_3**)** Consider the state space of toggle_3 in Figure 8.5. The diagram contains four states, so $|\text{toggle}_3| = 4$. The maximum length of a shortest path between two states is one, so $d = 1$. Consequently, the provability of Safe_1 is sufficient to decide the safety of toggle_3, and Safe_3 would also have been a possible (but worse) choice. □

A nice feature of the system diameter is that we can determine it directly from the propositional logic description of the system. The idea for computing d takes advantage of an equivalent definition of system diameter: the least number n such that for all states s and t, a path of length $n + 1$ between s and t implies the existence of a strictly shorter path connecting s to t.

Example 19 Again, consider Figure 8.5. It is easy to see that the existence of a length 0 or length 1 path between two states in this state space does not guarantee the existence of a strictly shorter path. However, if two states are connected by a length 2 path, there exists a strictly shorter path that connects them. Consequently $d = 1$ by the alternative definition of diameter. □

The alternative diameter definition is phrased in terms of finite length paths, so it can be encoded as a quantified boolean formula. The literal translation of the alternative definition is that the diameter is the least n that makes the following formula Diam_n provable:

$$\bigwedge_{i=0}^{n} T(s_i, s_{i+1}) \rightarrow \exists t_0,\ldots,t_n.\ (t_0 = s_0 \wedge \bigwedge_{i=0}^{n-1} T(t_i, t_{i+1}) \wedge \bigvee_{i=0}^{n} t_i = s_{n+1})$$

So, we can determine the system diameter by attempting to prove $\mathit{Reduce}(\mathrm{Diam}_n)$ for increasing values of n; when the formula is provable, the diameter has been found.

By combining an iterative search for the diameter of a system with the verification of bounded P-safety, we arrive at a sound and complete algorithm for deciding P-safety (Algorithm 3). The algorithm performs two tasks in iteration i: it checks that all paths of length i from the initial states globally contain P-states, and it attempts to prove that i is a sufficient bound. If a counterexample

Algorithm 3 Check if system is P-safe by bounded model checking

```
i=0
while True do
  if not $Taut(\mathrm{Safe}_i)$ then
    return Countermodel $[s_0, \ldots s_i]$
  end if

  if $Taut(Reduce(\mathrm{Diam}_i))$ then
    return True
  end if
  $i = i + 1$
end while
```

to P-safety is found in some iteration, the algorithm terminates; this guarantees that the generated counterexamples are minimal. This is important because the shorter the counterexample, the easier it is to determine the cause of the error. The algorithm can be modified to increment i more aggressively, if minimality can be forfeited. Provable Diam_i and Safe_i formulas entail P-safety even if i is larger than the smallest necessary value.

8.4.3 Induction

Earlier, when discussing reachability analysis in section 8.4.1, we saw that induction provides a simple solution to P-safety checking. But we have also seen that this proof method sometimes fails, even when the property to be proved is a reachable state invariant. How can we fix this? Standard induction focuses on proving that properties are preserved across one transition. A systematic, yet simple, way to strengthen the inductive scheme is to consider more than one transition at a time.

Example 20 (Considering more than one transition at a time) Let us reconsider the failed proof of equivalence between toggle$_1$ and toggle$_3$ from Example 14. This time we prove instead that the property holds in the first two states, and that if it holds in two states in a row then it holds in the next state. The base case and step are

$$\begin{array}{rl} \mathit{Init}(s_0) \wedge T(s_0, s_1) \rightarrow P(s_0) \wedge P(s_1) & \text{Base case} \\ P(s_i) \wedge P(s_{i+1}) \wedge T(s_i, s_{i+1}) \wedge T(s_{i+1}, s_{i+2}) \rightarrow P(s_{i+2}) & \text{Step} \end{array}$$

The base case is again trivially true. The new step is also a tautology. In fact, in this example it turns out to be sufficient to assume $P(s_i)$ to be able to prove $P(s_{i+2})$, but in general one will need both $P(s_i)$ and $P(s_{i+1})$. $\square$

We call this new proof method *induction with depth* 1. The method generalises to induction with depth k.

$$\begin{array}{lcrl} \text{Safe}_k & \equiv & \mathit{Init}(s_0) \wedge \mathit{Path}[s_0 \ldots s_k] \rightarrow \bigwedge_{i=0}^{k} P(s_i) & \text{Base case} \\ \text{Step}_k & \equiv & \bigwedge_{i=0}^{k} P(s_i) \wedge \mathit{Path}[s_0 \ldots s_{k+1}] \rightarrow P(s_{k+1}) & \text{Step} \end{array}$$

So a possible strategy for checking a safety property is to start with induction with depth 0, which is just ordinary induction, and then try depth 1, depth 2, and so on. We note that the base case of induction with depth k is exactly Safe$_k$, the formula that is used to look for counterexamples in bounded model checking.

The sad fact, though, is that although induction with depth $k + 1$ is strictly stronger than induction with depth k, we are not guaranteed to find a k that is sufficient to prove a given P-safe system correct! The method is not complete. There are P-safe systems in which one can find paths in which the first k states satisfy P and the final state does not, for all k. These troublesome paths are in the unreachable state space, and they arise because of the presence of loops in paths.

Example 21 (A failure for ordinary induction) Consider the state transition diagram shown in Figure 8.6. On the left are the reachable states, all of which satisfy the property P. The initial state is marked with an I. On the right are the unreachable states, some of which satisfy P, and some of which don't. And no matter what k we pick, there is a path in the unreachable states that

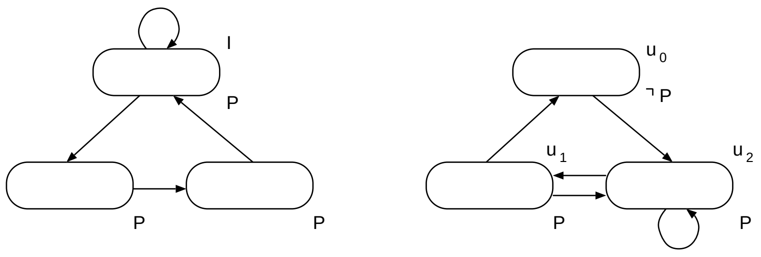

Figure 8.6: A state transition system that defeats ordinary induction with depth

starts with k states that satisfy P, and finishes with a state that does not satisfy P. The result is that the proof of the step never succeeds for any k, and we are stuck. Examples of troublesome paths are $[u_1, u_0]$, $[u_2, u_1, u_0]$, $[u_1, u_2, u_1, u_0]$, $[u_2, u_2, u_1, u_0]$, $[u_2, u_1, u_2, u_1, u_0]$, $[u_1, u_2, u_2, u_2, u_2, u_2, u_1, u_0]$; the list is endless. Even though the system is P-safe, this example defeats induction with depth k for all k. □

What can we do to make induction with depth complete? The answer is that we can restrict the sequences of states (or paths) that we consider in the step to be *loop free*. Or, to put it another way, we demand that the states in the unrolling of the transition function in the step all be different. Considering only loop free paths in the step still allows us to examine the same set of states, but it removes the kind of troublesome paths that we have just seen. In the above example, the first two troublesome paths that we noticed ($[u_1, u_0]$ and $[u_2, u_1, u_0]$) were loop free, but all of the others contain repeats. So by increasing k (in this case to 3), we can get to a point where troublesome paths in the unreachable states no longer interfere with the proof. And in the reachable states, we find that the first k states all satisfy P, and that k (different) states in a row satisfying P always lead to another state satisfying P. So, by induction, the reachable states all satisfy P. Thus, the base case of this strengthened induction is as before (Safe$_k$). We change the step to consider only loop free paths, by adding an additional constraint. The new step is then

$$\begin{aligned}
\text{LoopFree}[s_0 \ldots s_n] \quad &\equiv \quad Path[s_0 \ldots s_n] \wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j \\
\text{Step}'_k \quad &\equiv \quad \bigwedge_{i=0}^{k} P(s_i) \wedge \text{LoopFree}[s_0 \ldots s_{k+1}] \;\; \rightarrow \;\; P(s_{k+1})
\end{aligned}$$

What does the additional constraint that all states must be different in a path look like in practice? Each s_i is actually a vector of boolean variables, $\langle s_i(0), \ldots, s_i(m) \rangle$, so inequality between states corresponds to a disjunction of inequalities between the corresponding boolean variables. As a consequence, $s_i \neq s_j$ actually means $\bigvee_{l=0}^{m} \neg(s_i(l) \leftrightarrow s_j(l))$. We now have all the pieces needed to build a sound and complete verification algorithm.

Our strategy is to start with strengthened induction of depth 0 and then increment the depth until both the base case and the step go through. This is guaranteed to terminate. If one or more of the reachable states fails to satisfy P, then we find a countermodel $[c_0 \dots c_k]$ when we try to prove the base case at a particular k. The countermodel gives a shortest path starting at the initial state and ending in a state that does not satisfy P. So we are guaranteed to terminate if the system can reach a state violating P. When do we terminate if all reachable states satisfy P? Let *lfb* be the length of the longest loop free path ending in a state violating P, and with all other states satisfying P. Such a path must be in the unreachable states. The current algorithm iterates until k becomes equal to *lfb*. We can think of improving the algorithm slightly by making sure that we stop when we reach the length *lff* of the longest loop free path starting from an initial state, and proceeding through non-initial states. We can stop then because in that case the *base case* has checked all reachable states. We define

$$\text{ForwardTerm}_k \quad \equiv \quad \mathit{Init}(s_0) \wedge \bigwedge_{i=1}^{k+1} \neg \mathit{Init}(s_i) \wedge \text{LoopFree}\,[s_0 \dots s_{k+1}]$$

and can stop when ForwardTerm_k first becomes unsatisfiable. Then, algorithm 4 terminates when k reaches either *lfb* or *lff*, whichever is the smaller. To make the presentation more symmetrical, we can note that the negation of Step'_k can be written as

$$\text{BackTerm}_k \quad \equiv \quad \neg P(s_{k+1}) \wedge \bigwedge_{i=0}^{k} P(s_i) \wedge \text{LoopFree}[s_0 \dots s_{k+1}]$$

and that Step'_k is a tautology when its negation is unsatisfiable.

Algorithm 4 Check if system is P-safe by strengthened induction with depth

```
k=0
while True do
  if not $Taut(\text{Safe}_k)$ then
    return Countermodel $[c_0 \dots c_k]$
  end if
  if not $Sat(\text{BackTerm}_k)$ or not $Sat(\text{ForwardTerm}_k)$ then
    return True
  end if
  $k = k + 1$
end while
```

We summarise the resulting method in pseudo-code in Algorithm 4. The formulas to be checked are all in propositional logic, so that it suffices to use a propositional logic theorem prover to do satisfiability and tautology checking on the formulas for each iteration. This is a great advantage of the method.

The disadvantage is that it may iterate unnecessarily far. We have seen, for example, that in a P-safe system, it may iterate up to the length of the longest loop-free path whose first state is an initial state. But it might very well have actually checked all states along that path much earlier, having reached them by shorter paths. We would like to consider *shortest* paths between pairs of states, rather than loop-free ones. How do we express the fact that $[s_0 \dots s_{k+1}]$ is a shortest path between s_0 and s_{k+1}? We say that it is a path, and that there is no shorter path.

$$\begin{aligned}\text{Shortest}[s_0 \dots s_{k+1}] \quad \equiv \quad & \mathit{Path}[s_0 \dots s_{k+1}] \ \wedge \\ & \neg \exists t_0, \dots, t_k.\ (t_0 = s_0 \wedge \mathit{Path}[t_0 \dots t_k] \wedge \bigvee_{i=0}^{k} t_i = s_{k+1})\end{aligned}$$

Note that this is the negation of the formula Diam_k, which we have seen in the context of bounded model checking. This formula contains quantifiers, so we need to apply $\mathit{Reduce}(\cdot)$ to it in order to produce a formula in propositional logic, just as we did in methods described in section 8.4.1. So, we modify our two termination conditions to replace LoopFree by Shortest. (The base case is as before.)

$$\begin{aligned}\text{Safe}_k \quad & \equiv \quad \mathit{Init}(s_0) \wedge \mathit{Path}[s_0 \dots s_k] \rightarrow \bigwedge_{i=0}^{k} P(s_i) \\ \text{ForwardTerm}'_k \quad & \equiv \quad \mathit{Init}(s_0) \wedge \bigwedge_{i=1}^{k+1} \neg \mathit{Init}(s_i) \wedge \mathit{Reduce}(\text{Shortest}\,[s_0 \dots s_{k+1}]) \\ \text{BackTerm}'_k \quad & \equiv \quad \bigwedge_{i=0}^{k} P(s_i) \wedge \neg P(s_{k+1}) \wedge \mathit{Reduce}(\text{Shortest}\,[s_0 \dots s_{k+1}])\end{aligned}$$

The resulting algorithm has exactly the same shape as before. We give the pseudo-code in Algorithm 5 for completeness.

Algorithm 5 Check if system is P-safe by strong induction with depth

```
k=0
while True do
  if not $Taut(\text{Safe}_k)$ then
    return Countermodel $[c_0 \dots c_k]$
  end if
  if not $Sat(\text{BackTerm}'_k)$ or not $Sat(\text{ForwardTerm}'_k)$ then
    return True
  end if
  $k = k + 1$
end while
```

In between algorithms 4 and 5, there is a range of induction-based algorithms of increasing strength. These algorithms are briefly presented in reference [86], where an industrial application of algorithm 4 to the verification of Field Programmable Gate Array cores is also described.

8.5 Implementing the Analyses

In order to implement the model checking algorithms efficiently, we must solve two problems:

- The formulas we generate have to be compacted or built in such a way that unnecessarily large formulas are avoided as often as possible. To see that this is important, note that the formulas that are generated in the algorithms arise by a repetitive process. In reachability analysis, for example, each iteration concatenates a new formula to the result of a Quantified Boolean Formula (QBF) translation. The generated formulas can therefore quickly grow to be very large and may contain many redundancies such as repeated subformulas.
- QBF formulas must be translated to quantifier-free formulas without unnecessarily incurring an exponential size blowup.

The two problems are not orthogonal. In order to avoid intractable formulas for algorithms such as reachability analysis, we must have a good QBF translator, and a good formula compactor is certainly valuable when we implement a QBF translator.

In this section we investigate how the tool FixIt addresses the two problems. FixIt [2, 32] is a workbench for SAT based model checking that initially was written at Uppsala University and Prover Technology. It is now being developed at Chalmers University of Technology.

We would like to emphasise that what we present here is but one solution; readers interested in an alternative approach are referred to [99].

8.5.1 Formula Representation

FixIt uses a data structure for formula representation that attempts to combine a directed acyclic graph representation (DAG) for formulas with simple reductions. The name of the representation is motivated by the fact that the underlying graph can be seen as a combinational circuit: edges correspond to connections, operator nodes and markers correspond to primitive gates and variable nodes correspond to inputs.

Given a formula ψ that characterises a set S, it is clear that there are arbitrarily large formulas such as $\psi \wedge \psi \ldots \wedge \psi$ that characterise the same set. FixIt uses reductions to rewrite formulas so that such obvious redundancies are avoided. In order to achieve further reduction, FixIt also attempts to make the rewriting

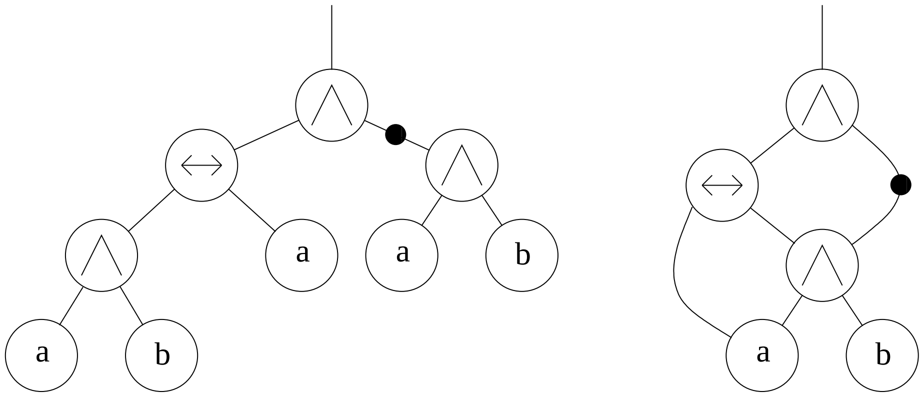

Figure 8.7: Tree and DAG representation of the formula $(a \wedge b \leftrightarrow a) \wedge \neg(a \wedge b)$.

rules applicable as often as possible. For example, the left hand side of the formula $(\phi \wedge \psi) \vee (\psi \wedge \phi)$ does not exactly match the right hand side, so the rule $a \vee a \;\Rightarrow\; a$ is not immediately applicable. But if the left-hand side of the disjunction is rewritten from $(\phi \wedge \psi)$ to $(\psi \wedge \phi)$, the rule will match. So, it is important to rewrite formulas by rearranging subformulas whenever the connective is symmetric.

The following are some examples of simple semantics-preserving transformations on formulas:

1. $\mathit{False} \rightarrow \phi \;\Rightarrow\; \mathit{True}$
2. $\phi \wedge \phi \;\Rightarrow\; \phi$
3. $\neg\phi \leftrightarrow \neg\psi \;\Rightarrow\; \phi \leftrightarrow \psi$

The first two rewrite rules are members of particular families of reductions: all formulas containing truth values can be reduced, as can any formula in the form $\tau \circ \tau$ for $\circ$ a boolean connective. The third reduction is more specific, but is still useful as it rewrites occurrences of equivalences into a standard form.

The use in FixIt of a representation built on directed acyclic graphs allows a more efficient formula encoding than a representation built on trees, as subformulas can be shared. This is important as larger formulas are likely to contain multiple copies of the same subformulas. Figure 8.7 shows a tree and DAG representation of the formula $(a \wedge b \leftrightarrow a) \wedge \neg(a \wedge b)$; negation is coded into the edges of the representations using explicit markers.

Much of the advantage of using DAGs is lost if the DAG has to be expanded again in order to create the formulas that are fed to the satisfiability checker. FixIt circumvents these problems by using tautology and satisfiability preserving mappings from a DAG representation back to propositional formulas. The mappings avoid the creation of multiple copies of shared subgraphs by *local definitions* using fresh variables. The formulas resulting from the mapping can be slightly larger than the original formula, but the savings incurred by DAGing can be dramatic in the general case.

Example 22 (Mapping back to a propositional formula) The formula corresponding to the DAG representation in Figure 8.7 is satisfiable if and only if the formula $(i_0 \leftrightarrow a \wedge b) \wedge ((a \leftrightarrow i_0) \wedge \neg i_0)$ is satisfiable, and is a tautology if and only if $(i_0 \leftrightarrow a \wedge b) \rightarrow ((a \leftrightarrow i_0) \wedge \neg i_0)$ is a tautology. In both cases the fresh variable i_0 is used to define the shared subformula $a \wedge b$. □

We call the data structure used to represent formulas in FixIt *Reduced Boolean Circuits* (RBC). It is a DAG structure with negation markers, much like the example in Figure 8.7, that conforms to a set of constraints:

1. All isomorphic subgraphs are shared (there are not more than one copy of any subgraph).
2. No truth values occur in any RBC except the RBCs representing *True* and *False*.
3. Children of internal nodes are distinct.
4. Children of vertices containing symmetric connectives are ordered by some total order on RBCs
5. Edges to children of equivalence nodes are never negated.

Requirements 1, 2 and 3 are examples of simple reductions on boolean circuits. The common denominator for all the enforced reductions is that they are local in the sense that they affect a very limited part of the graph. Requirements 4 and 5 are examples of constraints that aim to make reductions applicable.

The name Reduced Boolean Circuit is motivated by the fact that the underlying graph can be seen as a combinational circuit: edges correspond to connections, operator nodes and markers correspond to primitive gates, and variable nodes correspond to inputs.

8.5.2 QBF Translation

The second problem, translation of quantified boolean formulas, is addressed by FixIt in two parts: special purpose rules translate the formula stepwise while formula reduction and compaction is handled in the RBC representation. The translation rules are simple identities relating quantified and unquantified formulas; the strength of the translation comes from combining these rules with an efficient representation that shares subformulas and remove local redundancies.

As $\forall x.\ \phi(x)$ is equivalent to $\neg(\exists x.\ \neg\phi(x))$, it suffices that the translator can resolve existential quantifiers. The translation applies the following rules iteratively:

1. If ϕ does not contain x, then translate $\exists x.\ \phi$ by removing the quantifier.
2. Push quantifiers inwards using the identities

$$\begin{array}{lll} \exists x.\ \phi(x) \vee \psi(x) & \equiv \exists x.\ \phi(x) \vee \exists x.\ \psi(x) & \\ \exists x.\ \phi(x) \circ \psi & \equiv (\exists x.\ \phi(x)) \circ \psi & \text{if } \circ \in \{\vee, \wedge\} \text{ and } x \notin \mathrm{Vars}(\psi) \end{array}$$

3. If the formula to be translated is *definitional*, in the sense that it matches the pattern $\exists x.\ (x \leftrightarrow \phi) \wedge \psi(x)$ where $x \notin \mathrm{Vars}(\phi)$, then translate it to $\psi(\phi)$.
4. Translate $\exists x.\ \phi(x)$ to $\phi(\mathit{True}) \vee \phi(\mathit{False})$

Rules 1 and 2 reduce the scope of the quantifier. This is important as we want to quantify over as small a scope as possible in order to alleviate the blowup. Rule 3 is useful during backwards fixpoint analysis of deterministic systems. Recall that the quantified boolean formulas that are created during backwards fixpoint iteration have the form $\exists t.\ \psi_n(t) \wedge T(s,t)$. Deterministic systems have a transition relation T that can be put into the form $\bigwedge_{k=0}^{n} t_k \leftrightarrow \nu_k(s)$, where each conjunct defines a next state variable t_k in terms of the state variables s. The inlining rule is consequently always applicable.

The order of presentation of the rules indicates their relative efficiency. The rule that allows removal of quantifiers without variable occurrences within scope should of course always be attempted before any of the others. Also, it always makes sense to push the quantifiers as far as possible into the formula to avoid blowup before applying the inlining or expansion rule. The inlining rule is effective as it in one sweep removes all occurrences of the quantified variable, but it is vital to the rule that the RBC representation share common subformulas as the inlined subformula may otherwise be replicated many times.

Example 23 (Some quantified boolean formulas and their translations)

1. $\exists x.\ a \wedge b \rightarrow c$
2. $\exists x.\ x \wedge y \wedge z \wedge u \vee v$
3. $\exists x.\ (x \leftrightarrow a \wedge b) \wedge (x \wedge c \rightarrow d)$

The first formula does not contain any occurrences of the quantified variable, so the quantifier can be removed to yield $a \wedge b \rightarrow c$. The second formula contains occurrences of the quantified variable, but the quantifier can be pushed in to rewrite the formula to $(\exists x.\ x) \wedge u \wedge v \wedge y \vee z$. Naive expansion translates the quantified subformula to $\mathit{True} \vee \mathit{False}$ which is removed by the RBC representation. The last formula matches the definitional form and is therefore translated to $a \wedge b \wedge c \rightarrow d$. □

FixIt's translator works directly on the RBC level, and applies the translation rules eagerly in order of efficiency. Naive expansion of formulas is only applied if no alternatives can be found; however, the blowup can sometimes be controlled by the RBC representation even when naive expansion is necessary. Furthermore, the fact that formulas are rewritten and reduced in order to increase sharing makes it useful to cache the results of earlier quantifications. The resulting translation is quite simple, but still powerful enough to be applied in reachability analysis of systems that are challenging to verify using other methods [2].

8.6 Analysis

In this section we analyse the approaches to P-safety checking described in section 8.4. We outline their strengths and weaknesses, relate them to each other, and present some of our experiences of using the methods.

8.6.1 Comparison Between the SAT-based Methods

In the worst case, we should expect all of the verification algorithms to take exponential time. This can be understood by reviewing the complexity theoretic facts. The verification methods that we have presented all rely critically on SAT solving, which is NP-complete; and tautology checking, which is coNP-complete. Regardless of the efficiency of our algorithms, we can therefore not expect that the tools they make use of can solve all problems in polynomial time. Furthermore, FixIt and some of the induction variants also make use of translations between QBF and propositional logic. Satisfiability checking of quantified boolean formulas is PSPACE-complete, so it is also very improbable that we can find a translation between QBF and propositional logic that always avoids an exponential memory consumption.

As we do not commit to specific SAT-solvers and QBF translators, we are also not able to give very much of an average case analysis of the presented verification algorithms, nor make general recommendations. Nevertheless, we can try to relate the methods, and make observations about some characteristics that have impact on their time and space behaviour.

In our analysis, we assume that QBF solving is done by translation to propositional logic formulas. We also do not concern ourselves with the space behaviour of the external SAT solvers as most SAT-solvers require only polynomial space. We regard the following as the most important factors for the time and space behaviour of the SAT-based methods:

1. Number of iterations. (affects time behaviour)
2. Size of formulas that are generated. (affects time and space behaviour)
3. Complexity of the proofs. (affects time behaviour)

Number of Iterations. The number of iterations that the algorithms need to verify a particular system can have a large impact on the verification time, and even make verification infeasible. In fact, in the case of induction, the main motivation for developing more alternatives than the first complete induction variant was to decrease the number of iterations that were needed to verify certain hard systems. The number of necessary iterations of the methods for a given system is also the property of the methods that is easiest to use as a basis of comparison, as it is independent of the SAT-solver and QBF translator used.

Let us define three characteristics of a given system: take the forward diameter $forw$ of a system to be the longest shortest path between an initial state and an arbitrary other state, take the backward diameter *backw* to be the longest shortest path between an arbitrary state and a non-P state, and take the sequential depth *seqdepth* to be $min(forw, backw)$. If the system is *not* P-safe, then define $|mce|$ to be the length of the shortest counterexample.

The necessary number of iterations of the methods is as follows:

1. **Reachability analysis** The reachability analysis method needs exactly $|mce|$ iterations to terminate when the system is not P-safe. When the system is P-safe, the breadth first search guarantees that only as many steps as are taken as is necessary to reach the most distant reachable state are taken; it is easy to see that $forw$ and *backw* are upper bounds for the number of forwards and backwards mode iterations, respectively. So, in a mixed mode analysis that works both forwards and backwards at the same time, at most *seqdepth* iterations are needed. However, $forw$ and *backw* are not tight bounds; there can be a big difference between the length of the longest shortest path between any state and *the closest* initial state of the analysis, and the length of the longest shortest path between any state and any initial state. Therefore *seqdepth* or fewer steps are needed by a mixed mode analysis.

2. **BMC** Just like reachability analysis, bounded model checking needs $|mce|$ iterations to find a counterexample for a P-unsafe system. The number of iterations necessary for verifying a safe system by the original BMC method is equal to the system diameter, which can be substantially larger than the sequential depth. However, the paper that introduced BMC [6] recommended the diameter as a bound for a more general class of properties than P-safety properties. For our special case, the sequential depth is clearly a sufficient bound which can be determined by modifying the diameter formula so that it characterises *backw* and $forw$. Furthermore, the

number of breadth first iterations necessary for a forwards or backwards exploration of the statespace can in principle also be determined using QBF formulas. However, no results have to our knowledge been reported about the feasibility of determining either of these improved bounds on the length of counterexamples using QBF translators and SAT solvers.

3. **Induction** The induction methods terminate when either (1) the base case has a counter model, (2) both the base case and the step become provable, or (3) the termination formula becomes unsatisfiable.

   For a P-unsafe system the base case detects a countermodel after $|mce|$ steps.

   For a P-safe system, the termination is either due to (2) or (3). The number of necessary iterations for one of these conditions to become true depends on the lengths of the considered paths. A P-safe system will always have a provable base case, so the termination due to (2) will take l_1 iterations when l_1 is the longest considered path that ends in a non-P state. For the case of the strongest induction, paths are restricted to be shortest paths, so l_1 is equal to *backw*. Termination caused by condition (3) take place after l_2 iterations when l_2 is the length of the longest considered path from the initial states. In the case of the strongest induction, l_2 is equal to $forw$. The necessary number of overall iterations for verifying a P-safe system by the induction methods is therefore $min(l_1, l_2)$ iterations. For the case of the strongest induction, paths are restricted to be shortest paths, so l_1 is equal to *backw*, and l_2 to $forw$. The necessary number of iterations, $min(l_1, l_2)$, is thus the sequential depth of the transition system.

All the verification methods will thus discover the existence of a counterexample in the same number of iterations. However, the number of iterations that are needed for determining that a system is safe varies. The best variant of induction needs exactly *seqdepth* iterations; bidirectional fixpoint-based reachability analysis takes as many iteration steps as induction, or fewer: and BMC can be as good as reachability analysis, as good as induction, or worse than both of them depending on the termination criterion.

Formula Sizes. That one method can verify a system in fewer iterations than another does not automatically mean it is a better method; if the formulas that are generated are of exponential size, even a single iteration will take too long to finish.

Large formulas can arise for two different reasons: (1) the verification method relies on quantification, and the translation yields an exponential blowup, or (2) the formulas grow to contain too many state variables (regardless of the efficiency of the representation compaction, more variables means larger representation). Reachability analysis is one end of the spectrum as it characterises sets and uses quantification to keep the number of state variables in the system fixed. Induction with different states is on the other end of the spectrum, as no quantification is used, but the number of state variables increases quickly during the execution of the algorithm. Bounded model checking uses quantification in the generation of the diameter formulas, but avoids quantification in the formulas encoding counterexamples.

None of the three methods that we have presented is inherently superior to another in terms of the sizes of the generated formulas. Some systems do not blow up in the quantification; reachability analysis then works very well. Other systems can be very hard to quantify, but respond well to methods that avoid quantification altogether. In practice, we have found that SAT-based bounded model checking can find long error traces during the verification of real microprocessor components [12].

Complexity of Proofs. That one method generates larger formulas than another does not in imply that the resulting proofs take longer time; there is no such general connection. The time needed to do proofs in the different verification methods varies wildly between different systems, and different theorem provers. We have some general experiences, though.

The diameter formulas generated by the BMC procedure are almost always too hard to decide in a reasonable amount of time. For this reason, we believe that bounded model checking is best used for detecting bugs rather than proving correctness. Furthermore, our experience with induction is that proofs needed in induction iterations prior to the last iteration often take up a large part of the verification time, and that once a suitable induction depth has been found, the last iteration where we successfully prove correctness often is quicker than any of the previous, failing, iterations. Iterating up from depth zero is therefore in practice a bad idea; it is better to guess a first estimation of the depth using some heuristic and then continue from there. For reachability analysis, other researchers have observed that a large part of the verification time is often spent in the proofs that attempt to detect fixpoint convergence [99]. One possibility could therefore be to only check for convergence of the computation in some of the iterations.

8.6.2 A Comparison to BDD-based Model Checking

The main reason for why SAT-based verification methods have generated a lot of interest is that they can handle some systems that are very hard to verify using Binary Decision Diagrams. BDDs are used in standard model checkers for the same purposes as we have used formulas: to encode the systems, and the sets and traces that are generated during verification. Thus, SAT based model checking is an interesting complement to BDD based model checking. Why is this?

Firstly, BDDs need exponential space just to represent some circuits; and even worse, some of these effectively unrepresentable circuits actually turn up in practice. As a consequence of this representation problem, BDDs also need exponential space for satisfiability checking of certain formulas. In contrast, many standard satisfiability checkers operate within polynomial space.

An additional disadvantage to the use of BDDs for satisfiability checking is that the user cannot control the search for a satisfying assignment at all. Either the BDD can be built and then satisfiability can be determined in constant time, or it cannot be built and no information can be found. In stark contrast, some modern SAT solvers lets the user design a proof search strategy for each individual proof. Tools built on BDDs are therefore stuck with the pros and cons of BDDs for good and for bad, whereas there are many different SAT solvers available off the shelf, all with different characteristics.

We also believe that the automatic theorem proving approach to model checking better supports a move up in abstraction than decision diagrams. For example, if we would like to verify systems that are described on the level of data types such as lists and natural numbers, we have as of yet not seen an adaptions of decision diagrams that seems to scale up very well in practice. However, the techniques we have presented here can in principle be used for stronger logics than propositional logic. For example, the Swedish company Prover Technology AB has for a long time used a combination of induction and propositional logic with added support for integer arithmetic to handle systems generated from Lustre programs; these systems contain infinite domain arithmetic.

We would like to point out that even if we may sound negative about BDD-based model checking, BDDs in fact often work remarkably well. The value of the SAT based methods we have presented lies in that they can handle some of the systems that BDDs have problems with, not in that they consistently outperform BDDs. They should thus be seen as complements to, rather than replacements for, BDD-based model checking.

As a consequence of how well BDDs often work, there have been attempts to combine BDDs with SAT-based methods [99], aiming to develop procedures that combine the strong points of BDDs and SAT-based model checking. This is an interesting idea as it in principle would allow a model checker to chose the representation that best fitted the situation at hand, and even change representation in mid-analysis. The investigations reported in [99] in effect only uses BDDs as yet another SAT-solver, but the work is still a step towards more fundamental integrations of SAT and BDD-based model checking.

The area of quantification translation is where BDDs have a big advantage over the methods described here. There has not yet been much research into how quantification translation procedures should be constructed; the internal translation used in FixIt, for example, works well for some examples that are intractable for BDDs, but gives a blowup for many others that BDDs handle effortlessly.

8.7 Interesting Future Questions

There are many interesting directions for further research into SAT-based model checking.

There exists a plethora of verification algorithms that use BDDs as their fundamental data structure. The method we used to convert reachability analysis to SAT-solvers in Section 8.4.1 can be used to convert many of these algorithms, and we believe that the range of examples handled by these algorithms can be extended significantly in this way. BDDs also have many other uses than verification. An exciting direction of research would be to investigate the consequences of using SAT-solvers in these applications.

All of the verification methods presented here could benefit immensely from more insight into how formulas can be minimised efficiently. One direction that we believe could be fruitfully explored is circuit synthesis and standard boolean circuit minimisation methods. Also, more research into how QBF could be reduced to propositional logic without generating a space blowup could be very useful for SAT-based model checking in particular, but also in many other fields.

When we discussed generalised induction in Section 8.4.1, we showed that there always exists a largest property that could be used as a strengthening (the set of P-invariant states), and a smallest predicate (the reachable states). Sometimes the construction of these extremal properties through forwards and backwards reachability analysis is infeasible. However, as we remarked, any property that is closed under the successor operation, and that lies between these two predicates can be used as a strengthening. In [10] some first attempts to generate such predicates using a SAT-solver are investigated. However, much remains to be explored in this field.

In this paper, we have concentrated on verifying boolean systems. Many industrially relevant systems are most naturally described at a higher level, for example, in terms of datatypes like natural numbers, lists, and trees. Considering recent work on extending Stålmarck's method both to strengthen the arithmetic and to work for stronger logics, an interesting direction of research would be to further explore how the verification methods we have described can be extended to work in those logics. This opens the possibility to apply the automatic theorem proving directly at the level of QBF, or even at the level of temporal logic. The challenge of work along these lines is to go as far up as possible into the hierarchy of abstraction levels and logics while keeping a method that is automatic for practical systems.

Finally, we firmly believe that combinations of different technologies should be explored. As previously stated, there has already been some research into the combination of SAT solvers and BDDs [99]. This is of course not the only interesting combination. For example, what about on-the-fly model checking of certain components while other parts are handled with decision diagrams and SAT-solvers? A combination of techniques gives more freedom than each of the constituent techniques, and this can be necessary for industrial-scale systems. Of course, challenging issues arise as soon as we adopt this view. With flexibility comes freedom, and how do we control this freedom?

Acknowledgements

We would like to thank Magnus Björk for his careful reading of an earlier draft of this paper.

Bibliography

[1] M. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger. A methodology for large-scale hardware verification. In W. A. Hunt and S. D. Johnson, editors, *Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[2] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *Proc. 12th IEEE Int. Symp. on Logic in Computer Science*, pages 88–98. IEEE Computer Society Press, 1997.

[4] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automation Conf.*, 1999.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[7] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs. In *Proc. 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[8] P. Bjesse. Specification of signal processing programs in a pure functional language and compilation to distributed architectures. Master's thesis, Chalmers Univ. of Technology, Dept. of Computing Science, 1997.

[9] P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-103, Chalmers Univ. of Technology, Dept. of Computing Science, June 1999.

[10] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In W. A. Hunt and S. D. Johnson, editors, *Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[11] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava - Hardware design in Haskell. In *International Conference on Functional Programming*. ACM SigPlan, September 1998.

[12] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. Accepted for publication in Proc. 13th Int. Conf. on Computer Aided Verification, 2001.

[13] P. Bjesse, M. Sheeran, and G. Stålmarck. SAT-based model checking: A tutorial and overview. Submitted for publication, February 2001.

[14] A. Borälv. The industrial success of verification tools based on Stålmarck's method. In *Proc. 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer-Verlag, 1997.

[15] A. Borälv. Case study: Formal verification of a computerized railway interlocking. *Formal Aspects of Computing*, 10(4):338–360, 1998.

[16] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[17] J. Burch and D. Dill. Automatic verification of microprocessor control. In *Proc. 6th Int. Conf. on Computer Aided Verification*, volume 859 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1994.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[19] D. N. Chapman C. B. Jones, C. D. Allen. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory, Hursley, 1972.

[20] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Future-bus+ cache coherence protocol. *Formal Methods in System Design*, 6(2), 1995.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[22] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, December 1999.

[23] A. Cohn. A proof of correctness for the VIPER microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publishers, 1987.

[24] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems.* Marstrand, Sweden, 1998.

[25] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of complex Fourier series. In *Mathematics of Computation, 19*, pages 297–301, 1965.

[26] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[27] D. Cyrluk. Microprocessor verification in PVS. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[28] D. Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In *Proc. 1st Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[29] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *Proc. 2nd Int. Conf. on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science.* Springer-Verlag, September 1994.

[30] J. A. Darringer. The application of program verification techniques to hardware verification. In *Proc. 16th ACM/IEEE Design Automation Conf.* IEEE Computer Society Press, 1979.

[31] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[32] N. Eén. Symbolic reachability analysis based on SAT-solvers. Master's thesis, Dept. of Computer Systems, Uppsala university, 1999.

[33] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Conf. on Design, Automation and Test in Europe*, 1998.

[34] A. Eiriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *Proc. 2nd Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[35] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proc. ICALP '80*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980.

[36] R. W. Floyd. Assigning meanings to programs. In *Proc. of Symposia in Applied Mathematics: Mathematical Aspects of Computer Science*, volume 19, pages 19–31, 1967.

[37] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Lois Nebert, 1879.

[38] R. Gamboa. Mechanically verifying the correctness of the Fast Fourier Transform in ACL2. In 3^{rd} *International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 1998.

[39] K. Gödel. Über formal unentscheidbares sätze der Principia Mathematica und verwandter systeme. *Monatsheft Math. Phys.*, 37:349–360, 1931.

[40] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmayam, editors, *Formal Aspects of VLSI Design.* North-Holland, 1986.

[41] S. G. Govindaraju and D. L. Dill. Approximate symbolic model checking using overlapping projections. In *Electronic Notes in Theoretical Computer Science*, July 1999. Trento, Italy.

[42] J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proc.* 10^{th} *Conf. on Computer Assurance*, Gaithersburg, Maryland, 1995.

[43] A. Gupta, Z. Yang, and P. Ashar. SAT-based image computation with application in reachability analysis for verification. In W. A. Hunt and S. D. Johnson, editors, *Proc.* 3^{rd} *Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[44] J.V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, Sept. 1985.

[45] N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, Namur, Belgium, September 1994. Springer-Verlag.

[46] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[47] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[48] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[49] K. Hanna and N. Daeche. Dependent types and formal synthesis. *Phil. Trans. R. Soc. Lond. A*, (339), 1992.

[50] J. Harrison. The Stålmarck method as a HOL derived rule. In *Proc. 9th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234. Springer-Verlag, 1996.

[51] S. He. *Concurrent VLSI Architectures for DFT Computing and Algorithms for Multi-output Logic Decomposition.* PhD thesis, Lund Institute of Technology, 1995.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[53] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[54] H. Hulgaard, P. F. Williams, and H. R. Andersen. Combinational logic-level verification using boolean expression diagrams. In 3rd *International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1997.

[55] W. A. Hunt Jr. *FM 8501: A verified microprocessor.* PhD thesis, University of Texas at Austin, 1985.

[56] C. N. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

[57] S. D. Johnson. *Synthesis of Digital Designs from Recursive Equations.* MIT Press, Cambridge, Mass., 1984.

[58] C. B. Jones. *Software Development using VDM.* Prentice-Hall International, 1986.

[59] G. Jones. A fast flutter by the Fourier transform. In *Proceedings IVth Banff Workshop on Higher Order.* Springer Workshops in Computing, 1990.

[60] G. Jones and M. Sheeran. The study of butterflies. In *Proceedings IVth Banff Workshop on Higher Order.* Springer Workshops in Computing, 1990.

[61] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[62] Y. Li and M. Leeser. HML: An innovative hardware design language and its translation to VHDL. In *Proc. IFIP Conf. on Computer Hardware Description Languages and their Applications*, 1995.

[63] M. Ljung. Formal modelling and automatic verification of Lustre programs using NP-tools. Master's thesis, Royal Inst. of Technology, Dept. of Teleinformatics, 1999.

[64] L. Lundgren. Stålmarck's method in first order logic. Master's thesis, Chalmers Univ. of Technology, Dept. of Computing Science, 1999.

[65] Z. Manna and the STeP group. STeP: The Stanford temporal prover. Technical report, Computer Science Department, Stanford University, July 1994.

[66] W. W. McCune and L. Wos. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

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

[68] K. L. McMillan. The SMV language. Technical report, Cadence Berkeley Labs, 1999.

[69] G. J. Milne. CIRCAL: A calculus for circuit description. *Integration, the VLSI journal*, 1(2):121–160, 1983.

[70] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[71] J. Misra and K. M. Chandy. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[72] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languagues in Education*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234. Springer Verlag, 1996.

[73] S. Owicki and D. Gries. Verifying properties of parallell programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[74] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[75] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Verlag, 1994.

[76] M. Presburger. Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *1. Kongres matematyk'ow krajow slowia'nskich,*, pages 92–101, 1929.

[77] J. Proakis and D. Manolakis. *Digital Signal Processing.* Macmillan, 1992.

[78] Prover Technology AB. *Prover 4.0 Application Programming Reference Manual*, 2000. PPI-01-ARM-1.

[79] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In 5^{th} *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer-Verlag, 1982.

[80] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 89:25–29, 1953.

[81] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.

[82] R. Sharp and O. Rasmussen. Transformational rewriting with Ruby. In *Proc. IFIP Conf. on Computer Hardware Description Languages and their Applications.* Elsevier Science Publishers B.V., 1993.

[83] M. Sheeran. μFP, an algebraic VLSI design language. PhD thesis, Programming Research Group, Oxford University, 1983.

[84] M. Sheeran. Designing regular array architectures using higher order functions. In *Int. Conf. on Functional Programming Languages and Computer Architecture, (Jouannaud ed.)*, volume 201 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[85] M. Sheeran and A. Borälv. How to prove properties of recursively defined circuits using Stålmarck's method. In *Workshop on Formal Methods for Hardware and Hardware-like systems, Marstrand*, 1998.

[86] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt and S. D. Johnson, editors, *Proc.* 3^{rd} *Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[87] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000.

[88] R. E. Shostak. Formal verification of circuit designs. In *Proc.* 6^{th} *Int. Symp. on Computer Hardware Description Languages and their Applications.* IFIP, North-Holland, 1983.

[89] J. P. M. Silva. *Search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, EECS Department, University of Michigan, May 1995.

[90] S. Singh. *Analysis of Hardware Description Languages.* PhD thesis, Computing Science Dept., Glasgow University, 1991.

[91] J. M. Spivey. *Introducing Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[92] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. *Swedish Patent No. 467076 (1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995)* , 1989.

[93] G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In *Safety of Computer Control Systems*, pages 31–36. Elsevier Science Publishers B.V., 1990.

[94] S. Tahar, Z. Zhou, X. Song, E. Cerny, and M. Langevin. Formal verification of an ATM switch fabric using Multiway Decision Graphs. In *IEEE Proceedings of Sixth Great Lakes Symposium on VLSI*, March 1996.

[95] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[96] M. Velev and R. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Proc. 2^{nd} Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 18–35, Palo Alto, November 1998. Springer-Verlag.

[97] P. Wadler. Monads for functional programming. In *Lecture notes for Marktoberdorf Summer School on Program Design Calculi*, NATO ASI Series F: Computer and systems sciences. Springer-Verlag, August 1992.

[98] T. J. Wagner. Verification of hardware designs through symbolic manipulation. In *Proc. Int. Symp. on Design Automation and Microprocessors.* IEEE, 1977.

[99] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, July 2000. Springer-Verlag.

[100] H. Zhang. SATO: An efficient propositional prover. In *Proc. 14^{th} Int. Conf. on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275. Springer-Verlag, 1997.

[101] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin. Description and verification of RTL designs using Multiway Decision Graphs. In *Proc. IFIP Conf. on Computer Hardware Description Languages and their Applications*, August 1995.