Technical Report

UCAM-CL-TR-601
ISSN 1476-2986

Number 601

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Combining model checking and theorem proving

Hasan Amjad

September 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted March 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

We implement a model checker for the modal mu-calculus as a derived rule in a fully expansive mechanical theorem prover, without causing an unacceptable performance penalty.

We use a restricted form of a higher order logic representation calculus for binary decision diagrams (BDDs) to interface the model checker to a high-performance BDD engine. This is used with a formalised theory of the modal mu-calculus (which we also develop) for model checking in which all steps of the algorithm are justified by fully expansive proof. This provides a fine-grained integration of model checking and theorem proving using a mathematically rigourous interface. The generality of our theories allows us to perform much of the proof offline, in contrast with earlier work. This substantially reduces the inevitable performance penalty of doing model checking by proof.

To demonstrate the feasibility of our approach, optimisations to the model checking algorithm are added. We add naive caching and also perform advanced caching for nested non-alternating fixed-point computations.

Finally, the usefulness of the work is demonstrated. We leverage our theory by proving translations to simpler logics that are in more widespread use. We then implement an executable theory for counterexample-guided abstraction refinement that also uses a SAT solver. We verify properties of a bus architecture in use in industry as well as a pedagogical arithmetic and logic unit. The benchmarks show an acceptable performance penalty, and the results are correct by construction.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank...

Mike Gordon. I could not have wished for a better supervisor than Mike. He suggested the initial idea for the thesis, encouraged my better ideas and steered me away from countless dead-ends. He was always available and always gave excellent advice, deep insights and useful references.

Michael Norrish and Konrad Slind for help with HOL above and beyond the call of duty.

Anthony Fox and Joe Hurd for myriad helpful suggestions about everything under the sun.

Myra van Inwegen and Mick Compton for tea time.

The Computer Laboratory for providing a great working environment and for contributing towards participation in conferences.

Trinity College for awarding me an External Research Studentship which financed my studies and living from October 2000 to September 2003. I would also like to thank the Rouse Ball and Eddington Grant trustees for their financial contribution towards my participation in conferences.

Mama, Baba, and Sajjad, who, even from thousands of miles away, have been a never-ending source of support and encouragement.

Sadia, for love, happiness, and making it all worthwhile.

Chapter 1

Introduction

This dissertation addresses the *verification problem* in computer science. The broad area of research is known as formal verification. Formal verification provides a mathematical justification that a given system of artificial design and construction does what it was designed to do: it exhibits all the desired properties and no undesirable ones.

1.1 Motivating Formal Verification

On 4 June 1996, the Ariane 5 space launcher broke up 40 seconds into its maiden flight. The cost of this failure was estimated at about half a billion US dollars. The inquiry board set up to investigate this disaster traced the fault to a small error in the flight control software [115]. The inquiry board concluded:

> This loss of information was due to specification and design errors in the software of the inertial reference system.
>
> ...
>
> The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure.
>
> ...
>
> This means that critical software – in the sense that failure of the software puts the mission at risk – must be identified at a very detailed level, that exceptional behaviour must be confined, and that a reasonable back-up policy must take software failures into account.

The pervasive presence of computers in our lives means that we rely on the correct functioning of computer software and hardware systems for countless tasks, some of them critical. They are the engines that run modern information economies and mistakes in hardware and software design can have serious economic and commercial repurcussions [96, 187]. Flaws in such systems have caused loss of life [114] and brought humanity to the brink of global nuclear war [15]. The more critical the system, the more stringent and rigorous the design and implementation process should be. However, computer systems

have become so complex that exhaustive manual testing cannot cover even the obvious patterns of behaviour, let alone check for extreme situations.

Formal verification uses mathematical techniques to launch a more powerful attack on the problem. These techniques and the hardware on which they are implemented have now improved to the point where real-world systems can be analysed. NASA, the American space agency, formally verified the control software for deep space probes [84]. The UK Defence Evaluation and Research Agency used formal verification to analyse command and control systems for the Ministry of Defence [193]. Industry giants Intel Corporation and Microsoft Corporation use formal verification for critical sub-systems [11, 83]. Fuelled by these success stories, the perception of formal verification as the ultimate test of correct design and implementation for critical systems is gaining ground. It is now the case that in the Common Criteria,[1] qualifying for the highest security level essentially requires formal verification.

1.2 An Overview of Formal Verification

In theory at least, formal verification can be carried out by a human. However, the proofs involved in formal verification are typically not of the "deep" sort that require human insight, but do involve a mass of technical detail that computers are ideally suited for working with. In practice then, verification assumes the use of mechanised automatic or semi-automatic techniques. One of the earliest references to verification can be traced to a paper by Alan Turing, the Cambridge mathematician who is considered the founder of computer science. He writes (c. 1949) [185]:

> How can one check a large routine in the sense that it's right?
>
> In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

Turing talks about a human rather than a mechanised checker, but, broadly interpreted, this is effectively how formal verification works today. The designer of the system makes assertions about the design, in the sense that the design or implementation of the system satisfies its specification, i.e. the system does what it is supposed to do (called *liveness properties*) and nothing undesirable (called *safety properties*). Automatic or semi-automatic tools are then used to check that the assertions are true for the system. The mathematical underpinnings of the tools ensure – in theory at least[2] – that the truth of individual assertions can be combined into an assertion for the whole system. In other words, they ensure *compositionality*. Work on these ideas has continued since the 1960s [62, 87, 124, 141].

It follows from a celebrated result of Turing [184] that the verification problem in general is unsolvable. It is beyond the scope of this account to discuss his result here. Even if the problem were in theory solvable for a given system, there are two common

[1] An industry standard for secure systems [45].

[2] In practice, finding the right assertions, problems with tools and the human element complicate matters [120].

reasons that make a solution impractical: the behaviour of the system or parts of the system may not be amenable to a mathematical treatment, or the system may simply be too complex for verification to be feasible. This still leaves a large class of systems for which a verification attempt can be successful. Such systems include computer programs as well as electronic circuits and network protocols.

The systems one is attempting to verify exist in the physical world. A physical confirmation (as opposed to experimental validation) of mathematical properties is of course impossible. Formal verification techniques instead work with mathematical descriptions (or *models*) of the system. This is made possible by the uniquely logical nature of computer-based systems even at low levels of abstraction. This ensures that the assumptions justifying a mathematical abstraction of the physical process are never so unrealistic that the verification is not useful.

The verification problem can be approached from two opposing extremes. One way is to exhaustively examine all possible states the system can ever be in, and subject each state (or sequences of states) to all possible combinations of internal and external stimuli and check that only the desired properties hold, i.e. the system can never be in an undesirable state. This is the *state-based* approach to formal verification. The *proof-based* approach concentrates instead on using properties of the design to derive a proof, using formalised mathematics, that the system exhibits all and only the required properties.

1.3 Model Checking

Of state-based approaches, *model checking* has been particularly successful. A *state* of the system under consideration is modelled as a snapshot of the system at some point in time, given by the set of the values of the variables of that system at that time. The system is then modelled as a set of states together with a set of *transitions* between states that describe how the system moves from one state to another in response to internal or external stimuli. Model checking tools are then used to verify that desired properties (expressed in some *assertion language*) hold in the system.

In *global temporal* model checking, mathematical assertion languages that can describe the states of a system over time, also known as *temporal logics*, are used to express desired properties of the system. Then algorithmic techniques for state space exploration are applied to discover whether a given property expressed as a sentence of a temporal logic holds true for required states of the system. Well known temporal model checking tools are CADENCE SMV [127] and NUSMV [33, 109].[3]

In *local temporal* model checking, the model checking algorithm is based on a reduction relation over formulae of temporal logics. Certain properties can be verified without having to explore the entire state space. Thus, infinite state spaces can be explored. Work on this includes [17, 179, 191].

In *automata-theoretic* model checking, two models of the system at different levels of abstraction (typically the specification and implementation) are represented as automata and techniques to discover *containment* of one automaton in the other, i.e. that all behaviours of one are encompassed by the other, are applied. Thus we can check whether the design satisfies the specification. The SPIN [90] and COSPAN [106] model checkers fall

[3]There is a plethora of verification tools of any type. We list a well known representative subset.

in this category.

In *refinement-based* model checking we again have two models of the system at different levels of abstraction, but the behaviour is described in terms of sets of *traces* (sequences of input/output ending in termination) and optionally *failures* (sequences ending due to failure) or *divergences* (sequences that do not end) of the system. The tools can once again check containment (called *refinement* in this case) of the sets, with similar applications as before. The CONCURRENCY WORKBENCH [135] and FDR [63, 157] tools use this technique.

In *symbolic trajectory evaluation* (STE) [4], the states of circuits are modelled by assigning each state-carrying variable a value from a four-valued lattice. Temporal relationships between nodes in a circuit can be checked by evaluating trajectories (state sequences) symbolically instead of by brute-force simulation. This is a more specialised state based approach and in fact can be seen as a special case of temporal model checking [192]. The FORTE [1] and VOSS [166] systems are examples of this.

In *bounded* model checking [16], the transition relation is unfolded up to a fixed depth and satisfiability over paths is checked using efficient decision procedures for boolean formulae. The PROVER CL [7] and NUSMV [33] tools use this technology.

Most of these techniques have become powerful enough over time to attack real world problems. An important development that helped this progress is *reduced ordered binary decision diagrams* (ROBDDs or just BDDs) [25]. BDDs often provide an extremely compact yet canonical representation for boolean formulas, and efficient operations for their manipulation. BDD tools include BUDDY [144], BDDLIB [117] and CUDD [22, 176]; model checkers using these are often called *symbolic model checkers* [51, 126, 152].

Another technique that has helped extend the scope of model checking is an efficient decision procedure for boolean satisfiability, known as a *SAT solver* [55, 169, 170]. SAT solvers can decide satisfiability of very large boolean formulas (up to millions of variables and clauses) in reasonable time. Well known SAT solvers include ZCHAFF [136], GRASP [122] and BERKMIN [69].

Since state sets can be represented as boolean formulas, and since most model checking techniques manipulate state sets, BDDs and SAT solvers have enormously boosted their speed and applicability.

Several other more general techniques, which can be applied to most of the types of model checking described above, have improved the efficiency of model checking. These include partial order reduction [100, 188], symmetry reduction [60], assume-guarantee reasoning [65, 116, 134], and abstraction [37, 52, 111, 161], among others. It is beyond the scope of this introduction to discuss them in detail. Some will be discussed within the context of their use in this work.

A state-based approach known as *boolean equivalence checking* uses BDDs and SAT solvers to decide the equality of circuit models. In terms of expressiveness it lies somewhat below STE.

Boolean equivalence checking deserves a special mention, even though it is informally not regarded as model checking. It is the only formal verification techonology that has entered the mainstream electronic design automation (EDA) workflow in industry. There are several tools [47] which integrate industrial hardware specification languages, proprietary or standard assertion languages, equivalence checking and debugging facilities. Model checking has also been used for industrial EDA, but typically in niche applications

where the extra power is worth the effort.

The state-based approach has two important advantages. First, once the correct design of the system and the required properties have been fed in, the verification process is typically fast and fully automatic. Second, in the event of a property not holding true for some states, the verification process is able to produce a counterexample (i.e. an instance of the behaviour of the system that violates the property) which is extremely useful in helping the human designers pinpoint and fix the flaw.

Unfortunately, the number of possible states of real-world systems is typically very large, usually several orders of magnitude greater than the estimated number of atoms in the known universe. This restricts the scope of model-checking techniques. Nonetheless, significant systems have been verified using this method [41].

1.4 Theorem Proving

Theorem proving has come to dominate proof-based approaches to formal verification. Here the system under consideration is modelled as a set of mathematical definitions in some formal mathematical logic. The desired properties of the system are then derived as theorems that follow from these definitions. The method of derivation or proof borrows heavily from standard results in mathematical logic. However, techniques have been developed to automate much of this process by using computers to handle obvious or tedious steps in the proof. Theorem provers can be classified roughly by their underlying logic.

Classical theorem provers are based on some variant of classical higher order logic. This allows for a powerful treatment of functions and functional programs in particular are easy to verify. On the other hand, the high expressiveness of the logic restricts the level of automation. Well known systems of this type include HOL [89], PVS [149] and ISABELLE/HOL [151].

Constructive theorem provers are based on constructive logics. The advantage here is that the derivation of a proof simultaneously provides an executable version of the algorithm being verified, by leveraging the Curry-Howard isomorphism. The disadvantage is that constructive proofs are generally harder than classical ones. The COQ system [91], based on a calculus of constructions [49], and the NUPRL system [46], based on Martin-Löf type theory [123], are examples of constructive provers.

Set-theoretic provers use some axiomatisation of set theory as their basis. This provides for the best expressiveness and allows for proofs of a very foundational nature. However, the proofs are somewhat cumbersome because automated support for doing proofs in set theory is not as mature as that for higher order logic. ISABELLE/ZF [151], Z/EVES [160], PROOFPOWER/Z [10] and LARCH [67] are examples of this type of theorem prover.

First-order logic provers use first-order logic, typically augmented with some support for recursive data-types. These systems are expressive enough for most verification purposes. At the same time they support better automation both in terms of coverage and speed. Tools falling in this category include NQTHM [21], ACL2 [101] and ISABELLE/FOL [151].

Rewriting-systems are specialised theorem provers engineered for high-performance equational reasoning. MAUDE [43] is an example of such a tool.

Interactive theorem provers can also be distinguished by the style of proof that they support. The majority of interactive theorem provers support an *imperative* style of proof in which the user issues low-level commands for performing rewrites or calling decision procedures, and the eventual proof is just a list of these commands. In the declarative style, such as in the MIZAR system [139], proof is done more in a textbook mathematical style. Imperative proofs are easier to work with and favoured for verification. Declarative proofs are far more readable and better suited to formalising mathematics. Some tools support both styles.

As with model checkers, certain generic techniques are used by most, if not all, theorem provers. These include non-complete decision procedures for first-order logic such as model elimination [118] and resolution [156], and for Presburger arithmetic (linear arithmetic with comparators) [48, 153, 171]. Virtually all theorem provers implement some form of rewriting system to support equational reasoning.

There are several (optionally) non-interactive automatic tools that use highly engineered combinations of these decision procedures on fragments of first-order logic. These include OTTER [99, 125], VAMPIRE [155], GANDALF [182], SPASS [190], UCLID [27], SIMPLIFY [56] and CVC [180].

Work has also been done on allowing these techniques to cooperate with each other to attack larger problems [142, 172], as exemplified by the SIMPLIFY, PVS and CVC provers.

The advantage of the proof-based approach is that it can handle very complex systems because it does not have to directly check each and every state and because the logics are typically more expressive. The drawback is that it requires human insight and creativity to complete the proofs, which requires time-consuming manual labour. Another shortcoming is the inability to produce counterexamples in the event of a failed proof, because one does not know whether the required property is not derivable or whether the person conducting the derivation is not ingenious enough. Several successful verifications using theorem provers have been achieved [64, 83, 92, 131, 159].

1.5 A Hybrid Approach

Model checking is automatic; theorem proving is not. Theorem proving can handle complex formalisms; model checking can not. The strengths and weaknesses of model checking and theorem proving are clearly complementary. Over the past decade much research in formal verification has tried to combine the two approaches in synergistic ways, with varying degrees of success. Our focus is on ways of integrating model checkers and theorem provers, rather than on developing techniques that exploit such an integration.

Model checking is fully automated because propositional logic (or checking containment in automata) is decidable. Any formula of a logic (or fragment of a logic) with only finite types can be unfolded into propositional logic, but the succinctness gap between this and propositional logic causes an explosion in the size of the unfolded formula. Often this increase in size is big enough that propositional decision procedures take an infeasible amount of time or space if given such a formula as input. So more often than not a naive unfolding of an expressive logic into propositional logic will not work. Using model checkers as decision procedures for more than a temporal logic has not seen much research because of this reason. An exception to this is weak monadic second order logic. Though the worst-case complexity is high, there are niche cases where the extra succinctness has

proved useful [86, 103].

Instead, researchers have used theorem provers to split up a problem into model checkable pieces the correctness results of which are then recombined in the prover. Or, they have used the decision procedures in theorem provers to help abstract models to a checkable size. Recent research has focused on doing the core proofs in model checkers because most formal verification research is driven by industrial demand, which places the quick finding of bugs above proofs of overall correctness. Indeed, the automation and counterexample capabilities of model checkers are ideally suited for verification-as-debugging. Successful combinations of this kind have been achieved [14, 19, 57, 127, 154, 148].

However, for a powerful enough theorem prover, model checking is just a special case. Ideally, we would like a situation where a model checkable subset of a theorem proving problem can be passed to a model checker directly, and its results manipulated in the theorem prover. This way we could exploit the full power of model checking without sacrificing the expressive power of theorem provers.

The problem lies in achieving a smooth translation from the theorem prover logic to a formalism the model checker can understand and a smooth translation of the results (be it a success or a counterexample) into the logic of the theorem prover. By "smooth" we mean several things:

- The translation should be correct by construction,
- It should be two-way and allow working at multiple levels of abstraction.
- It must be efficient.
- The general framework should be powerful enough to integrate most model checking technologies.

If these criteria are not met, there is no point in using the resulting tool since the available tools already do a better job.

We believe that using a scriptable theorem prover as a programming platform for model checking techniques is one solution to this problem. Theorem provers are certainly powerful enough to be able to express any model checking formalism, and they have become efficient enough that most of the model checking work can be done by proof in the theorem prover without a great loss of efficiency. Thus the criteria of correctness by construction, efficiency, flexibility and expressiveness can be met.

1.6 Our Contribution

In this work we have taken first steps towards justifying the claim made at the end of §1.5. This dissertation uses the approach of constructing a model checking system within a theorem proving environment.

More precisely, we program global symbolic model checking techniques within the HOL theorem proving environment.[4] This way we retain full automation and the ability to produce counterexamples without sacrificing the ability to manipulate the system description

[4]We chose HOL because it is mature, powerful, programmable and well-documented.

in the theorem proving environment. Thus we are able to exploit the full power of model checking in the relatively more expressive environment of the theorem prover.

The HOL theorem prover is architected to minimise the amount of trusted code. Except for a small kernel implementing the inference rules of the deductive system, all other theories and algorithms are correct by construction because all proofs and procedures can be unrolled into applications of the inference rules. This approach – often called *LCF-style* or *fully expansive* – provides HOL developers with a high assurance of *security*, i.e. the development is sound.

However, this is not well-suited for implementing high-performance algorithms such as BDDs and SAT. This is because completely embedding a model checking system within a theorem prover would involve full formalisation of BDD and SAT tools within HOL. This has already been tried and the performance penalty was found to be unacceptable [81, 82] due to the fully-expansive nature of HOL.

We reduce the expected performance penalty to acceptable levels by leaving some core symbolic model checking primitives outside the theorem prover. More precisely, we use a calculus of BDDs [72, 74] to interface HOL to BUDDY, a mature, high-performance BDD engine written in the C programming language. As long as the primitive BDD operations in BUDDY are sound, we do not compromise soundness.

We also use external high-performance SAT solvers but check their results within HOL. Since proof search in this case is far harder than checking the proof, the performance penalty is acceptable.

Our contribution to formalised mathematics is a formal theory of the modal μ-calculus L_μ [104] embedded in higher order logic, up to proofs of the existence of greatest and least fixed-points under the assumption that the underlying state set is finite (it may be infinite before abstraction is applied). Thus we have formally proved a special case of the Knaster-Tarski theorem [183].

The mathematical argument is straightforward, but the formal proof is technically challenging primarily because HOL lacks native support for variable binding in higher order abstract syntax[5] and also because the choice of representation of the model and logical context significantly affects the efficiency with which we can execute the formalised logic. In fact, previous work attempting to do so [5] claimed that the monotonicity lemma required for the fixed-point theorems could not be formalised in a completely general fashion as we have done (see Lemma 2.15): it had to be derived on the fly during each model checking run.

In summary, we have created a new formal theory for an expressive temporal logic and used it to develope concrete technology to demonstrate that using a theorem prover as a tool programming platform provides us with several theoretical advantages without too high a performance penalty. We thus hope that this work will be of interest to the research community and also be of use to industrial practitioners.

[5]Providing automatic α-conversion is an active research area. Work done on this includes [61, 66, 70].

1.7 The Thesis

1.7.1 Prerequisites

We expect the reader to be familiar with the notion of functional programming. We note that the implementation is entirely in the Moscow ML [167] dialect of the SML [133] functional programming language.

A passing familiarity with higher order logic and the HOL theorem prover would be helpful. A brief overview of higher order logic and HOL can be found in Appendix A.

Similarly, some familiarity with symbolic model checking would aid in reading the text. A well-known textbook [36] is a good starting point.

We do not provide more than a superficial description of binary decision diagrams. More than this is not required to understand our work. There are, however, good tutorials [6] and surveys [26] available for the interested reader.

1.7.2 Structure

The thesis is divided into two parts. The first part addresses the foundational aspects of the work. Chapter 2 describes a calculus for interfacing a BDD engine with HOL. It then covers a formalisation of the modal μ-calculus in higher order logic. The chapter continues with the construction of a model checker based on this formalisation. Chapter 3 covers the formalisation of a simple caching algorithm, and an advanced caching technique that exploits nested non-alternating fixed-points in μ-calculus formulas.

The second part explores the practical uses for this work. Chapter 4 describes adding support for CTL, a well known and widely used temporal logic. Chapter 5 leverages the work in Chapter 4 to construct a formalised framework for counterexample-guided abstraction refinement. This demonstrates how the basic model checker integrates seamlessly with the latest research in formal verification. Chapter 6 puts the system through its paces by verifying an implementation of the Advanced Microcontroller Bus Architecture (AMBA) specification from ARM Limited. Chapter 7 concentrates on performance issues using a well-known pedagogical three-stage pipelined arithmetic and logic circuit. The empirical results show that the performance penalty caused by the theorem proving overhead is within acceptable bounds.

Chapter 8 discusses related work and in Chapter 9 we conclude with a summary of the work and directions for the future.

1.8 Terminology and Notation

Thus far we have used several terms quite loosely. We now make precise the meanings of some words that will occur frequently in this work.

Proofs A *proof* is a convincing mathematical justification for a claim. An *informal* proof is one that relies on intuitive arguments and leaves the audience to fill in the details. Such proofs are often known as "hand-waving" proofs in mathematical literature and are most often employed in lectures and conversation. A *non-formal* proof is one that has been achieved using rigorous mathematics but expressed in an informal meta-language such as English combined with mathematical notation to help with conciseness, accuracy

and standardisation. This style of proof is found in mathematical literature such as journals, conference proceedings and books. A *formal* proof of a claim is a derivation tree where the leaves of the tree are axioms or ground rules of some logic and each interior node represents the application of a rule of inference of that logic. The root of the tree is the claim that has been proved. Mechanical theorem provers use this form of proof exclusively. When we use the word "proof" we refer to formal proofs, or sketches thereof, unless explicitly stated otherwise.

Tools The HOL-4 theorem prover [89] has been used to formalise all the arguments and results presented in this work. With the exception of Chapter 8, a reference to "the theorem prover" or HOL is to be construed as a reference to HOL-4.

Terms A *term* is a well-typed formula or sentence in HOL's higher order logic (HOL). We denote syntactic equality of terms using the relation $\equiv$. This may or may not coincide with semantic equality, because of our explicit handling of variable binding in higher order abstract syntax.

Definitions, theorems etc. A *definition*, *theorem* or *lemma* will always refer to one that has been stated or derived formally by us in the theorem prover, unless explicitly stated otherwise. A *proposition* is not formally proved but provided as an aid to understanding the context. These words are initial-capitalised when referring to a specific instance. All such instances are uniquely numbered. The turnstile symbol $\vdash$ often precedes their formal statement.

Fonts We use SMALLCAPS font to refer to software tools. Important words are *italicised* the first time they occur. In addition, the statements of definitions and theorems are italicised. Textual input or output from a computer is indicated by `typewriter` font, as are references to terms and types as implemented in tools, e.g. `bool` refers to the type of booleans as implemented in HOL. References to well known theorems or logics use sans serif font.

Notation We use standard notation wherever possible. Where we introduce our own notation we make this explicit. A full index of such notation is provided at the end. Any notation not in the index should be read in the standard way.

Chapter 2

An embedded model checker

Model checking and theorem proving are two complementary approaches to formal verification. An increasing amount of attention has thus been focused on combining these two approaches. In this chapter we demonstrate an approach to embedding a model checker in the HOL theorem prover. The expectation is that this will ease the combination of state-based and definitional verification workflows.

Model checkers are typically written with an emphasis on performance. Theorem provers typically are not. However, preliminary benchmarking (see Chapter 7) shows that the loss in performance using our approach is within acceptable bounds.

2.1 Introduction

Symbolic model checking (§1.3) is a popular verification technique. A state of a system is represented by the set of the values of the variables of the system at some point in time. Sets of states are then represented by the BDDs of their characteristic functions. This representation is compact and provides an efficient[1] way to test set equality and do image computations. This is useful because evaluating temporal logic formulae almost always requires a fixed point computation that relies on image computations to compute the next approximation to the fixed point and a set equality test to determine termination. Most of the work is done by the underlying BDD engine.

Recent work by Gordon [74] represents primitive BDD operations as inference rules added to the core of the theorem prover. We use this work to model the execution of a model checker for a given property as a formal derivation tree rooted at the required property. These inference rules are hooked to BUDDY, a high-performance BDD engine [144] external to the theorem prover. Thus the loss of performance – due to the theorem proving overhead caused by the LCF-style implementation of HOL – is low.

The security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound. Since we do almost everything within HOL and use only the most primitive BDD operations of a mature tool, we expect a higher assurance of security than from an implementation that is written from scratch in, say, C.

Our aim is not to re-implement a model checking algorithm for which several excellent

[1] The problem is NP-complete. So this efficiency is of heuristic value only.

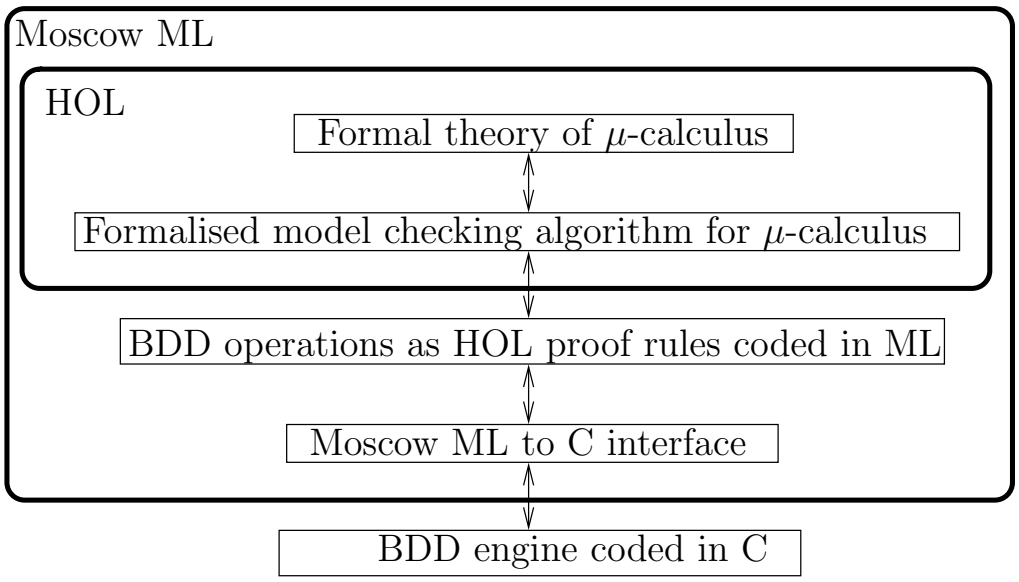

Figure 2.1: High-level Architecture

implementations are already available, but to establish a platform for securely programming new verification algorithms. The fine-grained integration given by our interface makes this possible.

Figure 2.1 gives a bird's eye view of the scheme. In the next section we describe the HOL to BUDDY interface.

2.2 Representing BDDs in a Theorem Prover

In order to provide a platform for programming model checking procedures from within HOL, BUDDY has been interfaced to ML so that BDDs can be manipulated as ML values of type `bdd` [74]. To represent BDD operations as inference rules, we use judgements of ML type `term_bdd` of the form

$$\rho\, t \;\mapsto\; b$$

where t is a HOL term (which we shall call the *term part* of the judgement) and b is a BDD (called the *BDD part* of the judgement). The only free variables of t are the boolean variables used in b (also called the *support* of b). Intuitively, if we collect these boolean variables together in a tuple s, the judgement is saying that for all assignments to the variables in s, an assignment will satisfy t if and only if it is also a satisfying assignment for b. We will often refer to instances of such judgements as *term-BDDs*.

The variable map ρ (of ML type `vm`) maps HOL variables to numbers. The map is required because BUDDY uses numbers to represent variables and for the moment its presence is only a technical requirement. In future work we may use BDD operations like restriction and composition that change the support of a BDD and hence require extending or contracting the associated variable map; thus we let each judgement carry its own variable map rather than having a global map.

Our approach to 'proving' such a judgement is implemented analogously to the man-

Table 2.1: Primitive Operations for Representation Judgements

$$(\mathsf{BddT} : \mathtt{vm} \rightarrow \mathtt{term\_bdd}) \quad \rho\,\mathtt{T} \mapsto \mathtt{TRUE}$$

$$(\mathsf{BddF} : \mathtt{vm} \rightarrow \mathtt{term\_bdd}) \quad \rho\,\mathtt{F} \mapsto \mathtt{FALSE}$$

$$(\mathsf{BddVar} : \mathtt{vm} \rightarrow \mathtt{term} \rightarrow \mathtt{term\_bdd}) \quad \frac{\rho(v) = n}{\rho v \mapsto \mathtt{ithvar}\,n}$$

$$(\mathsf{BddNot} : \mathtt{term\_bdd} \rightarrow \mathtt{term\_bdd}) \quad \frac{\rho t \mapsto b}{\rho\,\neg t \mapsto \mathtt{NOT}\,b}$$

$$(\mathsf{BddAnd} : \mathtt{term\_bdd} * \mathtt{term\_bdd} \rightarrow \mathtt{term\_bdd}) \quad \frac{\rho\,t_1 \mapsto b_1 \quad \rho\,t_2 \mapsto b_2}{\rho\,t_1 \wedge t_2 \mapsto b_1\,\mathtt{AND}\,b_2}$$

$$(\mathsf{BddOr} : \mathtt{term\_bdd} * \mathtt{term\_bdd} \rightarrow \mathtt{term\_bdd}) \quad \frac{\rho\,t_1 \mapsto b_1 \quad \rho\,t_2 \mapsto b_2}{\rho\,t_1 \vee t_2 \mapsto b_1\,\mathtt{OR}\,b_2}$$

$$(\mathsf{BddAppEx} : \mathtt{termlist} \rightarrow \mathtt{term\_bdd} * \mathtt{term\_bdd} \rightarrow \mathtt{term\_bdd}) \quad \frac{\rho(v_1) = n_1 \ \ldots \ \rho(v_p) = n_p \quad \rho\,t_1 \mapsto b_1 \quad \rho\,t_2 \mapsto b_2}{\rho\,\exists v_1 \ldots v_p.t_1\,op\,t_2 \mapsto \mathsf{appex}\,b_1\,b_2\,(n_1, \ldots, n_p)}$$

ner in which we prove theorems, i.e. BDD representation judgements cannot be freely constructed but may be derived using primitive inference steps.

Table 2.1 presents a subset (that is relevant to our work) of the rules that form the primitive operations for BDD judgements, along with the names of ML functions implementing them (in brackets). The BUDDY function $\mathtt{ithvar}\,n$ (as interfaced to ML) simply returns the BDD of the boolean variable v where $\rho(v) = n$. `TRUE` and `FALSE` denote the corresponding BDDs, `T` and `F` are HOL terms for truth and falsity, and `NOT`, `AND` and `OR` denote the eponymous BDD operations [74].

In practice, existential quantification of conjunction (often called the *relational product* or *image computation*) occurs frequently and is an expensive operation. BUDDY provides a special operation `appex` for performing quantification over a boolean operation in one pass, and we have a BDD inference rule BddAppEx corresponding to it.

Theorem proving support is provided by two rules. The first expresses the fact that logically equivalent terms should have the same BDD (up to variable orderings).

$$\frac{\rho\,t_1 \mapsto b \quad \vdash t_1 \Leftrightarrow t_2}{\rho\,t_2 \mapsto b}\mathsf{BddEqMp} \tag{2.1}$$

This rule enables us to use higher-order predicates in the term part of judgements to succinctly express the propositional content of the BDD part of the judgement.

The second rule is the only way to make theorems. It simply checks to see if the BDD

part of the judgement is TRUE and if so, returns the term part as a theorem.

$$\frac{\rho\, t \mapsto \texttt{TRUE}}{\vdash t}\mathsf{BddOracleThm} \tag{2.2}$$

This theorem is only as good as the BDD that was produced: the soundness of it depends on the soundness of the BDD engine and of our representation judgement inference rules. This fact is reflected within HOL by giving the theorem a tag to distinguish it from theorems proved completely within HOL.

Nonetheless, by treating BDD operations as inference applications, we restrict the scope of soundness bugs to single operations which are easy to get right. This is why this approach was chosen in favour of a single powerful rule which, given a term, would return its term-BDD.

2.3 Model Checking

Our general approach is independent of the choice of temporal logic. We shall apply it to the model checking procedure for the propositional μ-calculus L_μ from [104]. L_μ is very expressive and a model checker for it gives us model checkers for the popular temporal logics CTL and LTL.[2]

Formulae of L_μ describe properties of a system that can be represented as a state machine. In particular, the semantics of a formula is the set of states of the system satisfying the formula. The model checking algorithm computes this set given a formula and a system.

We need to make the notion of "system" precise. For our purposes, the system or model is represented by a Kripke structure [36, 105].

Definition 2.1 *A Kripke structure M is a tuple (AP, S, S_0, T, L) where*

- *AP is the finite set of* atomic propositions *of the system.*
- *S is a finite set of states.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *T is the set of actions (or transitions or program letters) such that for any action $a \in T$, $a \subseteq S \times S$.*
- *$L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Each $p \in AP$ is a proposition constructed from the variables v of some finite domain D_v and the constants and operators over that domain. Let $V = \bigcup_{p \in AP} freevars(p)$ and $n = |V|$. Thus the set $S = D_{v_0} \times D_{v_1} \times \ldots D_{v_{n-1}}$. A state $s \in S$ in M is thus a tuple[3]

[2]Though from a practical viewpoint, model checkers for L_μ are not as efficient as say SMV [126] or SPIN [90]. Note also that for logics that do not admit a direct syntactic embedding into L_μ, e.g. LTL, the translation into L_μ is non-trivial [53, 165] and a fully-expansive translation would provide much needed assurance of soundness.

[3]Mathematically of course, we can regard a state as a set rather than as a tuple. The tuple representation smoothes the implementation somewhat.

over V. We write $s \vDash p$ if p is true when the variables of p are assigned the corresponding values from the state s. The value of a variable v in the next state of a transition is denoted by v'. We write $s \vDash_M f$ if the state s of M satisfies the temporal property f constructed over the $p \in AP$ and $M \vDash f$ if f holds in all states of M.

We now present the syntax of L_μ, essentially as given in [36].

Definition 2.2 *Let VAR be the set of* relational variables*, $p \in AP$ be an atomic proposition and $a \in T$ be an action. Then if f and g are L_μ formulas, so are: $True$, $False$, p, $\neg f$, $f \wedge g$, $f \vee g$ (the propositional fragment); $[a]f$ and $\langle a \rangle f$ (the modal fragment); P, $\mu Q.f$ and $\nu Q.f$ (the relational or recursive fragment where $\{P, Q\} \subseteq VAR$ and all occurrences of Q in the negation normal form[4] NNF of f are not negated).*

We shall use f and g, often primed or subscripted, to refer to L_μ formulas. We often use the term "variable" instead of "relational variable" or "propositional variable"; the meaning should be clear from the context. We use $p, p_0, \ldots$ to denote propositional atoms and $Q, Q_0, Q_1, \ldots$ for relational variables. In the formulas $\mu Q.f$ and $\nu Q.f$, μ and ν are considered binders on Q, and thus we have the standard notion of bound and free variables. We use $f(Q_1, Q_2, \ldots)$ to denote that $Q_1, Q_2, \ldots$ occur free in f.

Intuitively, the propositional fragment behaves as expected. In the modal fragment $[a]f$ holds of a state if f holds in all states reachable from that state by doing an a action, and $\langle a \rangle f$ holds of a state if it is possible to make an a action to a state in which f holds. We abbreviate $(s, s') \in a$ by $s \xrightarrow{a} s'$. It is often convenient to use a dot (e.g. $[.]f$) to denote an arbitrary action. In the recursive fragment, $\mu Q.f$ and $\nu Q.f$ represent the least and greatest fix-points of the predicate transformer function on the state-set semantics of f.

The semantics of a formula f is written $[\![f]\!]_M e$, where M is a Kripke structure and the *environment* $e : VAR \to 2^S$ holds the state sets corresponding to the free relational variables of f. By $e[Q \leftarrow W]$ we mean the environment that has $e[Q \leftarrow W]Q = W$ but is the same as e otherwise. We denote the empty environment by $\perp$. We use $\tau : 2^S \to 2^S$ to refer to the predicate transformer on state sets given by $\tau(W) = [\![f]\!]_M e[Q \leftarrow W]$. We now define $[\![f]\!]_M e$.

Definition 2.3 *The semantics of L_μ are defined recursively as follows*

- $[\![True]\!]_M e = S$ *and* $[\![False]\!]_M e = \emptyset$
- $[\![p]\!]_M e = \{s | p \in L(s)\}$
- $[\![Q]\!]_M e = e(Q)$
- $[\![\neg f]\!]_M e = S \backslash [\![f]\!]_M e$
- $[\![f \wedge g]\!]_M e = [\![f]\!]_M e \cap [\![g]\!]_M e$
- $[\![f \vee g]\!]_M e = [\![f]\!]_M e \cup [\![g]\!]_M e$
- $[\![\langle a \rangle f]\!]_M e = \{s | \exists t. s \xrightarrow{a} t \wedge t \in [\![f]\!]_M e\}$

[4]This is a syntactic transformation that pushes all negations inwards to the atoms using the De Morgan style dualities $\neg(f \wedge g) = \neg f \vee \neg g$, $\neg(f \vee g) = \neg f \wedge \neg g$, $\neg[a]f = \langle a \rangle \neg f$, $\neg\langle a \rangle f = [a]\neg f$, $\neg\mu Q.f(Q) = \nu Q.\neg f(\neg Q)$ and $\neg\nu Q.f(Q) = \mu Q.\neg f(\neg Q)$, and the equation $\neg\neg f = f$.

- $[\![[a]f]\!]_M e = \{s | \forall t. s \xrightarrow{a} t \Rightarrow t \in [\![f]\!]_M e\}$
- $[\![\mu Q.f]\!]_M e$ *is the least fix-point of* τ
- $[\![\nu Q.f]\!]_M e$ *is the greatest fix-point of* τ

Note that it is never the case that a relational variable Q is not in the domain of the enviroment, because the environment contains mappings for all free variables (as stated above), and bound variables are added to the environment whenever the semantics of the subformulas within the variables' scope are being evaluated.

Environments can be given a partial ordering $\subseteq$ under component-wise subset inclusion. Now the semantics evaluate monotonically over environments [104],

Proposition 2.4 *For any Kripke structure M, environments e and e', relational variable Q and well-formed L_μ formula f, and $W \subseteq S$ and $W' \subseteq S$, we have*

$$e \subseteq e' \wedge W \subseteq W' \Rightarrow [\![f(Q)]\!]_M e[Q \leftarrow W] \subseteq [\![f(Q)]\!]_M e'[Q \leftarrow W']$$

so by Tarski's fix-point theorem in [183], the existence of fix-points is guaranteed. In fact, since S is finite, monotonicity implies continuity [183], which gives

Proposition 2.5

$$[\![\mu Q.f]\!]_M e = \bigcup_i \tau^i(\emptyset) \text{ and } [\![\nu Q.f]\!]_M e = \bigcap_i \tau^i(S)$$

where $\tau^i(Q)$ is defined by $\tau^0(Q) = Q$ and $\tau^{i+1} = \tau(\tau^i(Q))$. So we can compute the fix-points by repeatedly applying τ to the result of the previous iteration, starting with $[\![False]\!]_M e$ for least fix-points and $[\![True]\!]_M e$ for greatest fix-points. Since S is finite, the computation stops at some $k \leq |S|$, so that the least fix-point is given by $\tau^k([\![False]\!]_M e)$ and the greatest fix-point by $\tau^k([\![True]\!]_M e)$. We then have that

Proposition 2.6 *If $\tau^i(Q) = \tau^{i+1}(Q)$ then $k = i$.*

Essentially, the semantics describe the model checking algorithm itself. An executable version of Proposition 2.6 would rely on being able to efficiently test state sets for equality. Since states are boolean tuples, we can represent state sets by the BDDs of their characteristic functions. Since the semantics are constructed using set operations, every step of the algorithm can be represented by an operation on BDDs. Hence every step can be represented by the application of a BDD representation judgement inference rule.

From Table 2.1, we can give a more concrete semantics for μ-formulae, this time using representation judgements.

Definition 2.7 *The L_μ model checking procedure $\mathcal{T}[\![-]\!]^\rho_M e$ is defined recursively over the structure of μ-formulae as follows*

- $\mathcal{T}[\![True]\!]^\rho_M e = \mathsf{BddT}\,\rho$ *and* $\mathcal{T}[\![False]\!]^\rho_M e = \mathsf{BddF}\,\rho$
- $\mathcal{T}[\![p]\!]^\rho_M e = \mathsf{BddVar}(\rho\, p)$
- $\mathcal{T}[\![\neg f]\!]^\rho_M e = \mathsf{BddNot}(\mathcal{T}[\![f]\!]^\rho_M e)$

- $\mathcal{T}[\![f \wedge g]\!]^{\rho}_{M} e = \mathsf{BddAnd}(\mathcal{T}[\![f]\!]^{\rho}_{M} e, \mathcal{T}[\![g]\!]^{\rho}_{M} e)$
- $\mathcal{T}[\![f \vee g]\!]^{\rho}_{M} e = \mathsf{BddOr}(\mathcal{T}[\![f]\!]^{\rho}_{M} e, \mathcal{T}[\![g]\!]^{\rho}_{M} e)$
- $\mathcal{T}[\![\langle a \rangle f]\!]^{\rho}_{M} e = \mathsf{BddAppEx}(\wedge, [\![T(a)]\!], \mathcal{T}[\![f]\!]^{\rho}_{M} e)$
  where $[\![T(a)]\!]$ *is the term-BDD for the action* a
- $\mathcal{T}[\![[a] f]\!]^{\rho}_{M} e = \mathcal{T}[\![\neg\langle a \rangle \neg f]\!]^{\rho}_{M} e$
- $\mathcal{T}[\![\mu Q.f]\!]^{\rho}_{M} e = \bigcup_{i=0}^{k} \tau^{i}(\emptyset)$
  where $\tau = \lambda W.\mathcal{T}[\![f]\!]^{\rho}_{M} e[Q \leftarrow W]$ *and* $\tau^{k}(\emptyset) = \tau^{k+1}(\emptyset)$
- $\mathcal{T}[\![\nu Q.f]\!]^{\rho}_{M} e = \bigcap_{i=0}^{k} \tau^{i}(S)$
  where $\tau = \lambda W.\mathcal{T}[\![f]\!]^{\rho}_{M} e[Q \leftarrow W]$ *and* $\tau^{k}(S) = \tau^{k+1}(S)$

Executing the procedure in Definition 2.7 for some L_μ formula f with respect to a Kripke structure M and environment e will yield a judgement $\rho\ f' \mapsto b$ where ρ is the variable map, f' is the *boolean* semantic equivalent of f and b is the BDD of f'. So f' is a propositional logic formula that is likely to be large, unreadable and unsuitable for further manipulation by the theorem prover. We give a simple example to illustrate this problem.

Example 2.8 Suppose f is the simple L_μ formula

$$\mu Q.\, \langle . \rangle Q$$

which gives all states that start an infinite sequence of transitions in the model. The corresponding f' would look something like

$$\exists s_0.s \stackrel{\cdot}{\rightarrow} s_0 \wedge \exists s_1.s_0 \stackrel{\cdot}{\rightarrow} s_1 \wedge \ldots \wedge \exists s_n s_{n-1} \stackrel{\cdot}{\rightarrow} s_n$$

where $\stackrel{\cdot}{\rightarrow}$ abbreviates the entire transition relation and n is bounded above by $|S|$. For any interesting model, $|S|$ would at least be in the thousands and often in the googols[5]. □

This staggering blow-up in the size of the term part of the term-BDD would severely handicap the embedded model checker. Instead, we would like f' to be some term expressing satisfiability of f in M and e, i.e. we would like to obtain a judgement

$$\rho\ (s \models^{e}_{M} f) \mapsto b\,,$$

where the state s is a tuple of free boolean variables corresponding to the support of b. Deriving a judgement in the form above is what we shall now attempt. To do this, we must provide a definition of the semantics of L_μ to the theorem prover.

2.4 Model Checking Formalised

This section presents a mechanical formalisation of the theory and algorithms described above. To save space, the lengthy and theorem prover specific formal proofs are not given. Proof sketches are provided where they aid intuition.

[5] One googol $= 1.0 \times 10^{100}$

2.4.1 Formalising the Theory

The formalisation goes along the lines of §2.3. The atoms corresponding to the $p \in AP$ have type β. We use α to denote the type of a state. During a model checking run α would be specialised to $(\beta \times \beta \times \ldots \times \beta)$ where the size of the product would be $|AP|$. Currently the only type of D_v we support is the boolean lattice $\mathbb{B}$, so β is always specialised to the HOL boolean type `bool`. Kripke structures are represented by a simple record type KS, with fields $AP : \texttt{string}\,\texttt{set}$, $S : \alpha\,\texttt{set}$, $S0 : \alpha\,\texttt{set}$, $T : \texttt{string} \to (\alpha \times \alpha) \to \texttt{bool}$ and $L : \alpha \to \texttt{string} \to \texttt{bool}$ representing components so named in Definition 2.1. Environments have type $\texttt{string} \to \alpha \to \texttt{bool}$.

Note that atomic proposition and action names are modelled as strings. In the former case, this will need to change once we extend D_v support to other domains such as $\mathbb{N}$ and $\mathbb{Z}$, since arithmetic propositions are better represented using HOL's native term representation. The type of environments indicates that relational variables are represented by strings in the syntax. This is because HOL does not support automatic α-conversion for higher-order abstract syntax.

At the time of writing HOL did not support predicate subtypes, so a well-formedness predicate on Kripke structures had to be defined separately.

Definition 2.9 *A Kripke structure* M *is well-formed if* $S = \mathcal{U} : (\alpha\,set)$ *where* $\mathcal{U}$ *is the universal set of all things of type* α*.*

The identification of S with all states is not a strict requirement. It is a technical convenience[6] and does not result in loss of generality in the current context.

Formulas of L_μ are represented by a simple recursive data-type. The syntactic constraint on bound variables (see Definition 2.2) is enforced by a well-formedness predicate on μ-formulas. To define this predicate, we first need to formalise our notion of negation normal form and sub-formulas.

The sub-formula relation $\sqsubseteq$ is defined as expected. Since HOL has no native support for variable binding in higher order abstract syntax, the definition of negation normal form has to explicitly avoid free variable capture.

Definition 2.10 Negated relational variable substitution *of a variable* Q *in an* L_μ *formula* f *is written* $f[\neg Q/Q]$ *to denote the negation of all free occurrences of* Q *in* f *and is defined recursively over the structure of* L_μ *formulas (parameterised by* Q*):*

$$\begin{aligned} True[\neg Q/Q] &= True \\ False[\neg Q/Q] &= False \end{aligned}$$

[6] It makes the oft-used technical lemma $\vdash \forall f M e.[\![f]\!]_M e \subseteq S$ easy to prove offline. The inefficient alternative is to compute the set of reachable states of the model at runtime and then prove all theorems dependent on the lemma at runtime as well. As we shall see in Chapters 6 and 7, a large class of properties can be checked without carrying out the expensive computation of reachable states beforehand and thus we would like to avoid it whenever possible.

$$
\begin{array}{rcl}
(f \wedge g)[\neg Q/Q] & = & f[\neg Q/Q] \wedge g[\neg Q/Q] \\
(f \vee g)[\neg Q/Q] & = & f[\neg Q/Q] \vee g[\neg Q/Q] \\
(\neg f)[\neg Q/Q] & = & \neg f[\neg Q/Q] \\
p[\neg Q/Q] & = & p \\
Q'[\neg Q/Q] & = & \text{ if } (Q \equiv Q') \text{ then } (\neg Q) \text{ else } Q' \\
(\langle a \rangle f)[\neg Q/Q] & = & \langle a \rangle f[\neg Q/Q] \\
([a]f)f[\neg Q/Q] & = & [a]f[\neg Q/Q] \\
(\mu Q'.f)[\neg Q/Q] & = & \text{ if } (Q \equiv Q') \text{ then } \mu Q'.f \text{ else } \mu Q'.f[\neg Q/Q] \\
(\nu Q'.f)[\neg Q/Q] & = & \text{ if } (Q \equiv Q') \text{ then } \mu Q'.f \text{ else } \nu Q'.f[\neg Q/Q]
\end{array}
$$

Negated relational variable substitution can be thought of as a restricted form of substitution. Henceforth, the notation $f(\neg Q)$ abbreviates $f[\neg Q/Q]$.

Definition 2.11 *The* negation normal form *of an L_μ formula f, NNF f is defined recursively over the structure of L_μ formulas:*

- *If f is a not a top-level negation, the NNF predicate is applied recursively to each sub-formula of f.*
- *Otherwise NNF f is defined as follows:*

$$
\begin{array}{rcl}
NNF\, \neg True & = & False \\
NNF\, \neg False & = & True \\
NNF\, \neg(f \wedge g) & = & NNF\, \neg f \vee NNF\, \neg g \\
NNF\, \neg(f \vee g) & = & NNF\, \neg f \wedge NNF\, \neg g \\
NNF\, \neg p & = & \neg p \\
NNF\, \neg Q & = & \neg Q \\
NNF\, \neg(\langle a \rangle f) & = & [a] NNF\, \neg f \\
NNF\, \neg([a]f) & = & \langle a \rangle NNF\, \neg f \\
NNF\, \neg\neg f & = & NNF\, f \\
NNF\, \neg(\mu Q.f) & = & \nu Q.NNF\, (\neg f)[\neg Q/Q] \\
NNF\, \neg(\nu Q.f) & = & \mu Q.NNF\, (\neg f)[\neg Q/Q]
\end{array}
$$

We are now in a position to define well-formedness.

Definition 2.12 *The L_μ formulas $True$, $False$, p and Q are well-formed. Otherwise, a well-typed L_μ formula f is well-formed if and only if all sub-formulas of f are well-formed. However, if $f \equiv \mu Q.g$ or $f \equiv \nu Q.g$ then we additionally require that $\neg Q \not\sqsubseteq NNF\, g$.*

We shall assume that all formulas are well-formed and elide the well-formedness condition from theorem statements to avoid clutter.

The heart of the formalisation is the formal semantics, which follows Definition 2.3.

Definition 2.13 *The formal semantics* $\mathcal{FS}[\![-]\!]^e_M$ *of* L_μ *for a Kripke structure* M *and environment* e *are defined by the mutual recursion*

$$\begin{aligned}
\mathcal{FS}[\![True]\!]^e_M &= S \wedge \\
\mathcal{FS}[\![False]\!]^e_M &= \emptyset \wedge \\
\mathcal{FS}[\![p]\!]^e_M &= \{s | s \in S \wedge p \in AP \wedge p \in Ls\} \wedge \\
\mathcal{FS}[\![Q]\!]^e_M &= \{s | s \in S \wedge e\,Q\,s\} \wedge \\
\mathcal{FS}[\![\neg f]\!]^e_M &= S \backslash \mathcal{FS}[\![f]\!]^e_M \wedge \\
\mathcal{FS}[\![f \vee g]\!]^e_M &= \mathcal{FS}[\![f]\!]^e_M \cup \mathcal{FS}[\![g]\!]^e_M \wedge \\
\mathcal{FS}[\![f \wedge g]\!]^e_M &= \mathcal{FS}[\![f]\!]^e_M \cap \mathcal{FS}[\![g]\!]^e_M \wedge \\
\mathcal{FS}[\![\langle a \rangle f]\!]^e_M &= \{s | \exists q.q \in S \wedge s \xrightarrow{a} q \wedge q \in \mathcal{FS}[\![f]\!]^e_M\} \wedge \\
\mathcal{FS}[\![[a] f]\!]^e_M &= \{s | \forall q.q \in S \wedge s \xrightarrow{a} q \Rightarrow q \in \mathcal{FS}[\![f]\!]^e_M\} \wedge \\
\mathcal{FS}[\![\nu Q.f]\!]^e_M &= \{s | \forall n.s \in FP\ f\,Q\,M\,e[Q \leftarrow S]\,n\} \wedge \\
\mathcal{FS}[\![\mu Q.f]\!]^e_M &= \{s | \exists n.s \in FP\ f\,Q\,M\,e[Q \leftarrow \emptyset]\,n\} \wedge \\
FP\ f\,Q\,M\,e\,0 &= e\,Q \wedge \\
FP\ f\,Q\,M\,e\,(n+1) &= \mathcal{FS}[\![f]\!]^{e[Q \leftarrow FP\ f\,Q\,M\,e\,n]}_M
\end{aligned}$$

The form of the definition for the fix-point operators is equivalent to that in Proposition 2.5, but is more convenient for theorem proving purposes. *FP* here stands for *fixed-point iterator* and performs the function of τ in Definition 2.3 and in fact in notation, we will often use τ^n to stand in for $FP\ f\,Q\,M\,e\,n$ to avoid clutter.

Since we need to test semantics for boolean satisfiability (Proposition 2.6 requires this), we need to define satisfaction of a formula in a state.

Definition 2.14 *A* μ*-calculus formula* f *is satisfied by a state* s *of a Kripke structure* M *under an environment* e *if and only if* $s \in \mathcal{FS}[\![f]\!]^e_M$*. We denote this by*

$$s \models^e_M f$$

For efficiency, theorems expressing satisfiability of all L_μ operators (see Table 2.2) are proved in terms of $\models$ using Definition 2.13, assuming the Kripke structure is well-formed.[7] This is trivial due to the simple connection between $\models$ and $\mathcal{FS}[\![-]\!]$.

For fixed point computations, we require theorems that tell us when a fixed point has been reached. The first step is to prove monotonicity of the semantics of a formula with respect to the free variables, i.e. the environment. We show the results for the least fixed point operator only (greatest fixed points follow by duality).

[7] The missing modal and fix-point operator theorems follow by duality. Fix point computations require several other satisfiability theorems discussed later.

Table 2.2: Satisfiability theorems for model checking based on Defn 2.13

$$\forall\, s\, M\, e. \qquad s \models^e_M True \Leftrightarrow \mathtt{T} \tag{2.3}$$
$$\forall\, s\, M\, e. \qquad s \models^e_M False \Leftrightarrow \mathtt{F} \tag{2.4}$$
$$\forall\, s\, M\, e. \qquad s \models^e_M p \Leftrightarrow p \in L\, s \tag{2.5}$$
$$\forall\, s\, M\, e\, f. \qquad s \models^e_M \neg f \Leftrightarrow \neg(s \models^e_M f) \tag{2.6}$$
$$\forall\, s\, M\, e\, f\, g. \qquad s \models^e_M f \wedge g \Leftrightarrow s \models^e_M f \wedge s \models^e_M g \tag{2.7}$$
$$\forall\, s\, M\, e\, f\, g. \qquad s \models^e_M f \vee g \Leftrightarrow s \models^e_M f \vee s \models^e_M g \tag{2.8}$$
$$\forall\, s\, M\, e\, Q. \qquad s \models^e_M Q \Leftrightarrow e\, Q\, s \tag{2.9}$$
$$\forall\, s\, M\, e\, a\, f. \qquad s \models^e_M \langle a \rangle f \Leftrightarrow \exists q.(T\, a)(s,q) \wedge q \models^e_M f \tag{2.10}$$
$$\forall\, f\, M\, e\, Q\, s\, X\, n. \quad s \models^{e[Q \leftarrow FP\ f\, Q\, M\, e[Q \leftarrow X]n]}_M f$$
$$\Leftrightarrow s \in FP\ f\, Q\, M\, e[Q \leftarrow X]\,(n+1) \tag{2.11}$$

Lemma 2.15 Monotonicity *For a well-formed Kripke structure* M *and well-formed* μ*-calculus formula* $\mu Q.f$*, we have that*

$$\begin{array}{lll} \forall\, e\, e'\, X\, Y. & & \\ & X \subseteq Y & \\ \wedge & \forall Q'.\ \text{if } (\neg Q' \sqsubseteq NNF\ f) \text{ then } e\, Q' = e'\, Q' \text{ else } e\, Q' \subseteq e'\, Q' \\ \Rightarrow & \mathcal{FS}[\![f]\!]^{e[Q \leftarrow X]}_M \subseteq \mathcal{FS}[\![f]\!]^{e'[Q \leftarrow Y]}_M \end{array}$$

Proof By the remarks preceding Proposition 2.5. Monotonicity can be shown by the observations that

1. Each of the operators except negation is monotonic.

2. Only relational variables occur negated in the *NNF* of a formula.

3. Bound variables occur non-negated in the *NNF* of a formula.

4. The negation normal form of f has the same semantics as f.

So we can effectively remove all negations from a formula without affecting the semantics. Monotonicity follows immediately. The second hypothesis in the antecedent enforces the point-wise subset ordering on environments and also ensures that free-variables do not change values during a fixed-point computation. In the case of $e = e'$, this antecedent is trivially true. □

Using monotonicity, we are able to formally derive the equivalent of Proposition 2.6.

Theorem 2.16 *For a well-formed Kripke structure* M *and well-formed* μ*-calculus formula* $\mu Q.f$*, we have that*

$$\begin{array}{ll} & \forall\, e\, n. \\ & FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]n = FP\ f\, Q\, M\, e[Q \leftarrow \emptyset](n+1) \\ \Rightarrow & \mathcal{FS}[\![\mu Q.f]\!]^e_M = FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, n \end{array}$$

Proof It follows from Lemma 2.15 and [183] that $FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, n$ is the least upper bound (under subset inclusion) of all $FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, m$ for $m \leq n$, and that $FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, n = FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, m$ for $m \geq n$. Then

$$\begin{array}{ll} & FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, n \\ = & \bigcup\limits_{i \in \mathbb{N}} FP\ f\, Q\, M\, e[Q \leftarrow \emptyset]\, i \\ = & \mathcal{FS}[\![\mu Q.f]\!]^e_M \end{array}$$

using Definition 2.13. □

We stress once more that all definitions and proofs have been formalised in the theorem prover.

2.4.2 Formalising the Model Checker

We wish to be able to pass a well-formed Kripke structure M, a well-formed formula f, an environment e and a variable map ρ to the model checking procedure $\mathcal{T}[\![-]\!]^\rho_M e$ which returns a judgement of the form

$$\rho\ (s \models^e_M f) \mapsto \mathtt{b} \tag{2.12}$$

where the state s is a boolean tuple comprising the atomic propositions M is defined over.

Preliminaries

The first step in implementing the model checker is to prove that M and f are well-formed. This is trivial but does require that both be HOL terms. Representing M as a HOL term throughout the model checking would be inefficient as the T component can be quite large. So we simply use the zero arity predicate `M` as an abbreviation for the entire structure. This can easily be expanded back into the definition if required. However, this never happens as all the required theorems (Table 2.2) are cached separately. Additionally, evaluation of $T(a)$ directly from the term representation is $O(|T|)$. So we construct an ML binary search tree (BST) T_m which maps each action $a \in T$ to its transition relation. With T_m we can evaluate $T(a)$ in time $O(\mathtt{log}_2\, |T|)$. Since M does not change during the model checking, we do not require write access to it. Once we have proved well-formedness, the theorems of Table 2.2 are specialised to f, M and s.

The environment occurs in the term part of the result and therefore also needs to be represented as a HOL term. However, environments change during every iteration of a fixed point computation. Thus they cannot be abbreviated as for M above without

creating HOL definitions on the fly, which is messy and slow. Fortunately the term representation of environments is not large so they can be represented directly. The changing environments mean that all the satisfiability theorems we proved in Table 2.2 change with each iteration. The solution is to specialise them with the updated environment for each iteration. Finally, we construct an ML BST e_m which is an efficient version of e, analogous to T_m.

The kernel

The core algorithm is based on Definition 2.7, i.e. by recursion over the L_μ formula. Each step in the recursion consists of a one application of a BDD inference rule from Table 2.1 followed by an application of BddEqMp (from §2.2) together with the appropriate theorem (now specialised) from Table 2.2. Thus each step results in a judgement of the form of 2.12. In the end we use BddOracleThm to derive the final theorem. We illustrate this with a small example:

Example 2.17 Suppose we have a Kripke structure M over $\{p_0\}$ and an environment e that maps Q to the set $[\![\neg p_0]\!]^e_M$. Then $p_0 \vee Q$ should hold in all states. This theorem is derived (with mild notational abuse: the theorems for BddEqMp are derived from Table 2.2 and Defn. 2.3 rather than via the term-BDD inference rules as appears to be the case below) by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\rho(p_0)=0}{\rho\, p_0 \mapsto \texttt{ithvar}\,0}\ \text{BddVar} \quad \vdash p_0 \Leftrightarrow s \models^e_M p_0}{\rho\, s \models p_0 \mapsto \texttt{ithvar}\,0}\ \text{BddEqMp} \qquad \dfrac{\dfrac{\dfrac{\rho(p_0)=0}{\rho\, p_0 \mapsto \texttt{ithvar}\,0}\ \text{BddVar}}{\rho\, \neg p_0 \mapsto [\![\neg p_0]\!]_M e}\ \text{BddNot} \quad \vdash \neg p_0 \Leftrightarrow s \models^e_M Q}{\rho\, s \models^e_M Q \mapsto [\![\neg p_0]\!]_M e}\ \text{BddEqMp}}{\rho\, s \models p_0 \vee s \models^e_M Q \mapsto \texttt{TRUE}}\ \text{BddOr} \quad \vdash s \models^e_M p_0 \vee s \models^e_M Q \Leftrightarrow s \models^e_M p_0 \vee Q}{\rho\, s \models p_0 \vee Q \mapsto \mathit{TRUE}}\ \text{BddEqMp}}{\vdash s \models^e_M p_0 \vee Q}\ \text{BddOracleThm}$$

□

In the case of propositional atoms, relational variables and the modal operators, we do not have one theorem but sets of theorems AP_t, VAR_t and T_t indexed by the name of the atom, relational variable or action respectively; the appropriate theorem is picked out for passing to BddEqMp. The theorems in AP_t, VAR_t and T_t are obtained by specialising the theorems 2.5, 2.9 and 2.10 of Table 2.2. The HOL terms for the maps L and T are constructed so that a "lookup", i.e. a rewrite evaluating the application of an atom or action name to L or T respectively, has the same asymptotic cost as a lookup in a Patricia trie (see Appendix C page 112 for details). All theorem sets are cached in ML BSTs for efficiency. The case for fixed point operators is more involved and is discussed in the next section.

It should be noted that the model checker derives every step from theorems proved in HOL and thus the result is correct by construction (this does not apply to the BDD engine, but so far we have not *used* the BDD part of the representation judgements though of course it is updated by the inference rules). As we shall show, the performance penalty for this proof has been acceptable in regression tests. This is the justification for using representation judgements.

Computing fixed points

We limit our discussion to computing least fixed points and elide predicates (for the Kripke structure and formula) for well-formedness. If in a recursive descent $\mathcal{T}$ encounters a sub-formula $\mu Q.f$, we would like to derive the judgement

$$\rho\ (s \models^{e}_{M} \mu Q.f) \mapsto b$$

where, if we consider the BDD b as a set, we would like $b = \bigcup_{i=0}^{k} \tau^i(\emptyset)$ where $\tau = \lambda W.\mathcal{T}[\![f]\!]^{\rho}_{M} e[Q \leftarrow W]$ and $\tau^k(\emptyset) = \tau^{k+1}(\emptyset)$ (see Definitions 2.3 and 2.7).

Thus, by Lemma 2.16, if we can show that in the $(i+1)^{th}$ iteration

$$FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ i = FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ (i+1) \tag{2.13}$$

we have the required result (using Theorem 2.16 and Definition 2.14).

To start, we require a "bootstrap" theorem.

Theorem 2.18

$$\forall f\ M\ e\ Q\ s.FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ 0\ s = F$$

Proof Immediate from Definition 2.13 and the HOL definition of $\emptyset$ (i.e. $\lambda(x:\alpha).F$). □

Now we update the environment e with the mapping

$$[Q \leftarrow \rho\ (FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ 0) \mapsto \texttt{FALSE}]$$

justified by Theorem 2.18 and the BddF rule from Table 2.1. Intuitively we can say that $\rho\ (FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ 0) \mapsto \texttt{FALSE}$ is the mechanised version of $\tau^0(\emptyset)$.

For the iteration, we require a "substitution" theorem.

Theorem 2.19

$$\forall f\ Q\ M\ e\ n\ s.s \models^{e[Q \leftarrow FP\ f\ Q\ M\ e\ (n+1)]}_{M} Q \Leftrightarrow s \models^{e[Q \leftarrow FP\ f\ Q\ M\ e\ n]}_{M} f$$

Proof Simplification with Definitions 2.13 and 2.14. □

This just says that the semantics of f in the n^{th} iteration are the semantics of Q in the $n+1^{th}$ iteration. Then, for the i^{th} iteration, a single call to the model checker returns the judgement[8]

$$\rho\ (s \models^{e[Q \leftarrow FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ i]}_{M} f[f/Q]) \mapsto b$$

We use this and BddEqMp together with equation 2.11 of Table 2.2 and Theorem 2.19 to derive

$$\rho\ (FP\ f\ Q\ M\ e[Q \leftarrow \emptyset]\ (i+1)) \mapsto b$$

which is the mechanised version of $\tau^{i+1}(\emptyset)$. This is then made the new value of Q in e before calling the model checker again. At each iteration, all satisfiability theorems are recreated by specialising the theorems (or theorem sets) from Table 2.2 with the updated environment e. However, the theorem in VAR_t corresponding to Q has to be proved from scratch each time since the value of Q changes with every iteration.

[8]The substitution notation denotes that syntactic recursion occurs in the term part.

After each call to the model checker, we check if the BDD parts of the two most recent iterations are equal. If so, we are able to derive the condition in equation 2.13 using BddOracleThm and stop. This is the only point where BDDs are actually used. Thus the result is guaranteed to be correct assuming the BDD representation judgement inference rules are sound (modulo the soundness of the BDD engine and HOL itself). We have already commented on the relatively high likelihood of this.

2.5 Related Work

The work closest in spirit to our own is the HOL-VOSS system [166]. VOSS has a lazy functional language fl with BDDs as a built-in data-type. In [98] VOSS was interfaced to HOL and verification using a combination of deduction and symbolic trajectory evaluation (STE) was demonstrated. This system and subsequent developments are discussed in some detail in §8.2.

Global model checking has also been formalised in the ACL2 theorem prover [121]. As with our work, the syntax and semantics of L_μ up to the fixed point proofs are formalised. Since the ACL2 logic is executable, the development stops there and does not extend the formalisation to use BDDs for more efficient execution.

Local model checkers have been implemented in a purely deductive fashion. This is possible because local model checking [44, 179, 191] does not require external oracles like BDD engines for efficiency. Thus it is difficult to directly compare this work with our own global model checker. In [5] a local model checking algorithm is given for L_μ. However monotonicity conditions for assertions are proved at runtime rather than as a general theorem (e.g. Lemma 2.15) that can later be specialised. A deeper treatment for the less expressive logic CTL* can be found in [178]. This work also proves the proof system sound and complete using game-theoretic analysis.

There are earlier examples of combinations of theorem provers and model checkers [57, 107]. Typically, the theorem prover is used to split the proof into various sub-goals which are small enough to be verified by a separate model checker. There is no actual integration so the translation between the languages of the theorem prover and the model checker is done manually.

Improved integrations of theorem provers with global model checkers typically enable the theorem prover to call upon the model checker as a black-box decision procedure given as an atomic proof rule [148, 154]. The prover translates expressions involving values over finite domains into purely propositional expressions that can be represented by BDDs. This allows use of the result as a theorem (as in our framework) but this method does not extend readily to the fully expansive approach. It thus achieves better efficiency at the expense of coarser integration and lower assurance of soundness. In §8.3 we give more detail about this and later work.

Theorem provers have also been used to help with abstraction [39, 77, 161] for model checking. Decision procedures in the theorem prover are used to discharge assumptions added to refine an abstraction that turns out to be too coarse and adds too much non-determinism to the system resulting in spurious counter-examples. Decision procedures for some subsets of first order logic have been used in automatic discovery of abstraction predicates [54] and invariant generation [18]. There is no technical obstacle to implementing these frameworks in our setting.

There are several tools that implement some of the research sketched here [14, 19, 89, 149].

2.6 Concluding Remarks

The implementation presented here does not contain non-trivial pieces of code whose soundness might be suspect: the core model checker is straight-forward. Rather, this approach pays off when we add optimisations (Chapter 3) and enhancements (Chapter 5), and combine deductive and algorithmic verification (Chapter 6).

The implementation itself is about 2000 lines of definitions and theorems, and about 500 lines of executable ML code. The proof of Lemma 2.15 is technically difficult, on account of having to do explicit α-conversion for the higher order binders. The executable part is much easier to code, mainly because no soundness checks of the code are required: if the procedure terminates, the result is correct by construction.

Benchmarking for the core model checker showed a performance penalty of about 30% as compared to the system with all proof machinery turned off. This and other performance issues are discussed in Chapter 7, which conducts a case study focusing on performance.

Chapter 3

Optimisations

We now demonstrate that the theory developed so far is sufficient to integrate optimisations that exploit the semantics of L_μ. In this chapter we formalise two improvements to the simple model checker formalised in Chapter 2. The first is a naive caching scheme for the evaluation of common sub-formulas and the second is a more advanced caching algorithm for partially evaluated nested non-alternating fixed-points.

3.1 Introduction

There are several ways in which the basic model checking algorithm can be made more time and/or space efficient. These can be roughly classed as follows:

1. *Model reduction* techniques attempt to reduce the size of the model to be checked. This can be done by compositional reasoning [65, 116, 134], abstraction [39, 52, 77, 111, 161], partial order reduction [100, 188] and symmetry reduction [60] among others.

2. *Property reduction* techniques target the property being checked instead and use knowledge of the logic the property is expressed in to simplify the property or the computation of its semantics [59, 150].

3. *BDD and SAT* techniques are lower-level techniques that seek to improve the execution of the underlying BDD engine or SAT solver by exploiting the structure of the model and/or the property [28, 129].

4. *Hybrid* techniques combine one or more of the above to guide optimisations.

It is beyond the scope of this work to formalise all these techniques. BDD and SAT level techniques typically do not make for interesting formalisations, and most model reduction and hybrid techniques would take us too far afield. We have therefore chosen to concentrate on property reduction techniques.

Our choice of optimizations is motivated by the fact that they are *deep* optimizations in the sense that they do not treat the model checker as a black box but are tightly coupled with its execution at a low level. This implementation thus demonstrates that our approach admits deep optimizations that are nonetheless underpinned by formal proof.

Our first optimisation exploits the fact that for a given model and environment the semantics of an L_μ formula are the same for identical sub-formulas, modulo free variable valuations. Thus when checking a property represented by L_μ formula f, the BDDs computed for each sub-formula can be cached and reused for other occurrences of that sub-formula. This is a standard optimization implemented in all such tools.

The second optimisation uses a result from partial order theory that a least (greatest) fixed-point computation by iteration will terminate correctly as long as the iteration is begun below (above) the fixed-point [59]. Thus, subject to certain constraints, we need not start the iteration with $\emptyset$ (or S).

This will allow us to build upon our formal theory of L_μ developed in Chapter 2 and show that the development has covered enough ground to enable both simple and advanced optimisations. Both techniques make a dramatic difference in the efficiency of the model checking so this will also enable us to check just how badly the additional theorem proving overhead for the optimisations affects the performance boost. The work in Chapter 7 vindicates our belief that for complex enough models, the performance penalty is within acceptable bounds.

3.2 Naive Caching

It is clear from Definition 2.3 that closed formulas will have the same semantics for a given model. We make this idea precise.

Definition 3.1 *The set of* free variables *of an L_μ formula f, $FV(f)$ is computed recursively over the structure of L_μ formulas as follows:*

$$\begin{aligned} FV(False) &= \emptyset \\ FV(True) &= \emptyset \end{aligned}$$

$$\begin{aligned} FV(p) &= \emptyset \\ FV(Q) &= \{Q\} \\ FV(\neg f) &= FVf \\ FV(f \wedge g) &= FVf \cup FVg \\ FV(f \vee g) &= FVf \cup FVg \\ FV(\langle a \rangle f) &= FVf \\ FV([a]f) &= FVf \\ FV(\mu Q.f) &= FVf \setminus \{Q\} \\ FV(\nu Q.f) &= FVf \setminus \{Q\} \end{aligned}$$

Definition 3.2 *An L_μ formula f is* closed, *if $FVf = \emptyset$.*

Lemma 3.3 *If f and g are closed formulas, then for any model M and environments e and e',*

$$f \equiv g \Rightarrow [\![f]\!]^e_M = [\![g]\!]^{e'}_M$$

Proof By induction on the structure of f. The only falsifying case is when $f \equiv Q$, but this is ruled out because f is closed. □

So identical closed sub-formulas can be cached. Precomputing semantic equivalence theorems for all closed subformulae requires evaluating FV over all such formulae, which is computationally expensive. Since it is relatively painless to decide syntactic equality in HOL, we can instead use the fact that any two syntactically equal formulas have the same semantics if the relevant parts of the environment have not changed since the first formula was evaluated.

Lemma 3.4 *For any model M and environments e and e',*

$$\forall f g.(\forall Q.\, Q \sqsubseteq f \Rightarrow e(Q) = e'(Q)) \wedge f \equiv g \Rightarrow [\![f]\!]^e_M = [\![g]\!]^{e'}_M$$

Proof Similar to Lemma 3.3 except the falsifying case is no longer so because of the additional assumption. □

3.2.1 Implementation Issues

The main idea behind the implementation is simple: we cache the term-BDD for some sub-formula f and whenever we come across a formula g such that f and g satisfy the requirements of Lemma 3.4, we use the BddEqMp inference rule together with the Lemma 3.4 to derive the term-BDD for g rather than computing it from scratch.

Lemma 3.4 enables us to use caching proving only syntactic equality and unchanged environments. The former is fast, and for the latter we can pre-prove general theorems showing that except for the $f \equiv Q$ case, changed environments do not affect semantics.

This means we have to prove theorems that the environment is unchanged only for sub-formulas that are relational variables and simply ripple these theorems as assumptions up the super-formulas during model checking.

Example 3.5 Suppose we have a formula g such that its semantics are invariant with respect to the environment, i.e. it is closed. Consider a formula such that its subformula $g \wedge Q$ occurs twice and let us prime g and Q in the second occurrence of $g \wedge Q$ for ease of reference.

Suppose the first occurrence is evaluated first, and the term-BDD for $g \wedge Q$ is cached. In addition, when evaluating g we cache the theorem that

$$\vdash \forall e\, e'.[\![g]\!]^e_M = [\![g]\!]^{e'}_M$$

and for Q we cache the theorem

$$\vdash \forall e\, e'.e(Q) = e'(Q) \Rightarrow [\![Q]\!]^e_M = [\![Q]\!]^{e'}_M$$

Then when evaluating $g \wedge Q$ we derive and cache the theorem

$$\vdash \forall e\, e'.e(Q) = e'(Q) \Rightarrow [\![g \wedge Q]\!]^e_M = [\![g \wedge Q]\!]^{e'}_M$$

using a preproved theorem that gives invariance with respect to environments for conjunction in L_μ. Here the assumption $e(Q) = e'(Q)$ has been rippled up from the invariance theorem for Q to the invariance theorem for $g \wedge Q$.

Finally, when we see $g' \wedge Q'$, we find the corresponding term-BDD for $g \wedge Q$ in the cache. If the value of Q has not changed, we can instantiate e and e', discharge the assumption and use the BddEqMp rule to derive the term-BDD for $g' \wedge Q'$.

We now see the motivation for rippling up the assumption $e(Q) = e'(Q)$: if the environment had changed the valuation of Q since the first occurrence had been evaluated, i.e. $e(Q) \neq e'(Q)$, then the semantics of the first and second occurrence would have been different and blindly picking up the term-BDD for the first occurrence would have caused unsoundness. Reliance on formal proof avoids this scenario by insisting that we discharge the assumption first. □

Fortunately, in practice, the environment usually contains no more than two mappings, because most well known temporal logics can be embedded into L_μ without using free variables and with at most a single nesting of fixed-point operators [59] and L_μ itself is rarely used to write properties due to its non-intuitive semantics. Thus we do not expect, in practice, to have more than two assumptions to ripple around. Thus in practice the proof overhead for caching should be low.

This scheme will not cache α-equivalent subformulas,[1] which also have equivalent semantics modulo free variable valuations. For this we would have to precompute α-equivalence of all qualifying subformulas and then preprove theorems attesting to that fact. This would be impose a large computational overhead. Further, it would require a formal definition of α-equivalence and such definitions are notoriously error-prone.

For the moment, we have a partial solution to this shortcoming. We rename all binders to unique names except that we attempt (non-formally) to ensure that bound names in α-equivalent subformulas are named so as to give syntactic equality. The caching scheme above can then be applied as usual. Note that making a mistake in this non-formal approach does not compromise soundness since the caching scheme we use relies on formal proof. However, it may cause an abnormal termination if it considers non-α-equivalent formulas to be α-equivalent, or lose efficiency if it fails to recognize α-equivalent formulas and hence does not cache them.

3.3 The Alternation Depth Optimization

Unless explicitly stated otherwise, we consider only least fixed points in this section. The corresponding results for greatest fixed points follow by duality.

At the moment, computation of least fixed-points proceeds by initialising the bound variable with $\emptyset$ and iteratively computing approximations until the fixed-point is reached. Due to Lemma 2.15 and Theorem 2.16, the number of these iterations is bound above by $|S|$.

Let us refer to formulas of the form $\mu Q.f$ as σ-formulas.[2] If k fixed-point operators are nested, sub-formulas of the σ-formula will be evaluated up to $O(|S|^k)$ times. In cases where a fixed-point operator is nested immediately within another of the same type, we can exploit monotonicity (Lemma 2.15) to reduce this number. We need the concept of *alternation depth* [36] to continue.

[1] Subformulas that are syntactically identical upto renaming of bound variables.

[2] To avoid confusion between μ-formulas and formulas of L_μ.

Intuitively, the alternation depth of a formula is the number of alternations between nested least and greatest fixed point operators in the negation normal form of that formula.

Definition 3.6 *The* alternation depth *of a formula is defined as follows:*

1. *The alternation depth of constants, atomic propositions and relational variables is 0.*

2. *The alternation depth of propositional and modal formulas is the maximum of the alternation depths of their sub-formulas.*

3. *The alternation depth of a μ-formula is the maximum of: 1; the alternation depth of the body; 1 + the alternation depth of any top level ν-sub-formulas of the body. And similarly for a ν-formula.*

Emerson et al [59] show that we need only evaluate the bodies of σ-sub-formulas $O(|S|^d)$ times, where d is the alternation depth of the top-level formula.

We first need a proof that a fixed-point computation begun below the fixed-point will terminate.

Lemma 3.7 *For any model M where S is finite,*

$$\forall W.\exists k.\tau^k(W) = \tau^{k+1}(W)$$

Proof We can show $\forall n.\tau^n(W) \subseteq \tau^{n+1}(W)$ using monotonicity. Then since S is finite, there exists some k such that

$$\tau^k(W) = \tau^{k+1}(W)$$

□

Corollary 3.8 *For any model M where S is finite,*

$$\forall W.\exists k.\tau^k(W) = \bigcup_i \tau^i(W)$$

Proof By monotonicity,

$$\forall k.\tau^k(W) = \bigcup_{i \leq k} \tau^i(W)$$

By Lemma 3.7 and monotonicity,

$$\exists k.\tau^k(W) = \bigcup_{i \geq k} \tau^i(W)$$

So

$$\begin{aligned} \exists k.\tau^k(W) &= \bigcup_{i \leq k} \tau^i(W) \cup \bigcup_{i \geq k} \tau^i(W) \\ &= \bigcup_i \tau^i(W) \end{aligned}$$

□

We can now show that starting anywhere below the fixed-point is enough.

Lemma 3.9 *For any model M where S is finite,*

$$\forall M\ f.W \subseteq \bigcup_i \tau^i(\emptyset) \Rightarrow \bigcup_i \tau^i(W) = \bigcup_i \tau^i(\emptyset)$$

Proof

- $\subseteq$ direction. By monotonicity, we have for all k,

$$\begin{aligned} \tau^k(W) &\subseteq \tau^k(\bigcup_i \tau^i(\emptyset)) \\ &\subseteq \bigcup_i \tau^i(\emptyset) \quad \text{since } \bigcup_i \tau^i(\emptyset) \text{ is a fixed-point} \end{aligned}$$

  But

$$\exists k.\tau^k(W) = \bigcup_i \tau^i(W)$$

  by Corollary 3.8, so

$$\bigcup_i \tau^i(W) \subseteq \bigcup_i \tau^i(\emptyset)$$

  with the existential quantification on k being eliminated since it becomes vacuous.

- $\supseteq$ direction. By monotonicity, for all k

$$\begin{aligned} & \tau^k(\emptyset) \subseteq \tau^k(W) \\ \Rightarrow\ & \bigcup_i \tau^i(\emptyset) \subseteq \bigcup_i \tau^i(W) \end{aligned}$$

□

Intuitively, because of monotonicity, beginning the iteration anywhere below the least-fixed point is sufficient. Thus when computing the inner fixed-point for the i^{th} iteration of the computation of the outer fixed-point, we can set the initial value of the iteration to the value computed for the inner-fixed point during the outer fixed-point's $(i-1)^{th}$ iteration. And similarly for deeper levels of same-type fixed-point operator nesting.

This means that the number of evaluations required in the computation of any sequence of same-type nested fixed-points is now bounded above by $|S|$, and hence the total number of evaluations is $O(|S|^d)$.

3.3.1 Implementation Issues

Theorem 2.16 is no longer sufficient for our purposes because it assumes that all iterations are initialised with $\emptyset$.

Theorem 3.10 *For a well-formed Kripke structure* M *where* S *is finite, and well-formed* μ*-calculus formula* $\mu Q.f$*, we have that*

$$\begin{array}{lll} \forall\, e\, e'\, n\, k\, W. & & \\ & W \subseteq \bigcup_i \tau^i(\emptyset) & \\ \wedge & \tau^n(W) = \tau^{n+1}(W) & \\ \wedge & \forall Q'.\ \text{if } (\neg Q' \sqsubseteq \mathit{NNF}\ f) \text{ then } e\,Q' = e'\,Q' \text{ else } e\,Q' \subseteq e'\,Q' & \\ \Rightarrow & \mathcal{FS}[\![\mu Q.f]\!]^e_M = \bigcup_i \tau^i(W) & \end{array}$$

Proof We show

$$\mathcal{FS}[\![\mu Q.f]\!]^e_M = \bigcup_i \tau^i(W)$$

using techniques similar to those in the proof for Theorem 2.16. The required result follows by Lemma 3.9. □

Unfortunately, the term $\tau^n(W)$ formally expands out to

$$\mathit{FP}\ f\, Q\, M\, e[Q \leftarrow \mathit{FP}\ f\, Q\, M\, e'[Q \leftarrow \emptyset]\, m]\, n$$

for some m, since W is the final value of the previous computation of $\mu Q.f$ which is nested inside another least fixed-point. Since the value of the outer fixed-point has changed from when the previous computation was carried out, the environment for W is different from the environment for $\tau^n(W)$. So the third assumption

$$\forall Q'.\ \text{if } (\neg Q' \sqsubseteq \mathit{NNF}\ f) \text{ then } e\,Q' = e'\,Q' \text{ else } e\,Q' \subseteq e'\,Q'$$

can no longer be trivially disposed of as was the case earlier (see comments in the proof of Lemma 2.15).

In fact, the third assumption can only be proved at runtime during model checking, because the value of the condition depends on the actual formula being evaluated. Fortunately, this condition in the assumption can be evaluated by rippling up the truth or falsehood of the corresponding conditions for all sub-formulas of f. This mitigates the penalty imposed by the extra runtime proof obligation.

The need to annotate every node in the abstract syntax tree of the formula being evaluated with a specialised version of the third assumption slows down performance. The third assumption asserts just that the values of free variables do not change during evaluation. Currently we detect free variables by finding if a variable is negated in the negation normal form of a well-formed formula. An alternative would be to have a more explicit treatment for free variables, by carrying along a list of free variables occuring in the subformula currently being evaluated. It is not immediately obvious whether this will help or hit performance as there are technical issues involved (see comments after Lemma 2.15), but it is certainly worth investigating in the near future.

3.4 Concluding Remarks

Naive caching is so called because a completely general scheme would be able to equate all terms that have identical semantics whenever possible. In our case, we do not account for

associativity and commutativity in the propositional fragment, De Morgan-style equalities and several other algebraic laws for L_μ. There is no technical barrier to including these; in fact, a theorem prover is ideally suited for this sort of equational simplification. Their implementation has been delayed simply because they do not add any research or pedagogical value to the work.

Our formalisation and implementation of the alternation depth optimisation has similarly stopped short of implementing all possible refinements to the scheme. For instance, formulas with strict alternation of fixed-point operators have been considered and it has been shown that the number of evaluations can be further reduced to $O(|S|^{\lfloor d/2 \rfloor + 1})$ [23].

The well-developed formal theories for sets and natural numbers in HOL saved us considerable time. We have nonetheless demonstrated the power of our formal theory in easily integrating new results about the logic without the need to rework definitions or central lemmas such as Lemma 2.15 and without requiring a lot in the way of additional formal infrastructure.

Chapter 4

Extension I: A temporal logic

Virtually all combinations of model checking and theorem proving rely at some point on translating formalisms (e.g. input languages) in a semantics-preserving manner. In this chapter we show how such a translation would be done in our formalised environment. This translation will be used in chapters 5, 6 and 7.

4.1 Introduction

In Chapter 2 we demonstrated the embedding of a symbolic model checker for L_μ in HOL. This approach allows results returned from the model checker to be treated as theorems in HOL while retaining the advantages of the fully-expansive nature of HOL, without an unacceptable performance penalty.

One use of this approach would be to do property checking using logics that can be embedded into L_μ. Since L_μ is very expressive, most popular temporal logics can be embedded in it without loss of efficiency [36, 59]. A theorem returned by the model checker for an L_μ property can be translated into the corresponding theorem for whatever logic we are interested in, as long as the translation is correct.

If the translation is done via a semantics-based embedding in HOL, we can use the theorem prover to ensure correctness. Thus we can achieve a tight integration with established technology with little loss of efficiency, without having to write a verification tool for our new logic from scratch and with a high assurance of the correctness of our implementation. This chapter provides proof-of-concept of this approach using as an example the standard embedding of the popular temporal logic CTL [13] into L_μ. All definitions, propositions, lemmas and theorems presented here have been mechanised in HOL.

4.2 CTL

The temporal logic CTL is widely used to specify properties of a system in terms of the finite set AP of atomic propositions relevant to the system. The system is modelled by a Kripke structure as before, but we need a slightly different formal definition because in the case of CTL there are no actions and the transition relation is total, so all paths are infinite.

Definition 4.1 *The tuple* (AP, S, S_0, R, L) *represents a Kripke structure* M *where*

- AP *is the set of relevant atomic propositions.*
- S *is a finite set of states. A state is a vector enumerating the variables of* AP*.*
- $S_0 \subseteq S$ *is the set of initial states.*
- R *is a binary transition relation on states such that* $R(s, s')$ *iff there is transition in* M *from* s *to* s'*.* R *is* total*, i.e.* $\forall s \in S.\exists s' \in S.R(s, s')$*.*
- $L : S \to 2^{AP}$ *labels each state with the set of atomic propositions true in that state.*

CTL allows us to describe properties of the states and paths of a *computation tree*. A computation tree is formed by unwinding (infinitely) the transitions of M, starting with the initial states. Intuitively, a computation tree path starting with the state s is an infinite sequence of states $s_0, s_1, s_2, \ldots$ in M such that $s_0 = s$ and $\forall i.s_i \in S$. For a path π, we use π_i to denote the i^{th} state along the path, and ${}^i\pi$ to denote the prefix of π up to but not including the state π_i, and π^i to denote the suffix of π starting from the state π_i.

Definition 4.2 *A computation path* π *starting at some state* s *in a Kripke structure* M *is well formed, PATH* $M\pi s$*, if and only if* $\pi_0 = s$*,* $\forall n.\pi_n \in S$ *and* $\forall n.R(\pi_n, \pi_{n+1})$*.*

Paths are forced to be infinite by the totality of R. Though not a strict requirement, the totality of R simplifies the theoretical treatment of the semantics of CTL and is a standard assumption.

CTL consists of propositional logic augmented with *path quantifiers* and *temporal operators*. The two path quantifiers are **A** ("for all computation paths") and **E** ("for some computation path"). The five basic temporal operators are:

- **X** (unary, "next state") is satisfied by a path if the second state on the path satisfies the required property.
- **F** (unary, "future state") is satisfied by a path if some state on the path satisfies the required property.
- **G** (unary, "globally") is satisfied by a path if all states along the path satisfy the property.
- **U** (binary, "until") is satisfied by a path if the first property holds along the path until a state where the second property holds. The second property must hold eventually.
- **R** (binary, "release") is satisfied by a path if the second property holds up to and including the first state where the first property holds. However, the first property need not ever hold. Thus this is the logical dual of **U**.

The syntax of CTL is made out of *state formulas* which are true of states, and *path formulas* which are true of paths. A well-formed CTL formula is always a state formula. However, we need path formulas because the temporal operators talk about a state with respect to the path of the computation tree the state is on. Formally, formulas of CTL are constructed as follows:

Definition 4.3 *Let P be the set of atomic boolean propositions. Then CTL is the smallest set of all state formulas such that*

- *True is a state formula.*
- *$p \in P$ is a state formula.*
- *If f and g are state formulas then $\neg f$ and $f \wedge g$ are state formulas.*
- *If f and g are state formulas, then $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$ and $f\mathbf{R}g$ are path formulas.*
- *If f is a path formula, then $\mathbf{A}f$ and $\mathbf{E}f$ are state formulas.*

As only state formulas are allowed, we get ten compound operators, e.g. **EX**, **AX**, **EF**, **AF** and so on. Usually, during model checking P and AP are identified.

Formally, the semantics $[\![f]\!]_M$ of a CTL formula f are defined with respect to the states of the Kripke structure M. They represent the set of states of M that satisfy f. Satisfaction of f by a state s of M is denoted by $s \models_M f$. So $s \models_M f \iff s \in [\![f]\!]_M$. Note that in the case of CTL, no environment is needed for evaluating the semantics.

Definition 4.4 *Given the CTL formulas f and g, atomic proposition p, a Kripke structure M and a state s of M,*

- $s \models_M True$
- $s \models_M p \iff p \in L(s)$
- $s \models_M \neg f \iff \neg(s \vDash_M f)$
- $s \models_M f \wedge g \iff s \models_M f \wedge s \models_M g$
- $s \models_M \mathbf{EX}f \iff \exists\pi.PATH\ M\pi s \wedge \pi_1 \models_M f$
- $s \models_M \mathbf{EG}f \iff \exists\pi.PATH\ M\pi s \wedge \forall j.\pi_j \models_M f$
- $s \models_M \mathbf{E}[f\mathbf{U}g] \iff \exists\pi.PATH\ M\pi s \wedge \exists k.\pi_k \models_M g \wedge \forall j.j < k \Rightarrow \pi_j \models_M f$

This definition is sufficient since the ten compound operators can all be expressed in terms of the three operators **EX**, **EG** and **EU** [36]. We state the following without proof:

Proposition 4.5

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}[True\mathbf{U}f]$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}[f\mathbf{U}g] = \neg\mathbf{E}[\neg g\mathbf{U}(\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}(\neg g)$
- $\mathbf{A}[f\mathbf{R}g] = \neg\mathbf{E}[\neg f\mathbf{U}\neg g]$

- $\mathbf{E}[f\mathbf{R}g] = \neg\mathbf{A}[\neg f\mathbf{U}\neg g]$

The following standard lemmas will be useful later. We present the proofs in some detail as we were unable to find a formal treatment in the literature.

Lemma 4.6 *For any Kripke structure M and CTL formula f,*

$$[\![\mathbf{EG}f]\!]_M = [\![f \wedge \mathbf{EX}(\mathbf{EG}f)]\!]_M$$

Proof

- $\subseteq$ *direction.* For any $s \in S$,

$$\begin{array}{rl} & s \models_M \mathbf{EG}f \\ \iff & \exists\pi.\pi_0 = s \wedge \forall j.\pi_j \models_M f \quad \text{by D4.4} \\ \Rightarrow & s \models_M f \wedge \pi_1 \models_M \mathbf{EG}f \\ \iff & s \models_M f \wedge s \models_M \mathbf{EX}(\mathbf{EG}f) \quad \text{by D4.4} \\ \iff & s \models_M f \wedge \mathbf{EX}(\mathbf{EG}f) \quad \text{by D4.4} \end{array}$$

  and we are done by extensionality.

- $\supseteq$ *direction.* For any $s \in S$,

$$\begin{array}{rl} & s \models_M f \wedge \mathbf{EX}(\mathbf{EG}f) \\ \iff & s \models_M f \wedge \exists\pi.\pi_0 = s \wedge \pi_1 \models_M (\mathbf{EG}f) \quad \text{by D4.4} \\ \Rightarrow & \exists\pi.\pi_0 = s \wedge \forall j.\pi_j \models_M f \quad \text{by D4.4} \\ \iff & s \models_M \mathbf{EG}f \quad \text{by D4.4} \end{array}$$

  and we are done by extensionality. □

Effectively this is saying that $\mathbf{EG}f$ is a fixed point of the function $\tau(W) = f \wedge \mathbf{EX}(W)$. We have a similar result for $\mathbf{E}[f\mathbf{U}g]$.

Lemma 4.7 *For any Kripke structure M and CTL formulas f and g,*

$$[\![\mathbf{E}[f\mathbf{U}g]]\!]_M = [\![g \vee (f \wedge \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])]\!]_M$$

Proof For any $s \in S$,

$$\begin{array}{rl} & s \models_M \mathbf{E}[f\mathbf{U}g] \\ \iff & \exists\pi.\pi_0 = s \wedge \exists k.\pi_k \models_M g \wedge \forall j.j < k \Rightarrow \pi_j \models_M f \quad \text{by D4.4} \end{array}$$

Now consider the cases $k = 0$ and $k \neq 0$. In the first case, we have $s \models_M g$. In the second case, we have $\forall j.j < k \Rightarrow \pi_j \models_M f$. But $k \neq 0$ so certainly $\pi_0 \models_M f$, i.e. $s \models_M f$. Further, $\pi_1 \models_M \mathbf{E}[f\mathbf{U}g]$, i.e. $s \models_M \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])$, by Definition 4.4. Since one of the two cases on k must hold, we have $s \models_M g \vee (s \models_M f \wedge s \models_M \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])$ and we have the required result by Definition 4.4. □

4.3 The Translation

Formulas of L_μ also describe properties of a system that can be represented as a state machine. As with CTL, the semantics of a formula is the set of states of the system for which the formula holds true.

The greater expressive power of L_μ requires a slightly modified version of the Kripke structure presented for CTL. For L_μ we define a Kripke structure as in Definition 2.1. Recall that instead of a single transition relation, we now have a set of transition relations called actions. Note that the transition relations for L_μ need not be total and therefore paths need not be infinite. The syntax and semantics of L_μ that we follow are those in Definition 2.2 and Definition 2.3 respectively.

The semantics for both CTL and L_μ are in terms of sets of states. This allows a purely syntactic translation scheme [36].

Definition 4.8 *The translation $\mathcal{T}$ from CTL to L_μ is defined by primitive recursion over CTL formulas as follows*

- $\mathcal{T}(True) = True$
- $\mathcal{T}(p \in AP) = p$
- $\mathcal{T}(\neg f) = \neg\mathcal{T}(f)$
- $\mathcal{T}(f \wedge g) = \mathcal{T}(f) \wedge \mathcal{T}(g)$
- $\mathcal{T}(\mathbf{EX}f) = \langle . \rangle \mathcal{T}(f)$
- $\mathcal{T}(\mathbf{EG}f) = \nu Q.\mathcal{T}(f) \wedge \langle . \rangle Q$
- $\mathcal{T}(\mathbf{E}[f\mathbf{U}g]) = \mu Q.\mathcal{T}(g) \vee (\mathcal{T}(f) \wedge \langle . \rangle Q)$

We need to prove this translation correct with respect to the semantics. Since the underlying models for CTL and L_μ are slightly different, we need to be able to translate a CTL model into an L_μ model. We overload $\mathcal{T}$ for this purpose.

Definition 4.9 *If M is a Kripke structure as given in Definition 4.1, $\mathcal{T}M$ is the Kripke structure*

$$(AP, S, S_0, \lambda a.R, L)$$

satisfying the construction in Definition 2.1.

We now show that the translation preserves semantics.

Theorem 4.10

$$\forall M f.[\![f]\!]_M = [\![\mathcal{T}(f)]\!]_{\mathcal{T}M}\bot$$

Proof By induction on the definition of f. The propositional fragment is trivial. For **EX**, we observe from Definition 4.4 that π_1 is a state to which the current state has a transition. The cases for **EG** and **EU** follow from Lemmas 4.6 and 4.7 respectively. In the backwards direction of the translation for the fixpoints, we need to use Hilbert's selection

operator ε to extend the possibly finite paths of the L_μ model to the infinite paths of the CTL model. Details of the proof are given in Appendix B. □

The use of ε to extend finite paths to infinite ones is an interesting feature of this proof and we are considering whether the proof can be done without it. This is not idle speculation: one of the approaches to providing automatic α-conversion for theorem provers is based on FM set theory [66]. In FM set theory the axiom of choice does not hold. This is the only time we use choice in our proofs and removing its use makes a port to the FM-set-based system possible.

4.4 CTL Model Checking

The CTL symbolic model checking algorithm is simply a procedure that, given M and a CTL formula f, will return the set (as a BDD) of those states of M that satisfy f. The notation $R(\bar{v}, \bar{v}')$ denotes the transition relation for M, where $\bar{v}$ is shorthand for the vector $(v_1, \ldots, v_n)$ and $\bar{v}'$ denotes the next state vector. $R(\bar{v}, \bar{v}')$ can easily be expressed as a boolean term (and hence a BDD). Due to Proposition 4.5 it suffices to consider only **EX**, **EG** and **EU** from the set of operators.

Recall from Lemmas 4.6 and 4.7 that $\mathbf{EG}f$ is the greatest fix-point (under subset inclusion over state sets) of the function $\tau(Z) = f \wedge \mathbf{EX}Z$ and $\mathbf{E}[f\mathbf{U}g]$ is the least fix-point (under subset inclusion over state sets) of the function $\tau(Z) = f \vee (g \wedge \mathbf{EX}Z)$. These fix-points are computed by iteratively computing approximations, each step involving a computation of all states that still satisfy the required property that are reachable in one more step (also called the *relational product* computation). Since the system is finite state and the approximations are increasing sets, we are guaranteed termination. The model checking algorithm follows.

Definition 4.11 *The* CTL *model checking procedure* $[\![-]\!]^\rho_M$ *is defined recursively over the structure of* CTL *formulae as follows*

- $[\![p \in AP]\!]^\rho_M =$ *the BDD of the set of states of* M *in which* p *is true.*
- $[\![\neg f]\!]^\rho_M = \mathtt{NOT}\,[\![f]\!]^\rho_M$ *and* $[\![f \wedge g]\!]^\rho_M = [\![f]\!]^\rho_M\,\mathtt{AND}\,[\![g]\!]^\rho_M$
- $[\![\mathbf{EX}f(\bar{v})]\!]^\rho_M = [\![\exists \bar{v}[f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]]\!]^\rho_M$*, i.e. the relational product.*
- $[\![\mathbf{E}[f\mathbf{U}g]]\!]^\rho_M = [\![\mu Z.g \vee (f \wedge \mathbf{EX}\,Z)]\!]^\rho_M$ *by Lemma 4.7.*
- $[\![\mathbf{EG}f]\!]^\rho_M = [\![\nu Z.f \wedge \mathbf{EX}\,Z]\!]^\rho_M$ *by Lemma 4.6.*

From Table 2.1 and Theorem 4.10 we can derive a modified version of the model checker in Definition 4.11, this time using representation judgements:

Definition 4.12 *The* CTL *model checking procedure using representation judgements,* $\mathcal{T}[\![-]\!]^\rho_M$*, is defined recursively over the structure of* CTL *formulae as follows*

- $\mathcal{T}[\![p \in AP]\!]^\rho_M = \mathcal{T}[\![\mathcal{T}(p)]\!]^\rho_{\mathcal{T}M}\ \bot$
- $\mathcal{T}[\![\neg f]\!]^\rho_M = \mathcal{T}[\![\mathcal{T}(\neg f)]\!]^\rho_{\mathcal{T}M}\ \bot$ *and* $\mathcal{T}[\![f \wedge g]\!]^\rho_M = \mathcal{T}[\![\mathcal{T}(f \wedge g)]\!]^\rho_{\mathcal{T}M}\ \bot$

- $\mathcal{T}[\![\mathbf{EX}f]\!]^{\rho}_{M} = \mathcal{T}[\![\mathcal{T}(\mathbf{EX}f)]\!]^{\rho}_{\mathcal{T}M} \perp$
- $\mathcal{T}[\![\mathbf{E}[f\mathbf{U}g]]\!]^{\rho}_{M} = \mathcal{T}[\![\mathcal{T}(\mathbf{E}[f\mathbf{U}g])]\!]^{\rho}_{\mathcal{T}M} \perp$
- $\mathcal{T}[\![\mathbf{EG}f]\!]^{\rho}_{M} = \mathcal{T}[\![\mathcal{T}(\mathbf{EG}f)]\!]^{\rho}_{\mathcal{T}M} \perp$

By leveraging Theorem 4.10 and using our L_μ model checker, we have avoided the effort of coding this algorithm from scratch in a fully expansive manner within HOL. Thus adding support for a new temporal logic is as simple as formalising that logic in HOL and proving the correctness of its translation to L_μ (assuming such a translation is possible; this is true of most widely used temporal logics).

4.4.1 Totalising the Transition Relation

According to Definition 4.1, the transition relation R used in CTL model checking must be total, i.e. every state must have an outgoing transition. This need not be the case automatically and *terminal states* with no outgoing transitions may be present. If so, we need a way to totalise R. The least disruptive way of doing this is to add self-loops to all terminal states.

We first need to compute the set of reachable states of the system. A state is considered reachable if it lies on a path from the initial set of states. The formalisation is a simpler version of the fixed-point formalisation in §2.4.

Definition 4.13 *Let* $Reachable\,R\,X = \bigcup_n ReachableRec R\,X\,n$ *where*

$$\begin{aligned} &ReachableRec\,R\,X\;\;0 = X \\ &ReachableRec\,R\,X\;\;(n+1) = \{s\,|s \in ReachableRec\,R\,X\,n \\ &\qquad\qquad \vee \exists s'.R(s',s) \wedge s' \in ReachableRec\,R\,X\,n\} \end{aligned}$$

We then have,

Theorem 4.14

$$\begin{aligned} \vdash \forall\, R\,X\,n.& \\ & ReachableRec\,R\,X\,n = ReachableRec\,R\,X\,(n+1) \\ \Rightarrow\quad & Reachable\,R\,X = ReachableRec\,R\,X\,n \end{aligned}$$

Proof ReachableRec is monotone. Thus we have that

$$\forall i.ReachableRec\,R\,X\,i = \bigcup_{j \leq i} ReachableRec\,R\,X\,j$$

and since we have $ReachableRec\,R\,X\,n = ReachableRec\,R\,X\,(n+1)$, we get

$$\forall i \geq n.ReachableRecRXn = ReachableRecRXi$$

by monotonicity. Then we are done by the definition of *Reachable*. □

The set of reachable states is then computed analogously to the way fixed points are computed in §2.4.2, setting the initial value of X to S_0. Having a way of computing the

set of reachable states is useful in other ways as well. For instance, we use it for deadlock analysis in §6.3.3. However, our computation is primitive in execution and could take advantage of optimisations such as frontier set simplification [50].

We can now proceed with the totalisation. The approach is to redefine the transition relation by setting

$$R^{total}(s, s') = R(s, s') \vee (Reachable(s) \wedge TS(s) \wedge s = s')$$

where *Reachable* is true of a reachable state and *TS* is true of a terminal state. Then $R^{total}(s, s')$ behaves like $R(s, s')$ except that additionally, $R^{total}(ts, ts)$ holds for any reachable terminal state ts. We overload R and TS to represent the term-BDD as well as the term representation. The totalisation is then done as follows:

1. Compute a term-BDD for the set of terminal states using the model checker

   $$TS \equiv \mathcal{T}[\![\mathbf{AX}\, False]\!]^{\rho}_{M}$$

   noting that the semantics of **AX** are not affected by a non-total R.

2. Construct a term-BDD for all states having self-loops

   $$Loops \equiv \rho \bigwedge_i v'_i = v_i \mapsto b_{loops}$$

3. The term-BDD for the totalised R is then given by

   $$\mathsf{BddOr}(R, \mathsf{BddAnd}(Reachable\, R\, S_0\, \bar{v},\ \mathsf{BddAnd}(TS, Loops)))$$

   where we overload $Reachable\, R\, S_0\, \bar{v}$ to represent the term-BDD of the set of reachable states.

Henceforth, in the context of CTL model checking, we will assume that R is total.

4.5 Concluding Remarks

Formalised translations in the context of formal verification are not new. For instance, a translation of LTL to ω-automata has been formalised in HOL [165]. However, they are not common, because even simple semantics-based translations require some amount of manual effort. We have added a new translation that will be useful to us later on, and perhaps to others on account of the popularity of CTL.

As this result has been mechanised in HOL, we can convert a CTL property to L_μ, use the L_μ property checker, and convert the resulting theorem back to a CTL property. In general, we can leverage our existing property checker to verify specifications expressed in a new logic (embeddable in L_μ) without risking unsoundness caused by an incorrect translation.

This may seem trivial for CTL but the translations of other popular logics such as LTL or CTL* into L_μ are considerably more involved [34, 53] and the chance of an incorrect implementation correspondingly higher. Our fully-expansive approach towards integrating model-checking and theorem-proving removes this possibility assuming only the soundness of the HOL kernel and the operating environment.

Chapter 5

Extension II: An abstraction framework

This chapter demonstrates an integration with HOL of a fully automatic counterexample-guided model reduction technique that draws upon both BDD- and SAT-based technologies [37, 38, 39].

5.1 Introduction

Unaided model checking techniques are typically unable to verify real-world examples due to the large number of states involved. Abstraction [52] is considered a promising approach to handling this problem. The idea is to reduce the number of states of a model by abstracting away behaviours not essential to the verification. This usually over-approximates the state space (i.e. some states not reachable in the concrete system may become reachable in the abstract system) and results in a reduction in the size of the BDDs generated during model checking. Other techniques use abstraction dynamically to under-approximate the state space to more directly reduce the size of BDDs [150].

Abstractions can be constructed manually to exploit the structure of the model under consideration. There are also automatic abstraction techniques which exploit symmetries and redundancies in the underlying state space [38, 77, 111, 161].

We now build upon our embedded model checker by adding an automatic abstraction framework. The generic technique is known as counterexample-guided abstraction refinement. We formalise and implement a variant that combines several techniques [37], with some modifications.

The idea is to perform an initial over-approximating abstraction of the model and then check for the required property. A positive result in the abstracted system holds in the concrete system as well (Theorem 5.3). A negative result may be spurious (caused by extra states added during the over-approximation). In this case we iteratively partially concretize (or *refine*) the abstracted state space, until a positive or true negative result is obtained.

As before, the fully-expansive approach is used and all steps in the computation are justified by HOL proofs. This retains the high assurance of soundness, and also allows us to use the decision procedures and simplifiers of HOL without loss of compositionality.

5.2 Abstraction Refinement in HOL

Automatic abstraction techniques reduce (without manual guidance) the number of states of a system, so that it is more amenable to model checking. This is typically done using functional abstraction [38] or Galois connections [77]. Here we use the functional abstraction approach of Clarke et al [37] that also describes a refinement framework. A functional abstraction computes an abstraction function h from the concrete to the abstract state space that is then used to construct the initial abstract model.

As always, our system is represented by a Kripke structure (see Definition 2.1). The only difference is that each transition relation in T is either of the form $\bigwedge_{i=0}^{n-1} v_i' = \phi_i(s)$ for synchronous or $\bigvee_{i=0}^{n-1} v_i' = \phi_i(s)$ for asynchronous systems,[1] where the ϕ_i are *next-state formulas* over the variables v_i of s that determine what the value of each v_i should be in the next state. Each ϕ_i is of the form $\texttt{case}\,\psi_0(s) \rightarrow \chi_0(s) | \ldots | \psi_m(s) \rightarrow \chi_m(s) | \mathtt{T} \rightarrow \chi_{m+1}(s)$ so that v_i' gets the value of $\chi_j(s)$ where $j \leq m+1$ is the smallest number for which $\psi_j(s)$ in $\phi_i(s)$ holds true, ψ_{m+1} being always true. The method we use [37] requires that state transitions be described in this manner.

One modification we have made to this scheme is to consider the ψ_j in their entirety rather than breaking them up into atomic propositions [37]. This is because at the moment our atomic propositions are boolean atoms only and, as we shall see in §5.2.1, if all the ψ_j are boolean atoms than no abstraction takes place.

The abstraction function $h : S \rightarrow \hat{S}$ is a surjection to the set $\hat{S}$ of abstract states $\hat{s}$ (the construction of h is covered in §5.2.1). We use h to compute the abstract model:

Definition 5.1 *Given a model M and an abstraction function $h : S \rightarrow \hat{S}$, the abstract model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T}, \hat{L})$ is generated as follows:*

1. $\hat{S}(\hat{s}) = \exists s.(h(s) = \hat{s}) \wedge s \in S$
2. $\hat{S}_0(\hat{s}) = \exists s.(h(s) = \hat{s}) \wedge s \in S_0$
3. $\hat{T} = \{\xrightarrow{\hat{a}} | \forall a \in T.\hat{s_1} \xrightarrow{\hat{a}} \hat{s_2} \iff \exists s_1 s_2.h(s_1) = \hat{s_1} \wedge h(s_2) = \hat{s_2} \wedge s_1 \xrightarrow{a} s_2\}$
4. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$

$\hat{AP}$ is simply the smallest set of boolean variables required to enumerate $\hat{S}$.

This is called *existential abstraction* since we use existential quantification to hide the variables of the concrete states whose behaviour we wish to ignore.

Definition 5.2 *A* universal *property is a formula f of L_μ such that*

$$\forall\, a\, g.\, \langle a \rangle g \not\sqsubseteq_{NNF} f$$

The abstraction is conserved for universal properties only. We can now give the main result.

[1] In synchronous systems, all transition happen simultaneously, thus the system transition relation is the conjunction of the individual transition betweens states. For asynchronous systems, transitions may interleave, so the system transition relation is a disjunction.

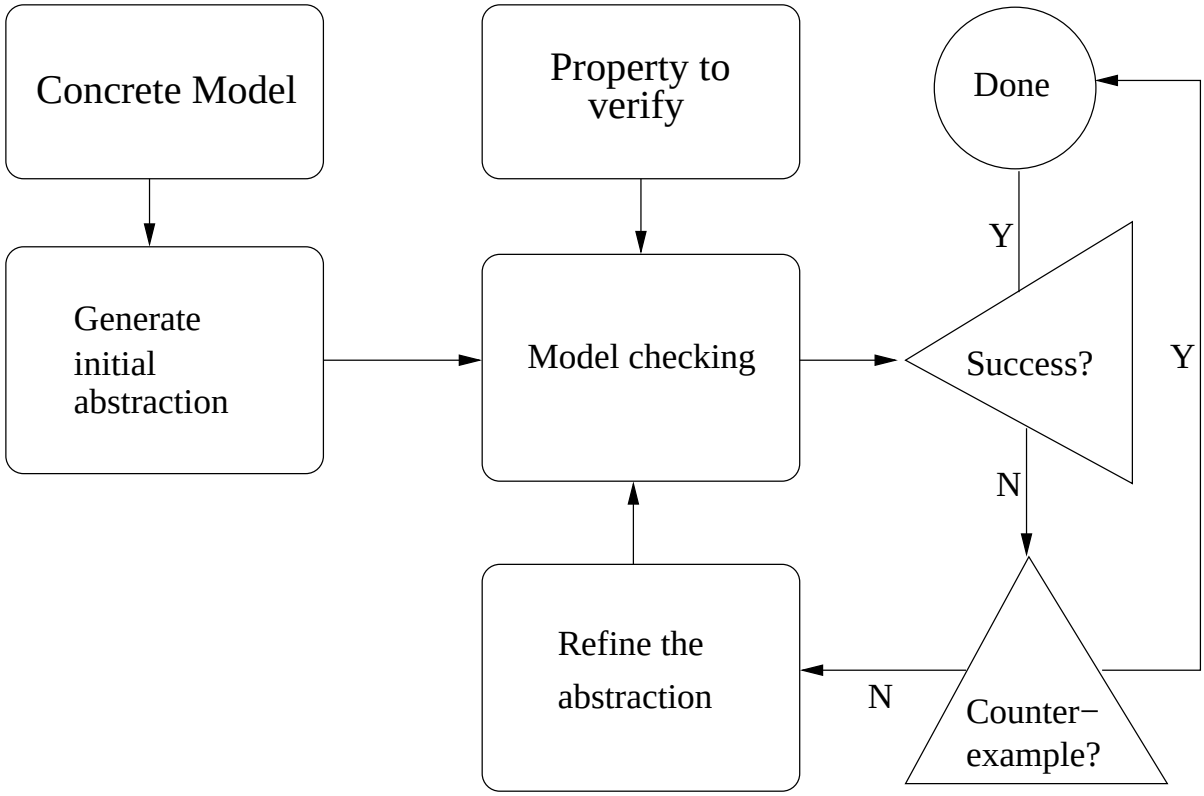

Figure 5.1: Overview of Abstraction Refinement Framework

Theorem 5.3 *For any universal property* f, $\hat{M} \vDash f \Rightarrow M \vDash f$

Proof Sketch By induction on f. The abstraction adds extra behaviours to $\hat{M}$ that were not present in M. However it does not take away any behaviour, i.e. it computes an over-approximation. Thus if a property holds in the abstract state space it will hold in the concrete state space. If a property fails in the abstract it may be because of the spurious behaviour. Note that this does not hold for a non-universal property such as $\langle . \rangle g$ because it might hold true in the abstract system on account of transitions not present in the concrete system. A universal property never asserts the presence of a transition and avoids this problem. □

This proof is done partially at runtime because at the moment it relies on the actual abstraction function in use, which only becomes available during execution. Ideally, the entire proof should be offline.

Figure 5.1 shows the overall framework. If we limit ourselves to universal properties, then if a property fails in the abstract system we can generate a counterexample trace in the abstract system and attempt to find a corresponding concrete trace. If one exists then the property is false in the concrete system and the verification fails. Otherwise the abstract counterexample was spurious and abstraction is too coarse so we refine it by splitting some abstract state (found by analysing the counterexample) into smaller sets of concrete states. We then recheck the property. This is continued until either the property is verified or a concrete counterexample found. We are guaranteed termination because each refinement is strict (i.e. all the sets resulting from splitting an abstract state

are non-empty) and eventually we will end up with the concrete system that cannot be further refined.

5.2.1 Generating the Initial Abstraction

The first step is to generate the initial abstraction. The idea is to compute an abstraction function h that can then be used to generate the abstract model $\hat{M}$ as in Definition 5.1. Intuitively, this function groups together into one abstract state all concrete states that cannot be distinguished by the atomic propositions in AP. For example, if $AP = \{x < y, x > y\}$ then a state set $\{(1,2),(3,4),(6,5),(9,8)\}$ would be partitioned into $\{(1,2),(3,4)\}$ and $\{(6,5),(9,8)\}$. One additional improvement is to first partition AP so that no partition has a free variable in common.

More precisely,

1. Let $(v_0, v_1, \ldots, v_n)$ be any state s_i.
2. Let the transition relation $\rightarrow$ be given by $\bigwedge_i v_i' = \phi_i(s)$ (or $\bigvee_i v_i' = \phi_i(s)$ for asynchronous systems). Let F be the set of all ψ_j occuring in the next-state formulae ϕ_i.
3. Let $\psi_i \equiv_\psi \psi_j \iff vars(\psi_i) \cap vars(\psi_j) \neq \emptyset$.
4. Let FC_i be the m partitions of F induced by $\equiv_\psi$.
5. Let $VC_i = \bigcup_{\psi \in FC_i} vars(\psi)$. Let $D_{VC_i} = \prod_{v_j \in VC_i} D_{v_j}$.
6. Let $h_i : D_{VC_i} \rightarrow \hat{D}_i$ (where $\hat{D}_i$ is a component of the abstract domain) be defined by $h_i(s_j) = h_i(s_k) \iff \forall \psi \in FC_i . s_j \vDash \psi \iff s_k \vDash \psi$
7. Then $h : \prod_i D_{VC_i} \rightarrow \prod_i \hat{D}_i = (h_0, \ldots, h_{m-1})$.

Note that $\prod_i D_{VC_i} = S$ up to reordering of the positions of the $v_i \in V$ in the state tuple. Since the actual computation for application of h to concrete states occurs in the BDD operations where this order is irrelevant, an explicit reordering is not required. Thus we can consider the domain of h to be S without any problems.

The manner in which h is computed in step 6 above is the reason why we do not break the ψ-formulas down to boolean atoms (we remarked on this on page 54 in §5.2). Clearly, no two states will agree on the value of all boolean atoms unless they are the same state and so each abstract state would contain only one concrete state, defeating the purpose of the whole exercise.

We can now construct $\hat{M}$. Once we have $\hat{M}$, we have no further use for h. Technical details about the construction of h are covered in §5.3.1.

5.2.2 Counterexample Generation

A counterexample is a sequence of states starting with an initial state and following transitions to a state violating the property being verified.

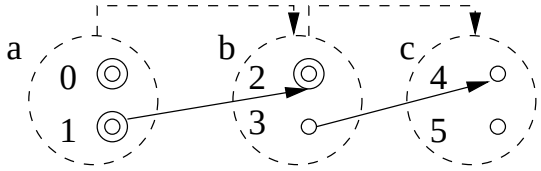

Figure 5.2: Counterexample Detection Example

Computing a path to a given set of states using representation judgements in HOL has already been done [74]. Given a model M and a target set X, the method returns a list of theorems of the form

$$[s_0 \xrightarrow{a_0} s_1, s_1 \xrightarrow{a_1} s_2, \ldots, s_{n-1} \xrightarrow{a_{n-1}} s_n]$$

tracing out a path in M such that $s_0 \in S_0$ and $s_n \in X$.

This is easily adapted to generate counterexamples. We set X to the set of states satisfying the property being checked (this set already having been computed by a model checking run that ended in failure) and retrieve the desired list of theorems. It is then a simple matter of using L_μ semantics together with the structure of M to represent the counterexample by the judgement

$$\rho\, s \vDash^{\perp}_{M} \langle a_0 \rangle \langle a_1 \rangle \ldots \langle a_{n-1} \rangle \neg X \mapsto b$$

where each $a_i \in T$.

Depending on circumstances, either this or the earlier form of the counterexample can be used for further analysis.

5.2.3 Concrete Counterexample Detection

If the verification fails, the model checker can generate a counterexample trace. Our technique for detecting whether a concrete counterexample exists uses SAT solvers [39] and is best illustrated by an example.

Figure 5.2 shows a system with $S = \{0, 1, 2, 3, 4, 5\}$. The abstract states are $\{a, b, c\}$ and the dashed circles indicate the concrete states they contain. Solid arrows represent transitions in the concrete system and dashed arrows represent transitions in the abstract system. The initial states are 0 and 1 and concentric-circles represent states reachable from the initial states. Note that the abstraction introduces extra behaviour by making states 4 and 5 reachable in the abstract system by making c reachable.

Suppose we check for a property f that holds in $\{0, 1, 2, 3\}$ but not in $\{4, 5\}$. We will get an abstract counterexample trace[2] $\langle a, b, c \rangle$.

In general, given an abstract counterexample $\langle \hat{s_0}, \hat{s_1}, \ldots, \hat{s_k} \rangle$, we attempt to find a corresponding concrete counterexample $\langle s_0, s_1, \ldots, s_k \rangle$. To determine whether such a concrete trace exists, we try to find a satisfying assignment for the formula

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} s_i \dot{\rightarrow} s_{i+1} \wedge \bigwedge_{i=0}^{k} h(s_i) = \hat{s_i}$$

[2]We overload the $\langle - \rangle$ notation to represent a sequence of states.

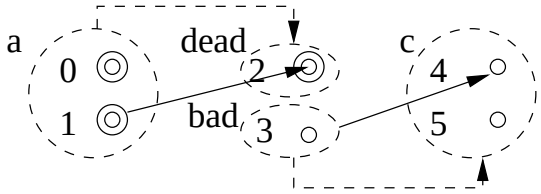

Figure 5.3: Abstraction Refinement Example

using a SAT solver.

If a concrete trace is found, the verification has failed and we are done. Otherwise we set $k = 0$ and attempt to find the longest prefix of $\langle \hat{s_0}, \ldots, \hat{s_k} \rangle$ for which there is a concrete trace by running the SAT tool on the formula above for increasing values of k.

In our example, there is no concrete counterexample and the longest prefix is $\langle a, b \rangle$. This information is then used to refine the offending (because it is at the end of the prefix) state b into smaller abstract states.

5.2.4 Refining the Abstraction

We would like to find the coarsest possible refinement, i.e. the fewest possible splits of the abstract state. Our technique for refinement employs BDDs to get good results. We developed this technique ourselves since the work we are using [37, 39] provides incompatible refinement methods.

Consider Fig. 5.3 representing the same system as Fig. 5.2. We know that we need to refine abstract state b. Since the spurious behaviour is created by an unreachable state 3 in b having a transition to 4, we need to split all such states from the reachable states (in this case just $\{2\}$).

To do this, we compute the state sets

$$bad = \mathbf{EX}c = \{3\}$$

and

$$dead = b \backslash bad = \{2\}$$

where $\mathbf{EX}f$ computes all states such that there is a transition to a set in which the property represented by f is satisfied. In our case the property c yields precisely the set $\{4, 5\}$. This computation is done using standard BDD methods.

Intuitively the *dead* states are reachable "dead-ends" and the *bad* states are the ones causing the trouble by contributing to the creation of spurious behaviour.

We can now replace the abstract state b in the abstraction by the abstract states *dead* and *bad* and repeat the procedure until the property is verified or a counterexample is found. In this case the abstract state c is no longer reachable and thus f is verified.

In a more complex system, several such refinements may be required before the either the property is verified or a concrete counterexample is found.

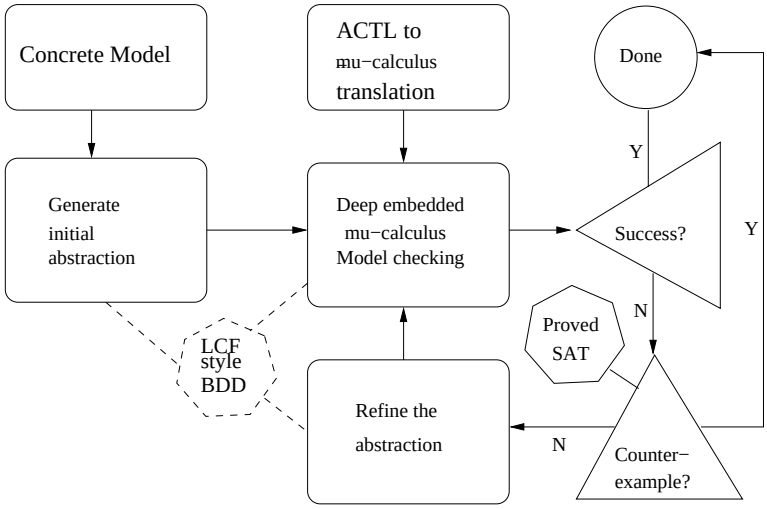

Figure 5.4: Overview of Implementation in HOL

5.3 Implementation Issues

Figure 5.4 gives an overview of the implementation. It should be noted that the system is fully automatic and relies on fully-expansive proof at every step of the way.

The only exception to this are the dotted lines indicating the use of an external BDD engine. However, we use an LCF-style interface [74] to this engine which gives higher assurance of soundness than integrating it as a one-shot proof rule as has been done in [14, 154].

The SAT engine we used is also external to HOL [73, 94]. However, checking the satisfiability of an assignment is in general much easier than finding such an assignment. Thus we use the SAT engine to obtain an assignment but check its validity by proof in HOL. Thus the use of the SAT engine does not risk introducing unsoundness in the system.

The ACTL to L_μ translation is not a requirement. However, recall that only universal properties can be used in this framework. L_μ is a fairly non-intuitive logic and it is hard to manually check that an L_μ property is indeed universal. Thus we also accept properties in the more intuitive logic ACTL, which is the universal fragment of CTL. That a formula is in ACTL can easily be checked by a predicate on CTL formulas. Thus universality is enforced automatically because the HOL translation from CTL to L_μ in Chapter 4 is done by proof based on the semantics of the two logics.

5.3.1 Constructing Equivalence Classes

Note that step 6 of initial abstraction generation is in effect inducing equivalence classes of concrete states over each component D_{VC_i} of S. When constructing a BDD representation of h to assist with the construction of $\hat{M}$ it is easier to compute these equivalence classes directly.

However, the standard BDD methods for finding equivalence classes work by detecting strongly connected components (SCCs) in the graph representation of the model. SCC

detection requires a total transition relation which is usual when verifying properties in CTL. This is not guaranteed in our more general case with L_μ. We have devised the following algorithm for partitioning a given D_{VC_i} with respect to satisfiability of the $\psi \in FC_i$.

The term $h_i(s_j) = h_i(s_k)$ can be considered as a relation $R'(s_j, s_k)$ on states. This relation induces the partitions we wish to compute. However R' can also be considered a transition relation on D_{VC_i}, with there being a transition between two states precisely when h_i agrees on them as defined by step 6 of §5.2.1.

Define the modality **EP** as the temporal inverse of **EX**, i.e. $\mathbf{EP}f$ computes states such that there is a transition from states satisfying f to these states.

Now we compute as follows:

1. Let $X = D_{VC_i}$. Let $P = []$.
2. If $X = \emptyset$ we are done.
3. Let s be an arbitrary "seed" state in X (found using the BDD engine's satisfying assignment finder).
4. Set $S = X, T = R'$ and use the model checker to compute[3]

$$Y = \mathcal{T}[\![(\mu Q.s \vee \mathbf{EX}Q) \vee (\mu Q.s \vee \mathbf{EP}Q)]\!]^{\rho}_{M} \perp$$

   Then Y is the set of all states from which s is reachable together with the set of all states reachable from s.
5. Let $P = Y :: P$
6. Let $X = X \backslash Y$ and repeat from step 2.

At the end of this the list P will contain the required partitions. Now each abstract state in $\hat{D}_i$ corresponds to one of these partitions. Thus we need $\hat{n} = \lceil \log_2 |P| \rceil$ abstract state variables $\hat{v}_0, \ldots, \hat{v}_{\hat{n}}$. Our formalisation equates each element Y in P with the predicate $SCC\, M\, s_i$ where s_i was the particular seed state used to generate that Y. More precisely,

$$SCC\, M\, s_i = ReachFrom\, M\, s_i \cup ReachTo\, M\, s_i$$

where

$$ReachFrom\, M\, s_i = [\![\mu Q.s_i \vee \mathbf{EX}Q]\!]_M \perp$$

and

$$ReachTo\, M\, s_i = [\![\mu Q.s_i \vee \mathbf{EP}Q]\!]_M \perp$$

The required fixed-point theorems for *ReachFrom* and *ReachTo* are formalised in a manner similar to the predicate *Reachable* in §4.4.1. In fact, *ReachTo* and *Reachable* are exactly the same.

[3]We abuse notation here by mixing L_μ and CTL operators. This is not a problem as L_μ admits a syntactic embedding of CTL. We also need to add the ability to compute **EP** to the model checker. This is easy.

Then the judgement representing the component h_i of the abstraction function looks like

$$\rho \bigwedge_{i}^{|P|} SCC\ M\ s_i = t(i) \mapsto b$$

where the function t simply generates a binary representation of i in terms of the abstract state variables, e.g. $t(0) = \neg\hat{v}_0 \wedge \ldots \wedge \neg\hat{v}_{\hat{n}}$ and so on. $|P|$ can be quite large, but since we have no use for the term representation of h_i, we once again use the trick of abbreviating the term part of the judgement above using a predicate specially defined for the purpose. Nevertheless, simply having to generate such a large term is less than ideal and we are looking at ways of abbreviating the term on the fly.

5.4 Related Work

Abstraction refinement for model checking in the context of theorem proving has also been done before [14, 161].

The first work [161] presents an abstraction framework based on Galois connections in which refinement is done by adding the failed predicates from the previous proof attempt to get a richer abstract domain. Since the system is implemented as an atomic proof rule, access to the procedures and simplifiers in PVS itself for the purposes of the system cannot be done within the encompassing derivation tree. This inhibits a fully-expansive implementation of the system. It also restricts compositionality because, for instance, the system is unable to return a counterexample trace upon failure thus any counterexample guided predicate discovery system [54] cannot be used.

This work as well as the method we have used were both influenced by earlier work on refinement [57], in which both the abstraction and refinement processes are manual but the general approach is the same.

The second work [14] has a proof system for L_μ that relies on proof rules being executed by various decision procedures than can be plugged in according to need. This gives the framework great versatility in the choice of tools to be used to attack a given problem. More details on this work are given in §8.4.

5.5 Conclusion

Our approach is flexible in that any proof rule of HOL at any level of abstraction can be called upon at any time during the execution of the procedure. This enables us to provide an entire run of the procedure as a derivation tree that can be plugged into any other proof. This high level of integration guarantees compositionality and makes the framework extendible. Thus we are able to use a BDD engine, a SAT engine and various HOL procedures in the same framework.

At the same time, the execution is fully expansive. All steps are accompanied by the application of a HOL proof rule. Thus we have a high assurance of soundness.

However, having to do fully expansive proof for the equivalent of fast BDD operations necessarily involves a performance penalty. We can ameliorate this by manipulating the higher order equivalents of the propositional terms being manipulated by the BDD and

SAT engines. Performance optimizations are often non-trivial and not just a question of better engineering (see end remarks in §5.3.1 for example). The performance is being improved continually, but it may be some time before we can compete with state-of-the-art non-proof-driven systems.

In the current system the $p \in AP$ are restricted to propositional expressions. We hope to extend this to include Presburger formulae and – trading off some automation – full arithmetic and real numbers, leveraging the facilities in HOL for deciding these.

For concreteness, Appendix C demonstrates this system for a trivial example.

Chapter 6

Case study I: A bus architecture

We have seen how our platform can be seamlessly and securely extended to support new model checking techniques. However, we have yet to exhibit any synergy (other than increased confidence in the soundness of the tool) from the combination thus achieved. In this chapter we use a case study to demonstrate how the theorem proving component can help with the verification in various ways, and also that the model checking component, though in its infancy, can handle more than just toy examples.

Typical microprocessor and memory verifications assume direct connections between processors, peripherals and memory, and zero latency data transfers. They abstract away the data transfer infrastructure as it is not relevant to the verification. However, this infrastructure is in itself quite complex and worthy of formal verification.

The Advanced Microcontroller Bus Architecture[1] (AMBA) is an open System-on-Chip bus protocol for high-performance buses on low-power devices. In this chapter we implement a simple model of AMBA and use the tools developed so far to check latency, arbitration, coherence and deadlock freedom properties of the implementation.

6.1 AMBA Overview

The AMBA specification defines three buses:

- Advanced High-performance Bus (AHB): The AHB is a system bus used for communication between high clock frequency system modules such as processors and on-chip and off-chip memories. The AHB consists of bus masters, slaves, an arbiter, a signal multiplexor and an address decoder. Typical bus masters are processors and DMA devices.
- Advanced System Bus (ASB): The ASB is also a system bus that can be used as an alternative to the AHB when the high-performance features of AHB are not required.
- Advanced Peripheral Bus (APB): The APB is a peripheral bus specialised for communication with low-bandwidth low-power devices. It has a simpler interface and lower power requirements.

[1]

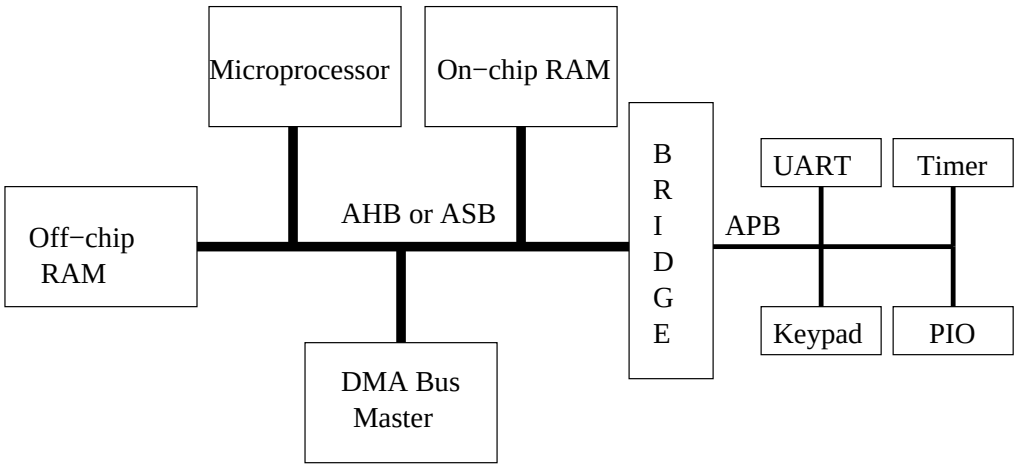

Figure 6.1: Typical AMBA-based Microcontroller

Designers can use either the AHB or the ASB in conjunction with the APB. The APB has a single bus master module that acts as a bridge between the AHB or ASB and the APB. The AMBA specification is hardware and operating system independent and requires very little infrastructure to implement. Figure 6.1 shows a typical AMBA-based microcontroller. We follow revision 2.0 of the AMBA specification [8].

6.2 AMBA APB

The APB is optimized for low power consumption and low interface complexity. It is used for connecting the high-bandwith system bus to low-bandwidth peripherals such as input devices. There is a single bus master, a single global clock and all transfers take two cycles. The bus master also acts as a bridge to the system bus, to which it can be connected as a slave. The address and data buses can be up to 32 bits wide.

6.2.1 Specification

The operation of the APB consists of three stages, all of them are triggered on the rising edge of the clock:

1. *IDLE*. This is the initial and the default state of the bus when no transfer is underway.

2. *SETUP*. The first stage of a transfer is a move to the SETUP state. The address, data and control signals are asserted during this phase but may not be stable. This stage always lasts for one clock cycle and then the operation moves to the ENABLE stage.

3. *ENABLE*. The address, data and control signals are stable during this phase. This phase also lasts one clock cycle and then moves to the SETUP or the IDLE stage depending on whether or not another transfer is required.

Table 6.1: AMBA APB Signals

Signal	Description
PCLK	The bus clock. The rising edge is used to trigger all APB signals.
PRESET	The reset signal. Resets the bus to the IDLE state. It is the only signal that is active low.
$PSEL_x$	This signal indicates that slave x is selected and a transfer is required, thus moving the bus from the IDLE to the SETUP stage. There is a unique line for each slave on the bus.
PENABLE	This signal triggers a move from the SETUP to the ENABLE stage, when the data and address buses are actually sampled.
PWRITE	When high this signal indicates a write access, when low a read access.
PADDR[31:0]	The address bus. Can be up to 32 bits wide and is driven by the bus master.
PRDATA[31:0]	The read data bus. It can be up to 32 bits wide and is driven by the selected slave (see $PSEL_x$) during a read access.
PWDATA[31:0]	The write data bus. It can be up to 32 bits wide and is driven by the bus master during a write access.

Table 6.1 lists all APB signals and their function. Each signal name is prefixed with P to denote that this is an APB signal.

Bus Master

There is a single bus master on the APB, thus there is no need for an arbiter. The master drives the address and write buses and also performs a combinatorial decode of the address to decide which $PSEL_x$ signal to activate. It is also responsible for driving the $PENABLE$ signal to time the transfer. It drives APB data onto the system bus during a read transfer.

Slave

An APB slave drives the bus during read accesses. This can be done when the appropriate $PSEL_x$ is high and $PENABLE$ goes high. $PADDR$ is used to determine the source register.

In a write transfer, it can sample write data at the edge of $PCLK$ or $PENABLE$, when its $PSEL_x$ signal is high. Then $PADDR$ can be used to determine the target register.

6.2.2 Implementation

We implement the APB by following the specification in a straightforward manner without any optimizations. We need to implement the model as a state machine M_{APB}. In this case, a state is a tuple of all the signals considered as boolean variables. We use the standard convention of using primes to denote components of the target (or next) state in a transition.

Definition 6.1

$$\begin{aligned}\bar{s}_{APB} &= (PCLK, PRESET, PSEL_x, PENABLE, PWRITE,\\ &\quad PADDR[31:0], PRDATA[31:0], PWDATA[31:0])\end{aligned}$$

and $\bar{s}'_{APB}$ *represents* $\bar{s}_{APB}$ *with all components primed.*

Note that $PSEL_x$ represents several variables, and $PADDR[31:0]$, $PRDATA[31:0]$ and $PWDATA[31:0]$ can each represent up to 32 variables. Henceforth, we will use this notational convention to abbreviate parameterised signals and address and data buses.

Assumptions

Some simplifying assumptions:

1. All signals are valid throughout, i.e. there is no glitching.

2. Sub-cycle timing (i.e. timing delays between signals becoming stable after changing) is ignored.

3. Since there is a single global clock triggering all signals, transitions of the state machine are synchronous. For the same reason, it suffices to model the clock implicitly by equating one transition of the system to one clock cycle.

4. Endian-ness is not fixed, but is required to be consistent throughout.

5. We do not model reset as it is easy to do so but its presence trivially guarantees absence of deadlock.

These assumptions preserve the properties of the model that we are interested in.

The Model

We first need to define our state machine as a Kripke structure following Definition 4.1. S and L are defined in the obvious manner. M_{APB} is then described by an initial states predicate S_{0APB} on states, and a transition predicate R_{APB} which is a relation on states and is a conjunction – since the state machine is synchronous – of the transition relations for the components of the bus. As much as possible of the internal behaviour of the master and slaves has been abstracted.

The initial states predicate says simply that we start in the $IDLE$ stage.

Definition 6.2

$$S_{0APB}(\bar{s}_{APB}) = \bigwedge_x \neg PSEL_x \wedge \neg PENABLE$$

We need two transition relations, for the master and for slaves.

Definition 6.3

$$
\begin{aligned}
& R^{master}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& (PENABLE' \iff PSEL_x \wedge \neg PENABLE) && (6.1)\\
\wedge\ & (PWRITE' \iff PSEL_x \Rightarrow PWRITE) && (6.2)\\
\wedge\ & (PADDR'_b \iff PSEL_x \Rightarrow PADDR_b) && (6.3)\\
\wedge\ & ((PSEL_x \iff \neg PENABLE) \Rightarrow PSEL'_x) && (6.4)\\
\wedge\ & (Mst'_{r,b} \iff \text{if}\,(\neg PWRITE \wedge ((r,b) = DECODE(PADDR)) \\
& \qquad\qquad \wedge PSEL_x \wedge PENABLE) \\
& \qquad \text{then}\, Slv_{x,r,b}\, \text{else}\, Mst_{r,b}) && (6.5)
\end{aligned}
$$

Note that some of the transition conditions represent schema. $PADDR_b$ represents line b of the address bus, $Mst_{r,b}$ represents bit b of register r of the master, and $Slv_{x,r,b}$ represents bit b of register r of slave x, where the x is the same as the x in $PSEL_x$. We use Mst and Slv to model actual master and slave registers because it is easier to check coherency properties this way rather than by modelling the data buses $PRDATA$ and $PWDATA$, specially since we are ignoring glitching and sub-cycle timing.

Line 6.1 of Definition 6.3 drives $PENABLE$ to high immediately after the cycle in which $PSEL_x$ goes high. Line 6.2 latches the value of $PWRITE$ once $PSEL_x$ is high. Line 6.3 does the same for $PADDR$. Line 6.4 ensures that $PSEL_x$ stays high in the ENABLE stage. Finally, line 6.5 ensures that the master regisers are updated correctly; the $DECODE$ function recovers which bit of which register of the master is to be updated.

Slaves have a very simple transition relation.

Definition 6.4

$$
\begin{aligned}
& R^{slave}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& Slv'_{x,r,b} \iff \text{if}\,(PWRITE \wedge ((r,b) = DECODE(PADDR)) \\
& \qquad\qquad \wedge PSEL_x \wedge PENABLE) \\
& \qquad \text{then}\, Mst_{r,b}\, \text{else}\, Slv_{x,r,b}
\end{aligned}
$$

The definition ensures that slave registers are updated correctly.

The APB transition relation is just the conjunction of the transition relations for the master and slave modules.

Definition 6.5

$$
\begin{aligned}
& R_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& R^{master}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) \wedge R^{slave}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB})
\end{aligned}
$$

6.2.3 Verification

We verify three types of properties for our APB implementation. In all cases, a property is considered verified if the set of satisfying states include the initial states. This condition can be built into the properties but we do not do so to avoid computing the set of initial states repeatedly.

Latency

Latency properties check that the bus becomes available within a given number of cycles. We can use them to check that wait and/or transfer times do not exceed design specifications. In our case, we want to confirm that all transactions take precisely two cycles.

Unfortunately this property cannot be represented in CTL, since it needs to be of the schematic form

$$\mathbf{AG}(\mathbf{A}(\mathbf{X}(\neg PENABLE \wedge PSEL_x) \Rightarrow \mathbf{XAXAX}(PSEL_x \wedge PENABLE)))$$

which is a CTL* property.

We have not yet implemented a translation from CTL* to L_μ, so we are unable to check this property. The best we can do with CTL is the property schema

$$\mathbf{AG}(\neg PENABLE \wedge PSEL_x \Rightarrow \mathbf{AX}(PSEL_x \wedge PENABLE))$$

which checks that once a transfer starts, it finishes in the next cycle. Running this through the model checker returns the required theorem.

Theorem 6.6

$$\begin{array}{l} \vdash \forall \bar{s}_{APB}. \\ \quad \bar{s}_{APB} \models_{M_{APB}} \mathbf{AG}(\neg PENABLE \wedge PSEL_x \\ \qquad\qquad\qquad\qquad \Rightarrow \mathbf{AX}(PSEL_x \wedge PENABLE)) \end{array}$$

Theorem 6.6 is actually a family of theorems indexed by x, since the property was stated as a schema. Each theorem in the family is model checked separately. This applies to all theorems in this chapter that correspond to property schema, though we shall refer to each family as a Theorem to preserve the correspondence with the the associated property.

We mention in passing that the model checker would actually have returned the theorem

$$\begin{array}{l} \vdash \forall \bar{s}_{APB}. \\ \quad \bar{s}_{APB} \models^{\perp}_{\mathcal{T}(M_{APB})} \mathcal{T}(\mathbf{AG}(\neg PENABLE \wedge PSEL_x \\ \qquad\qquad\qquad\qquad \Rightarrow \mathbf{AX}(PSEL_x \wedge PENABLE))) \end{array}$$

from which Theorem 6.6 is derived using Theorem 4.10. This applies to all other theorems returned by the model checker where we checked for CTL properties.

We can express the CTL* property directly in L_μ but this approach is best avoided as we currently lack the safety net of a formal translation from CTL* and L_μ is fairly non-intuitive to work with. However, we are able to get around this problem in the next section.

Coherence

Coherence properties check data coherency, i.e. registers are updated correctly at the end of transfers. Since transfers are multi-cycle, target registers are not updated immediately.

Thus by checking that the update happens in precisely two cycles, we can also check the transfer time. The required CTL property schema is

$$\begin{array}{ll}
\mathbf{AG} & \\
 & ((\neg PENABLE \wedge PSEL_x \wedge PWRITE \\
 & \quad \wedge ((r,b) = DECODE(PADDR)) \\
 & \qquad \Rightarrow ((\mathbf{AXAX}Slv_{x,r,b}) \iff Mst_{r,b})) \\
\wedge & (\neg PENABLE \wedge PSEL_x \wedge \neg PWRITE \\
 & \quad \wedge ((r,b) = DECODE(PADDR)) \\
 & \qquad \Rightarrow (Slv_{x,r,b} \iff (\mathbf{AXAX}Mst_{r,b}))))
\end{array}$$

in which we can check coherency and a two-cycle transfer time simultaneously. The two conjuncts check for coherency during write and read cycles respectively. The model checker returns the required theorem.

Theorem 6.7

$$\begin{array}{ll}
\vdash & \forall \bar{s}_{APB}. \\
 & \quad \bar{s}_{APB} \models_{M_{APB}} \\
\mathbf{AG} & \\
 & ((\neg PENABLE \wedge PSEL_x \wedge PWRITE \\
 & \quad \wedge ((r,b) = DECODE(PADDR)) \\
 & \qquad \Rightarrow ((\mathbf{AXAX}Slv_{x,r,b}) \iff Mst_{r,b})) \\
\wedge & (\neg PENABLE \wedge PSEL_x \wedge \neg PWRITE \\
 & \quad \wedge ((r,b) = DECODE(PADDR)) \\
 & \qquad \Rightarrow (Slv_{x,r,b} \iff (\mathbf{AXAX}Mst_{r,b}))))
\end{array}$$

Deadlock Freedom

In concurrency theory, the term *deadlock* refers to an abnormal termination or freeze of the system. In terms of automata such as Kripke structures, this may be represented by a state with no outgoing transitions.

We can check that this undesirable situation does not occur. Since our transition relation has been defined by assigning all next-state variables some value in each cycle, the simple CTL property

$$\mathbf{AG}\,\mathbf{EX}\,True$$

(to check that there is no terminal state) is in a sense vacuously true and does not tell us anything.

On account of this, we need to have some criterion for system deadlock. We know that once a transfer is underway, it always completes, by Theorem 6.6. So it remains only to check that a transfer can always be initiated. This can be checked by the following property schema:

$$\mathbf{AG}(\mathbf{AF}(PSEL_x \bar{\oplus} PENABLE \Rightarrow \mathbf{EX}\,PSEL_y))$$

where $\bar{\oplus}$ is negated exclusive-OR. This property checks that $PSEL$ (for any slave) can go high if the APB is idle or has just finished a transfer. The model checker returns the required theorems.

Theorem 6.8

$$\vdash \forall \bar{s}_{APB}.\ \bar{s}_{APB} \vDash_{M_{APB}} \mathbf{AG}(\mathbf{AF}(PSEL_x \bar{\oplus} PENABLE \Rightarrow \mathbf{EX}\, PSEL_y))$$

6.3 AMBA AHB

The AHB is a pipelined system backbone bus, designed for high-performance operation. It can support up to 16 bus masters and slaves that can delay or retry on transfers. It consists of masters, slaves, an arbiter and an address decoder. It supports burst and split transfers. The address bus can be up to 32 bits wide, and the data buses can be up to 128 bits wide. As before, there is a single global clock.

We choose to model the AHB rather than the ASB because the AHB is a newer design and also because it has been designed to integrate well with the verification and testing workflow.

6.3.1 Specification

The operation of the AHB is too complex to be specified in terms of a few fixed stages. A simple transfer might proceed as follows (the list numbering below is not cycle accurate):

1. The AHB is in the default or initial state. No transfer is taking place, all slaves are ready and no master requires a transfer.
2. Several masters request the bus for a transfer.
3. The arbiter grants the bus according to some priority-scheduling algorithm.
4. The granted master puts the address and control information on the bus.
5. The decoder does a combinatorial decode of the address and the selected slave samples the address.
6. The master or the slave put the data on the bus and it is sampled. The transfer completes.

Items 4-5 above constitutes the *address phase* of a transfer, and 6 constitutes the data phase. Since the address and data buses are separate, the address and control information for a transfer are driven during the data phase of the previous transfer. This is how transfers are pipelined. Several events can complicate the basic scenario above:

- The master or the slave may extend the transfer by inserting idle cycles or wait states during the transfer.
- The master may indicate a burst in which case several transfers occur end-to-end.

Table 6.2: AMBA AHB Master Signals

Signal	Driver	Description
HADDR[31:0]	Master	The address bus. Up to 32 bits wide.
HTRANS[1:0]	Master	Indicates the type of the current transfer. These can be idle (IDLE), busy (BUSY), non-sequential (NSQ) or sequential (SEQ).
HWRITE	Master	When high this indicates a write transfer (master to slave) and a read when low.
HBURST	Master	Indicates if the transfer forms part of a burst.
HWDATA[127-31:0]	Master	The write data bus. Can be up to 128 bits wide.
$HBUSREQ_x$	Master	When high indicates to the arbiter that master x is requesting the bus. There is a separate bus request line for each master.

- The slave may report an error and abort the transfer.
- The slave may signal a split or a retry, indicating it cannot at the moment proceed with the transfer. In this case the master may relinquish the bus and complete the transfer later (split) or not leave the bus and complete the transfer once the slave is ready (retry).

Tables 6.2, 6.3 and 6.4 list the AHB master, slave and other signals respectively and their function. Each signal name is prefixed with H to denote that this is an AHB signal.

Masters

The AHB supports up to 16 bus masters. Each master wishing to initiate a transfer competes for a bus grant from the arbiter and has its control and address signals driven to the slave when it gets the bus.

If a master x does not wish to initiate a transfer it drives $HBUSREQ_x$ to low and if it owns the bus it also drives $HTRANS$ to IDLE. To initiate a transfer, it drives $HBUSREQ_x$ high. Upon getting bus ownership (checked via $HGRANT_x$, $HMASTER$ and $HREADY$), the address and control signals are driven onto the bus for exactly one cycle. To initiate the transfer, the master drives $HTRANS$ to NSQ (which abbreviates "non-sequential"). It also drives $HBURST$ to low indicating a single transfer, or to high indicating a four-beat burst. All this happens during the one-cycle address phase.

In the next cycle, the master drives the data on to the data buses (or samples it in case of a read). If this is a burst, then the master also continues to drive the control signals and increment the address signals to prepare for the next beat of the burst. In the middle of a burst $HTRANS$ is driven to SEQ.

Table 6.3: AMBA AHB Slave Signals

Signal	Driver	Description
HRDATA[127-31:0]	Slave	The read data bus. Can be up to 128 bits wide.
HSPLIT_x[15:0]	Slave	This is used by a slave x to tell the arbiter which masters should be allowed to re-attempt a split transfer. Each bit corresponds to a single master.
HREADY	Slave	When high indicates that a transfer is complete. Slaves can drive this low to insert wait states.
HRESP[1:0]	Slave	Allows the slave to provide additional information about a transfer. The reponses are okay (OK), error (ERR), retry (RETRY) and split (SPLIT).

Table 6.4: AMBA AHB System, Arbiter and Decoder Signals

Signal	Driver	Description
HCLK	Clock	The bus clock. The rising edge is used to trigger all AHB signals.
HRESET	System	The reset signal. Resets the bus to the default state. It is the only signal that is active low.
HGRANT_x	Arbiter	When high indicates that master x currently has the highest priority for getting the bus. Bus ownership does not actually change till the current transfer ends.
HMASTER[3:0]	Arbiter	Indicates which master is currently performing a transfer (and thus has the bus). Its timing is aligned with the address phase of the transfer. Used by SPLIT-enabled slaves.
HSEL_x	Decoder	This signal indicates that slave x is selected for the current transfer. There is a unique line for each slave on the bus. This signal is arrived at by decoding the higher order bits of the address bus.

It should be noted that in the last beat of a burst (or the one-cycle data phase of a single transfer) the information on the control and address buses is driven by the master that next has control of the bus, or by a *default master* (usually the highest priority master) if no master wishes to acquire the bus. In the latter case, the default master can simply drive $HTRANS$ to IDLE in which case the other signals are ignored. We will assume that Master 1 is the default master. Master 0 is reserved as a *dummy master* which guarantees to generate only IDLE transfers, and is granted the bus if all other masters are waiting on SPLIT transactions.

Responses to Slave Signals Masters need to respond to the following slave signals:

- If the slave drives $HREADY$ to low, then the master must continue the assert the same control, address and data signals in the next cycle, and continue this until $HREADY$ is high again.

- If the slave drives $HRESP$ to ERR, the master may abort the transfer or continue with it.

- If the slave drives $HRESP$ to SPLIT, the arbiter will grant the bus to another master. In this case the first master waits until it is given the bus again. The bus protocol only allows for masters to have one outstanding SPLIT transfer. Thus upon regaining the bus the master can continue with the transfer as before. However, a slave need not remember the control and address information and the master should broadcast this information first before driving/sampling the data buses.

- If the slave drives $HRESP$ to RETRY, the master simply retries the transfer until it is completed, which is indicated by the slave signalling OK. To prevent deadlock, only one master can access a slave that has issued the RETRY reponse.

Multiplexor

The bus uses a central multiplexor interconnect scheme. All masters drive their address and control signals and the arbiter decides which master's signals are routed on to the slaves.

Arbiter

The arbiter uses some arbitration algorithm (e.g. round-robin scheduling; AMBA does not specify or recommend any particular algorithm) to decide which master to grant the bus to. Actual bus ownership is not handed over until the current transfer completes.

Additionally, the arbiter is responsible for keeping track of masters (by internally masking their bus requests) that have SPLIT transfers outstanding and granting the bus to the highest priority one when the corresponding slave signals (via $HSPLIT_x$) that is it ready to continue the transfer.

Decoder

The decoder simply performs a direct decode of the address bus. The appropriate higher order bits give the value of $HSEL_x$ and the rest are used by slaves to determine source/target registers.

Slaves

Once a transfer begins it is up to the slave to determine how it proceeds. The slave can do one of the following:

- If all is well, the slave responds by driving $HREADY$ to high and $HRESP$ to OK, and the transfer is straightforward.
- If the slave needs a little time during the data phase, it can extend the phase by inserting wait states by driving $HREADY$ to low and $HRESP$ to OK. Note that the address phase cannot be extended.
- If the slave cannot complete the transfer immediately it can issue a SPLIT response if it is SPLIT-capable. SPLIT-capable slaves need to be able to record the numbers of up to 16 masters to prevent deadlock. When ready, they activate the appropriate bits on $HSPLIT_x$ to indicate which master(s) the slave is ready to communicate with and continue with the transfers.
- If a non-SPLIT-capable slave cannot complete a transfer immediately it drives $HRESP$ to RETRY. To prevent deadlock, it must record the number of the current master and ensure that an ensuing transfer is with the same master, until the RETRY'd transfer is complete. If the master is not the same, the slave has the option of issuing an ERR, generating a system level interrupt or a complete reset.
- In case of a complete failure, the slave drives $HRESP$ to ERR, and ignores the rest of the transfer.

The RETRY, SPLIT and ERROR responses take two cycles (HREADY is low in the first cycle, high in the second), to give the master time to re-drive the address and control signals onto the bus.

6.3.2 Implementation

We implement the AHB by following the specification in a straightforward manner without any optimizations. We need to implement the model as a state machine M_{AHB}, representing a state of M_{AHB} by $\bar{s}_{AHB}$.

Definition 6.9

$$\begin{aligned}\bar{s}_{AHB} \;=\; & (HTRANS[1:0], HREADY, HRESP[1:0], HSPLIT_x[15:0],\\ & HGRANT_x, HBUSREQ_x, HSEL_x, HADDR[31:0],\\ & HMASTER[1:0], HBURST, HWS_x, BB_x, HMASK_x,\\ & HSLVSPLIT_x)\end{aligned}$$

We write $\bar{s}'_{AHB}$ to represent $\bar{s}_{AHB}$ with all components primed.

The HWS_x, BB_x, $HMASK_x$ and $HSLVSPLIT_x$ signals are not part of the specification but are required by the implementation to count elapsed wait states and burst beats, and for the arbiter and slaves' internal bookkeeping. We shall refer to HWS_x and BB_x as counters.

Assumptions and Limitations

All assumptions made in §6.2.2 hold. We have made some additional assumptions to simplify the implementation a little.

Most importantly, we have not implemented the datapath. Datapath implementation and verification has already been demonstrated in §6.2 and verifying datapath properties for AHB is presently beyond the capabilities of our under-development model checker. The interesting aspects of the AHB all lie in the control circuitry. Other assumptions are:

- All bursts are four beats long. This encompasses all possible interactions that would be added by considering longer bursts.
- All bursts align at word boundries. Having non-aligned data does not affect the logical behaviour of the system but would increase the time to implement a working model. In fact, we restrict transfer size to be of word length.
- Slaves can insert up to four wait states. The specification leaves the actual number up to the implementer, but recommends no more than 16.
- We implement only three masters and two slaves. Again, this is the minimum number that encompasses all possible interactions and was considered sufficient for the purposes of this case study. With no datapath, the current system should scale up to the maximum easily without increasing the difficulty of model checking.

We have also not implemented some aspects of the specification (these and any signals they use have been left out of §6.3.1):

- Protection mechanisms are left out. These are given as optional in the specification.
- Locked bus access is left out.

The Model

As before, M_{AHB} is then described by an initial states predicate S_{0AHB} on states, and a transition predicate R_{AHB}. Due to the added complexity in the AHB, we define initial and transition predicates for the arbiter, decoder, multiplexor, masters, slaves and counters separately and take their conjunction to give the predicates for the system as a whole.

To improve readibility, we first define some predicates that abbreviate commonly used signal combinations:

Definition 6.10 *Abbreviations:*

- *Transfer types*

1. *Idle:* $IDLE(HTRANS[1:0]) = \neg HTRANS_0 \wedge \neg HTRANS_1$
2. *Busy:* $BUSY(HTRANS[1:0]) = HTRANS_0 \wedge \neg HTRANS_1$
3. *Non-seq:* $NSQ(HTRANS[1:0]) = \neg HTRANS_0 \wedge HTRANS_1$
4. *Sequential:* $SEQ(HTRANS[1:0]) = HTRANS_0 \wedge HTRANS_1$

- *Slave reponses*

1. *Okay:* $OK(HRESP[1:0]) = \neg HRESP_0 \wedge \neg HRESP_1$
2. *Error:* $ERR(HRESP[1:0]) = HRESP_0 \wedge \neg HRESP_1$
3. *Retry:* $RETRY(HRESP[1:0]) = \neg HRESP_0 \wedge HRESP_1$
4. *Split:* $SPLIT(HRESP[1:0]) = HRESP_0 \wedge HRESP_1$

- *Burst types*

1. *Single transfer:* $SINGLE(HBURST) = \neg HBURST$
2. *4-beat incrementing burst:* $INC4(HBURST) = HBURST$

To further avoid clutter, we will elide the arguments to the abbreviation predicates. Thus $IDLE$ stands for $IDLE(HTRANS[1:0])$. Of course this elision is not carried out in the theorem prover itself since that would cause a typing error. Also, we will prime the abbreviation name to denote the priming of the signals it is defined over.

We now define the initial state predicates:

Definition 6.11 *Initial state predicates are defined as follows :*

- *Arbiter*

$$S_{0_{AHB}}^{arbiter}(\bar{s}_{AHB}) = \bigwedge_{x \neq 1} \neg HGRANT_x \wedge HGRANT_1 \wedge HMASTER = 1$$

- *Decoder*

$$S_{0_{AHB}}^{decoder}(\bar{s}_{AHB}) = \bigwedge_{x} \neg HSEL_x$$

- *Counters*

$$S_{0_{AHB}}^{counters}(\bar{s}_{AHB}) = \bigwedge_{x} \neg HWS_x \wedge \bigwedge_{x} \neg BB_x$$

- *Masters*

$$S_{0_{AHB}}^{master}(\bar{s}_{AHB}) = IDLE \wedge \bigwedge_{x \neq 1} \neg HBUSREQ_x \wedge HBUSREQ_1$$

- *Slaves*

$$S_{0_{AHB}}^{slave}(\bar{s}_{AHB}) = HREADY \wedge OKAY$$

These defaults are those recommended by the specification document [8]. The notational abuse $HMASTER = 1$ above simply means that the bits of $HMASTER$ are set to the binary representation of 1, under the given endianness.

The system initial state predicate is simply the conjunction.

Definition 6.12

$$\begin{aligned} S_{0AHB}(\bar{s}_{AHB}) \;=\; & S_{0AHB}^{counters}(\bar{s}_{AHB}) \wedge S_{0AHB}^{arbiter}(\bar{s}_{AHB}) \\ & \wedge S_{0AHB}^{decoder}(\bar{s}_{AHB}) \wedge S_{0AHB}^{master}(\bar{s}_{AHB}) \\ & \wedge S_{0AHB}^{slave}(\bar{s}_{AHB}) \end{aligned}$$

The transition predicates are more complicated:

Definition 6.13 *Arbiter transitions:*

$$\begin{aligned} & R_{AHB}^{arbiter}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\ & (HGRANT'_0 \iff (HMASK_0 \wedge HMASK_1)) \wedge \\ & (HGRANT'_1 \iff (\text{if } HMASK_0 \text{ then } \mathtt{F} \text{ else } HBUSREQ_1) \vee \\ & \qquad \neg(\text{if } HMASK_1 \text{ then } \mathtt{F} \text{ else } HBUSREQ_2)) \wedge \\ & (HGRANT'_2 \iff \neg(\text{if } HMASK_0 \text{ then } \mathtt{F} \text{ else } HBUSREQ_1) \wedge \\ & \qquad (\text{if } HMASK_1 \text{ then } \mathtt{F} \text{ else } HBUSREQ_2)) \wedge \\ & (HMASTER' \iff \text{if } \neg HREADY \text{ then } HMASTER \text{ else } \neg HGRANT_1 \\ & (HMASK'_x \iff \text{if } SPLIT \wedge \neg HREADY \wedge (HMASTER = x) \text{ then } \mathtt{T} \\ & \qquad \text{else if } HSPLIT_x \text{ then } \mathtt{F} \text{ else } HMASK_x) \end{aligned}$$

Recall we are implementing three masters only. The dummy master 0 gets granted if and only if both the other masters are waiting on split transfers. Master 1 which is the default master gets priority over Master 2, but bus requests are masked for masters waiting on split transfers. A grant by itself does not give bus ownership. This happens when $HREADY$ is high. $HMASTER$ then indicates who has the bus, according to the value of $HGRANT_x$.

Definition 6.14 *Decoder transitions:*

$$\begin{aligned} & R_{AHB}^{decoder}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\ & (HSEL'_0 \iff \text{if } HREADY \text{ then } \neg HADDR_0 \text{ else } HSEL_0) \wedge \\ & (HSEL'_1 \iff \text{if } HREADY \text{ then } HADDR_0 \text{ else } HSEL_1) \end{aligned}$$

The decoder simply does a combinatorial decode of the higher order bits of the address bus. Since we have only two slaves and no datapath, a single-bit address bus suffices.

Definition 6.15 *Counter transitions:*

$$\begin{aligned} & R_{AHB}^{counter}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\ & (HWS'_0 \iff \neg HREADY) \wedge \\ & (HWS'_1 \iff \neg HREADY \wedge HWS_0) \wedge \\ & (HWS'_2 \iff \neg HREADY \wedge HWS_1) \wedge \\ & (bb0' \iff HREADY \wedge NSQ) \wedge \\ & (bb1' \iff \neg BB_2 \wedge SEQ \wedge \text{if} HREADY \wedge \neg BUSY \text{ then } bb_0 \text{ else } bb_1) \wedge \\ & (bb2' \iff \neg BB_2 \wedge SEQ \wedge \text{if} HREADY \wedge \neg BUSY \text{ then } bb_1 \text{ else } bb_2) \end{aligned}$$

The wait state counter simply counts up to 3 if $HREADY$ is low, resetting if $HREADY$ goes high. The burst counters count four beats when a burst starts (signalled by a NSQ followed by a SEQ transfer type). This is used by the arbiter to determine when it is safe to hand the bus over to another master (recall that in our implementation bursts may not be interrupted).

Definition 6.16 *Multiplexor transitions:*

$$
\begin{array}{l}
R^{mux}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(HTRANS' \iff \text{if } \neg HREADY \text{ then } HTRANS \\
\qquad\qquad\qquad\quad \text{else if } HGRANT_1 \text{ then } HTRANS_{m_1} \\
\qquad\qquad\qquad\quad \text{else if } HGRANT_2 \text{ then } HTRANS_{m_2} \\
\qquad\qquad\qquad\quad \text{else } HTRANS_{m_0}) \wedge \\
(HBURST' \iff \text{if } \neg HREADY \text{ then } HBURST \\
\qquad\qquad\qquad\quad \text{else if } HGRANT_1 \text{ then } HBURST_{m_1} \\
\qquad\qquad\qquad\quad \text{else if } HGRANT_2 \text{ then } HBURST_{m_2} \\
\qquad\qquad\qquad\quad \text{else } HBURST_{m_0})
\end{array}
$$

Drives the selected master control signals (i.e. $HTRANS_{m_x}$ etc) to the bus.

Definition 6.17 *Master transitions:*

$$
\begin{array}{l}
R^{master_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(\neg(OK \wedge HREADY) \wedge HGRANT_x \Rightarrow HBUSREQ'_x) \wedge \\
(HREADY \wedge IDLE \wedge OK \wedge HGRANT_x \Rightarrow NSQ') \wedge \\
(NSQ \wedge OK \wedge INC4 \wedge HGRANT_x \Rightarrow (BUSY' \vee SEQ') \wedge INC4') \wedge \\
(SEQ \wedge \neg BB_2 \wedge OK \wedge HGRANT_x \Rightarrow (BUSY' \vee SEQ') \wedge INC4') \wedge \\
(BUSY \wedge \neg BB_2 \wedge OK \wedge HGRANT_x \Rightarrow SEQ' \wedge INC4') \wedge \\
(\neg HREADY \wedge RETRY \wedge HGRANT_1 \Rightarrow IDLE') \wedge \\
(HREADY \wedge RETRY \wedge HGRANT_1 \Rightarrow NSQ') \wedge \\
(ERROR \wedge HGRANT_1 \Rightarrow IDLE')
\end{array}
$$

A line-by-line explanation of this transition relation follows: if master has bus ownership it will continue to request it until the transfer completes, otherwise the arbiter may think the master no longer requires the bus in the middle of a transfer; starting a transfer by asserting NSQ; switching to the SEQ transfer signal if transfer is a burst, i.e. INC4 is being asserted; continuing to assert SEQ or BUSY as burst takes place; forcing a SEQ assert if BUSY was asserted previously (the specification does not mention this but it is clearly required to prevent an infinite sequence of BUSYs, i.e. a livelock); response to first and second cycle of retry; response to first and second cycle of error. According to the specification, the master can do whatever it likes if split, since it loses the bus.

The one exception to the above is Master 0, the dummy master. This simply generates IDLE no matter what happens, and never requests the bus.

Definition 6.18 *Slave transitions:*

$$
\begin{aligned}
&R_{AHB}^{slave_x}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
&(HSEL_x \wedge HWS_2 \Rightarrow HREADY') \wedge \\
&(HSEL_x \wedge (NSQ \vee SEQ) \wedge OK \Rightarrow OK' \wedge HREADY') \wedge \\
&(HSEL_x \wedge IDLE \Rightarrow HREADY' \wedge OK') \wedge \\
&(HSEL_x \wedge BUSY \Rightarrow OK' \\
&(HSEL_x \wedge \neg HREADY \wedge \neg(IDLE \vee BUSY) \wedge RETRY \\
&\qquad\qquad \Rightarrow HREADY' \wedge RETRY') \wedge \\
&(HSEL_x \wedge \neg HREADY \wedge \neg(IDLE \vee BUSY) \wedge ERROR \\
&\qquad\qquad \Rightarrow HREADY' \wedge ERROR') \wedge \\
&(HSEL_x \Rightarrow \neg SPLIT')
\end{aligned}
$$

A line-by-line explanation: this line together with the wait state counter ensures that $HREADY$ never stays low for more than four consecutive cycles, enforcing the rule that slaves may not insert more than four wait states; signal end of transfer by asserting $HREADY$ and OK; reponse to IDLE signal is $HREADY$ and OK; response to BUSY signal is OK; drive second cycle of RETRY; drive second cycle of ERROR; do not ever signal SPLIT.

A SPLIT-capable slave is slightly more complex. To add SPLIT ability, we conjoin the above transition relation (excepting the last line) with the following.

$$
\begin{aligned}
&(HSLVSPLIT'_x \iff \\
&\qquad \text{if}\,(HSEL_x \wedge \neg HREADY \wedge SPLIT \wedge (HMASTER = x)) \\
&\qquad \text{then}\, HMASTER_x \,\text{else}\, HSLVSPLIT_x) \wedge \\
&(HSEL_x \wedge HGRANT_0 \Rightarrow OK) \wedge \\
&(HSLVSPLIT_x \wedge (y \neq x) \Rightarrow \neg HSLVSPLIT_y)
\end{aligned}
$$

The first conjunct is for recording the current bus master's number so when the slave is later ready it can assert the appropriate $HSPLIT_x$ line. We abstract as much of the behaviour as possible, but the next two conjuncts are required to prevent undesirable behaviour. The first disables splits if the dummy master has the bus, and the last ensures that the slave does not split if it has already done so. The specification recommends that slaves should be able to split on as many masters as are present. However, this simplification does not affect logical behaviour, only efficiency.

The conjunction gives the system transition relation:

Definition 6.19

$$
\begin{aligned}
R_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \;=\; & R_{AHB}^{counters}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R_{AHB}^{arbiter}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \\
& \wedge R_{AHB}^{decoder}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R_{AHB}^{master_x}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \\
& \wedge R_{AHB}^{slave_x}(\bar{s}_{AHB}, \bar{s}'_{AHB})
\end{aligned}
$$

6.3.3 Verification

We verify arbitration, latency and deadlock freedom properties for AHB. As there is no datapath we do not verify coherence. The BDD variable ordering used was an interleaving of the current and next-state variables, which was then reordered after a manual dependency analysis.

Arbitration

The first properties we verify relate to arbitration. Typically such properties confirm that the arbiter is fair in some sense. The first property we verify is mutual exclusion, i.e. two masters never simultaneously get granted. The CTL property for this is

$$\mathbf{AG}(HGRANT_x \wedge (x \neq y) \Rightarrow \neg HGRANT_y)$$

The required theorem is given by running the model checker.

Theorem 6.20

$$\vdash \forall \bar{s}_{AHB}. \\ \bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}(HGRANT_x \wedge (x \neq y) \Rightarrow \neg HGRANT_y)$$

Our implementation is a simple priority based one and is obviously not meant to be fair in the sense that all requests are ultimately granted. This should hold true for the highest priority Master 1 however. This can be checked using the CTL property

$$\mathbf{AG}(HBUSREQ_1 \wedge \neg HMASK_1 \Rightarrow \mathbf{AF} HGRANT_1)$$

Note that a grant is not the same as getting bus ownership (Master 1 may de-assert its request while waiting for the bus). Thus this property holds and the model checker gives the required theorem.

Theorem 6.21

$$\vdash \forall \bar{s}_{AHB}. \\ \bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}(HBUSREQ_1 \wedge \neg HMASK_1 \Rightarrow \mathbf{AF} HGRANT_1)$$

For other masters, the best we can hope for is that the possibility of a grant exists, as given by the CTL property schema

$$\mathbf{AG}(HBUSREQ_x \wedge \neg HMASK_x \Rightarrow \mathbf{EF} HGRANT_x)$$

and the model checker confirms that this is so.

Theorem 6.22

$$\vdash \forall \bar{s}_{AHB}. \\ \bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}(HBUSREQ_x \wedge \neg HMASK_x \Rightarrow \mathbf{EF} HGRANT_x)$$

Latency

Latency checking for the AHB is more complicated than for the APB, as the presence of bursts, busy signals and wait states means that the transfer times are variable.

First, we do a quick sanity check to confirm that all transfers do indeed end, as given by this CTL property:

$$\mathbf{AG}(NSQ \Rightarrow \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK) \vee RETRY \vee ERROR \vee SPLIT])$$

and this is easily checked:

Theorem 6.23

$$\vdash \forall \bar{s}_{AHB}.\ \bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}(NSQ \Rightarrow \mathbf{AX}\,\mathbf{A}[\neg NSQ\mathbf{U}(HREADY \wedge OK) \vee RETRY \vee ERROR \vee SPLIT])$$

Since we have limits on the length of bursts, the number of consecutive busy signals and the number of consecutive wait states, we should be able to confirm that a transfer will take at most a given number of cycles. This number is in fact ten cycles (1 address phase cycle + 4 burst cycles + 4 wait states + 1 BUSY signal) in the case of our implementation so far. The CTL property saying this is more neatly expressed if we first define a function LAT:

$$\begin{array}{ll} LAT & f\,0 = f \\ LAT & f\,(n+1) = f \vee \mathbf{AX}(LAT\,f\,n) \end{array}$$

This expresses in CTL a latency of at most n cycles until the event described by f holds. The required property is then given by the following CTL property:

$$\begin{array}{ll} \mathbf{AG} & ((NSQ \wedge SINGLE \Rightarrow LAT\,(HREADY \wedge OK)\,2) \wedge \\ & (NSQ \wedge INC4 \Rightarrow LAT\,((HREADY \wedge OK) \vee RETRY \vee ERROR \vee SPLIT)\,10 \wedge \\ & \qquad \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK) \vee RETRY \vee ERROR \vee SPLIT])) \end{array}$$

noting that a single transfer takes only two cycles and that a burst, if not interrupted, must finish within ten cycles. An unfolding of LAT would reveal several relational product computations, which are time and space consuming. We can make our task easier by using the following lemma derived from the CTL semantics.

Lemma 6.24

$$\vdash \forall f\,g\,M\,s.s \vDash_M \mathbf{AG}(f \wedge g) \iff s \vDash_M \mathbf{AG}f \wedge s \vDash_M \mathbf{AG}g$$

Proof Simple rewriting with Definition 4.4 and Proposition 4.5. □

We can thus split[2] the latency property above into the two conjuncts

$$\mathbf{AG}\quad (NSQ \wedge SINGLE \Rightarrow LAT\,(HREADY \wedge OK)\,2) \tag{6.6}$$

and

$$\begin{aligned}\mathbf{AG}\quad (NSQ \wedge INC4 \Rightarrow LAT\,((&HREADY \wedge OK)\\ &\vee RETRY \vee ERROR \vee SPLIT)\,10 \wedge\\ &\mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK)\\ &\vee RETRY \vee ERROR \vee SPLIT])\end{aligned} \tag{6.7}$$

We then observe that the propositional fragment of L_μ has all the properties of normal propostional logic. In particular, we have

Lemma 6.25

$$\begin{aligned}&\forall M\,e\,s\,f_1\,f_2\,f_3\,f_4.\\ &s \vDash^e_M f_1 \wedge f_2 \Rightarrow f_3 \wedge f_4 \iff (f_1 \wedge f_2 \Rightarrow f_3) \wedge (f_1 \wedge f_2 \Rightarrow f_4)\end{aligned}$$

and

Lemma 6.26

$$\forall M\,e\,s\,f_1\,f_2\,f_3.s \vDash^e_M (f_1 \Rightarrow f_2) \Rightarrow f_1 \wedge f_3 \Rightarrow f_2$$

proved easily by the HOL simplifier given the satisfiability theorems from Table 2.2.

Using Lemma 6.25 together with Theorem 4.10 we can further split conjunct 6.7 above into

$$\begin{aligned}\mathbf{AG}\quad (NSQ \wedge INC4 \Rightarrow LAT\,((&HREADY \wedge OK)\\ &\vee RETRY \vee ERROR \vee SPLIT)\,10)\end{aligned} \tag{6.8}$$

and

$$\begin{aligned}\mathbf{AG}\quad (NSQ \wedge INC4 \Rightarrow \mathbf{AXA}[&\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK)\\ &\vee RETRY \vee ERROR \vee SPLIT])\end{aligned} \tag{6.9}$$

Now the satisfiability theorem for conjunct 6.9 follows from Theorem 6.23 using Lemma 6.26 and Theorem 4.10. The satisfiability theorems for conjuncts 6.6 and 6.8 are derived by model checking. All three resulting theorems can then be recombined in HOL using lemmas 6.24 and 6.25 to give the required theorem.

[2]Technically of course, we are not splitting the formula but the statement of its satisfiability. We elide these details to avoid clutter.

Theorem 6.27

$$\begin{aligned}\vdash\ & \forall \bar{s}_{AHB}.\\ & \bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}((NSQ \wedge SINGLE\\ & \qquad \Rightarrow LAT\,(HREADY \wedge OK)\,2) \wedge\\ & \quad (NSQ \wedge INC4\\ & \qquad \Rightarrow LAT\,((HREADY \wedge OK)\\ & \qquad\quad \vee RETRY \vee ERROR \vee SPLIT)\,10 \wedge\\ & \qquad \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK)\\ & \qquad\quad \vee RETRY \vee ERROR \vee SPLIT]))\end{aligned}$$

Lemma 6.24 could also have been used in the derivation of Theorem 6.7 but in that case not much is gained by doing so as neither conjunct's evaluation results in large BDDs.

Deadlock Freedom

The transition relation for the AHB is not obviously total, unlike that for the APB. Thus the obvious way of checking for deadlock is the CTL property

$$\mathbf{AG}\,\mathbf{EX}\,True$$

Since CTL model checking requires the transition relation to be totalised (see Definition 4.1), this property check needs to be carried out before totalisation. But then we cannot check for the CTL property.

Fortunately, due the fine-grained nature of our integration, we are not reliant on just getting a true/false answer from the model checker. We can simply "check" the property

$$\mathbf{EX}\,True$$

whose semantics are not affected by a non-totalised transition relation (only fix-point computations are affected), and then separately check whether the set of states returned by the model checker for the above property contains the set of reachable states $Reachable\,R_{AHB}\,S_{0AHB}$ of the system (§4.4.1). Thus we have the theorem

Theorem 6.28

$$\vdash Reachable\,R_{AHB}\,S_{0AHB} \subseteq \{s | s \vDash_{M_{AHB}} \mathbf{EX}\,True\}$$

which tells us that all reachable states have a next state and thus the system cannot deadlock. Subset inclusion here is modelled by propositional implication between the characteristic functions of the sets. The functions are boolean, so symbolic model checking can be used.

This property does not uncover situations where even though the transition system does not deadlock, it ends up in a useless loop doing nothing. To some extent, this is a liveness property and beyond the expressive power of CTL. We are considering how to best address this problem, either by writing L_μ properties or by finding a halfway solution that can be expressed in CTL.

6.4 Verifying AMBA

So far, we have separately checked correctness properties for the AHB and APB components of AMBA. Ideally, since the signals of the AHB and the APB do not overlap, these properties hold in the combined system, in which the APB is connected via a bridge to the AHB. However, conjoining R_{AHB} and R_{APB} will result in a large system which may be infeasible or time consuming to model check directly. We can instead construct a compositional proof in the theorem prover.

The first task is to define the bridge. This is the APB master that acts as a slave to the AHB. We first define the states over which the bridge would operate.

Definition 6.29

$$\bar{s}_{bridge} = \bar{s}_{AHB} \times \bar{s}_{APB}$$

and as before we write $\bar{s}'_{bridge}$ do denote the "next" state. The bridge transition relation R_{bridge} follows from this.

Definition 6.30

$$R_{bridge}(\bar{s}_{bridge}, \bar{s}'_{bridge}) = R^{slave_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R^{master}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB})$$

Now we can define a new transition relation for the APB with R_{bridge} as the master. We shall call this transition relation R_{APB2}.

Definition 6.31

$$\begin{aligned} &R_{APB2}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\ &(\exists \bar{s}_{AHB} \bar{s}'_{AHB}. R_{bridge}(\bar{s}_{bridge}, \bar{s}'_{bridge})) \wedge R^{slave}_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) \end{aligned}$$

As in §5.2, we use existential abstraction to hide behaviours we wish to ignore. This allows us to show that the new transition relation preserves all behaviours.

Lemma 6.32

$$\vdash R_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = R_{APB2}(\bar{s}_{APB}, \bar{s}'_{APB})$$

Proof In the $\Rightarrow$ direction we need to furnish the appropriate witnesses for the existentially quantified variables. This is done by using the integrated SAT solver in HOL to find a satisfying assignment for $R^{slave_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB})$. We know that such an assignment exists from Theorem 6.28, since the only way there is no satisfying assignment is if there are no transitions in the system. The rest follows by simplification. The $\Leftarrow$ direction is straightforward. $\Box$.

Using Lemma 6.32, it is trivial to show that the properties proved in the model M_{APB} with transition relation R_{APB} also hold in the model M_{APB2} with transition relation R_{APB2}.

Theorem 6.33

$$\vdash \forall f. \bar{s}_{APB} \vDash_{M_{APB}} f \Rightarrow \bar{s}_{APB} \vDash_{M_{APB2}} f$$

We can similarly define R_{AHB2} in which we can replace one of the generic slaves with R_{bridge}, this time hiding the APB signals, and conclude that all properties proved for the AHB hold when one of the slaves is the APB master.

At a more general level, we can show, without any extra model checking, that properties proved for for AHB and APB hold in the combined system. First we need a technical lemma.

Lemma 6.34 *If any M_1 and M_2 are the same except that $M_1.AP \subseteq M_2.AP$, then*

$$\forall f\, s_1\, s_2.\, s_1 \vDash_{M_1} f \iff s_2 \vDash_{M_2} f$$

This just states that adding extra unused propositions to a model does not change its behaviour. Note that the underlying state type of the two models is different and thus trivial amendments have to made to M_2 to satisfy the type checker. The main result then states that properties proved for a sub-system can be shown to be true of the entire system, provided certain conditions hold.

Theorem 6.35 *For any universal property f and models M_1 and M_2,*

$$\forall s.\, s \vDash_{M_1} f \Rightarrow s \vDash_{M_2} f$$

provided every behaviour of M_1 is a behaviour in M_2.

Note that Theorem 6.35 requires both models to have the same state type. This is where Lemma 6.34 is used (to add the extra propostions of the system M_2 to the sub-system M_1).

We can now define the full AMBA model M_{AMBA} by defining

$$R_{AMBA} = R_{AHB2} \wedge R_{APB2}$$

and defining the rest of the M_{AMBA} tuple in the usual manner. Then, for example, we can take M_{APB} as M_1 and M_{AMBA} as M_2, and use Theorem 6.33 and Theorem 6.35 to show that all universal APB properties hold in the AMBA system. And similarly for the AHB. We have thus proved, without using the model checker, that all universal properties proved for AHB and APB separately also hold in the combined system. This result does not apply to the non-universal deadlock freedom properties; deadlock freedom in a sub-system does not imply deadlock freedom overall.

Though we used interactive theorem proving, the general technique can be applied in any similar situation and it is possible to envision writing proof script generation functions in ML that would automate much of the task.

6.5 Related Work

Two recent verifications targeting AMBA AHB were presented in 2003. The first work [164] uses the ACL2 theorem prover to prove arbitration and coherence properties for the bus. Time is abstracted away and intra-transfer complications (such as bursts, wait states, splits and retries) are ignored. This makes sense as theorem provers are better

suited for attacking datapath properties at a high level of abstraction, without the clutter of cycle-level control signals.

The second work uses the SMV model checker to fix bugs in an academic implementation of AMBA AHB [158]. They concentrate on a no-starvation violation (a master is denied access to the bus forever) which however is caused by an error in the implementation of their arbiter rather than in the protocol itself. The error is very subtle however and we concur with their conclusion that this particular case should be highlighted in the FAQ if not in the specification.

More recently, work is in progress on porting a Z specification of AMBA AHB [143] to HOL. This work is still in the draft stage. A recent Ph.D. thesis [181] verifies roughly the same set of AHB properties as ours (it also verifies the datapath) for a more complex implementation using the CADENCE SMV model checker and imports the results in HOL as trusted theorems. The emphasis here is on using specialist tools as oracles for HOL and the verification process itself is not discussed at length. The almost complete lack of interaction between control and data in bus designs makes it relatively easy to do the kind of abstractions that model checkers are good at. Bus architectures and the somewhat related domain of cache coherence protocols have thus long been staples of model checking case studies [30, 35, 68, 128].

6.6 Conclusion

The AMBA AHB and APB specification is a 110 page document, laying out the design in the usual mix of english, timing diagrams and interface diagrams, supplemented by a FAQ. We have developed a formal HOL version of the AHB and APB components at the cycle-level and model-checked useful properties. We have then used HOL to compose the two verifications.

However, while the case study is a useful show-case for our framework, there is much to be done for a complete verification. Priorities are verifying datapath properties for the AHB, implementing locked access, having a more sophisticated arbitration policy and non-word-aligned transfers.

During the case study it became clear that most of the time in a model checking oriented verification is spent patching failed properties or flawed models. Thus, the development of good failure analysis and debugging capabilites will go far in making the tool usable in practice.

The model checking runs were not particularly time or space intensive and all went through in a few minutes at most. We attribute this to our simplified model, the decomposition and abstraction we did, and our focus on control properties.

The case study illustrates how we can seamlessly combine theorem proving, model checking and SAT solvers to perform decomposition (e.g. Theorem 6.27 and Theorem 6.35) and abstraction (e.g. Theorem 6.33) for model checking. All steps are backed up by fully-expansive formal proof. We have thus enabled verifications that would be hard, if not infeasible, using only one of these technologies.

Chapter 7

Case study II: An ALU

A central concern of our work is to ensure that our approach does not create an unacceptable penalty in terms of the performance of the model checker, due to the additional theorem proving overhead. This case study complements the previous one in which the focus was on demonstrating the capabilities of the work, whereas this one focuses on performance.

This chapter describes results in evaluating the performance of the model checking algorithm for L_μ formalised in Chapter 2. We use the formalisation in Chapter 4 to use this model checker to check CTL properties for a three-stage pipelined arithmetic and logic unit (ALU) [29].

We chose a well-known but rather old (c.1990) model to check performance primarly because as far as the implementation of a user interface and standard optimisations is concerned, our symbolic model checker is for the moment about as powerful as state-of-the-art tools were around 1990, and is at present unable to attack systems with large state spaces. Pedagogical accessibility and the availability of a ready-made BDD variable ordering were also influencing factors.

The BDD method for testing boolean satisfiability is only of heuristic value: the problem is NP-complete. Using BDDs to represent state sets is similarly claimed to be efficient only in a practical sense. Thus a performance evaluation needs to demonstrate empirical results.

We compare the performance of our proof-driven model checker with a version of itself in which all the proof machinery is turned off. This gives us an idea of the performance penalty caused by the extra theorem proving overhead. We note that the HOL to BUDDY interface itself does not contribute significantly to this overhead [110].

We analyse only the execution times. Memory consumption is also an important performance measure for symbolic model checking. However this is not a factor in our case study because the memory requirement of BDDs created while verifying this model is already well understood and the additional HOL overhead is not an inhibiting factor given the relatively large amounts of RAM available on current systems.

7.1 The Test System

Our test circuit (Fig. 7.1) performs three-address logical operations on a register file (whose registers are denoted by $reg_0, reg_1, \ldots$). The pipeline has three stages:

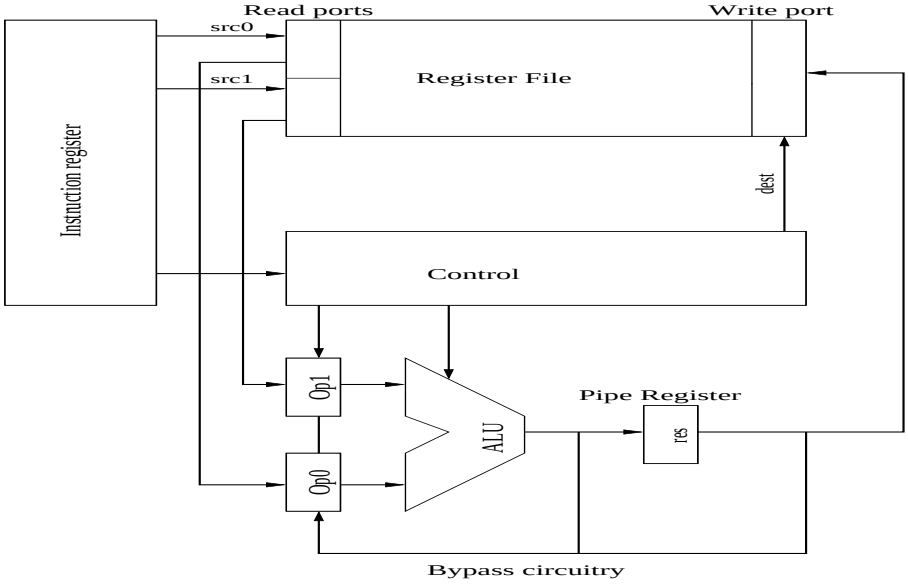

Figure 7.1: Simple Pipelined ALU

1. *Fetch*: The operands are read from the register file (the source registers being pointed to by the addresses *src0* and *src1*) into the operand registers *op0* and *op1*.

2. *Execute*: The ALU computes the result and writes it into the pipe register *res*.

3. *Write back*: The result is written back into the register file, at the location pointed to by *dest*.

There is a register bypass path, required for data forwarding. The circuit thus contains both data-path and control circuitry. Addition of extra pipe registers would result in as many new stages, each propagating the result down the pipeline. The number of registers, the number of instructions and the width of the data-path are variable. For simplicity, we fix the number of instructions to two (logical OR and NOR). For the timing measurements, we work with increasing values for the width of the data-path and the number of registers in the register file.

With these parameters, an instruction to the circuit has five components that form the inputs:

- A one-bit opcode, *ctrl*.
- n-bit addresses for the two source and one destination registers (*src0*, *src1* and *dest* respectively), giving 2^n addressable registers.
- A one-bit *stall* input. If this is true, signalling for example a cache miss, then a no-op is propagated down the pipeline.

In this simple circuit, we are concerned with verifying two properties at the RTL level. The first property is expressed by the CTL formula

$$\mathbf{AG}(\neg stall \Rightarrow ((aluop(src\_op0_i, src\_op1_i) = dest\_res_i)) \tag{7.1}$$

where *aluop* abbreviates a simple propositional formula to ensure that the correct operation is applied given the value of *ctrl*. The place holders *src_op0*, *src_op1* and *dest_res* abbreviate the source registers for the operands and for the destination register respectively, with the subscript encoding the bit. Thus, this specifies that the destination register is always updated correctly.

To express *src_op0*,*src_op1* and *dest_res* in CTL, we must factor in the latency of the pipeline. For example, for a given operation, the values in the source registers at the time the operation begins are *not* the values that are input to the operation. The values that are required are from the state of the register file after the previous instruction has finished, i.e. two clock cycles in the future. Similarly, the value required for the destination register is the value three cycles in the future.

The assumption here is that an instruction begun at time t will not affect the register file until time $t + 3$, i.e. three clock cycles in the future. To check that this assumption holds, we check that,

$$\mathbf{EX}^k reg_{j,i} \Leftrightarrow \mathbf{AX}^k reg_{j,i} \qquad 1 \leq k \leq 3 \tag{7.2}$$

where $\mathbf{EX}^k$ abbreviates k applications of $\mathbf{EX}$, and $reg_{j,i}$ is bit i of register j. This can also be done in the model checker.

After accounting for the latency in the pipeline, these abbreviations expand out as follows (for simplicity, we assume there are only two file registers):

$$src\_op0_i = (\neg src1 \wedge \mathbf{AX}(\mathbf{AX}(reg_{0,i}))) \vee (src1 \wedge \mathbf{AX}(\mathbf{AX}(reg_{1,i}))) \tag{7.3}$$

and similarly for *src_op1*. Similarly, *dest_res* expands to

$$\begin{aligned} dest\_res_i \;\; = \;\; & (\neg dest \wedge \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{0,i})))) \\ & \vee (dest \wedge \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{1,i})))). \end{aligned} \tag{7.4}$$

The second property of interest is that for each instruction, the register not being written to (which is all registers if the pipeline stalls) is not changed. So for example for register 1:

$$\mathbf{AG}((stall \vee \neg dest) \Rightarrow (\mathbf{AX}(\mathbf{AX}(reg_{1,i})) = \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{1,i}))))). \tag{7.5}$$

7.2 Benchmarks

We now compare the execution time of our model checker with one we wrote in plain ML that bypasses HOL and works directly with the BDD engine.

The same variable ordering was used for all BDDs. It is essentially the ordering given in [28] which generally yields good results: the source address registers are closest to the root, with their bits interleaved. Next we interleave the stall and destination address registers, for all three pipeline stages, starting with the fetch stage. These are followed by the opcode, followed by the interleaved bits of the operand, general and pipe registers (this time in big-Endian order).

Our proof-driven model checker's CPU time can be split into two phases: setting up the model (overhead time) and checking it. The overhead work needs to be done only once per model. When timing the fully-expansive model checker we calculate the amortised

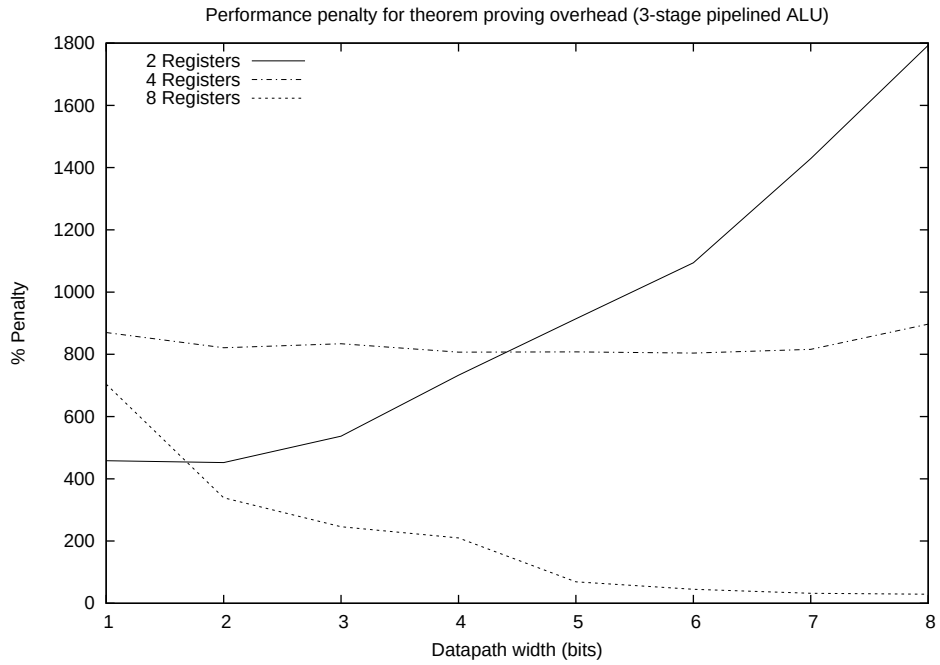

Figure 7.2: Relative benchmarks

checking time by spreading overhead time over the various properties checked for the same model.

The graph in Fig. 7.2 shows the performance penalty for using the proof-driven versus the plain model checker (the latter of course being faster due to the absence of theorem proving overhead).

Both programs were run for data-paths of one to eight bits, and address spaces of one to three bits (i.e. two to eight registers), giving over 10^{30} reachable states in the most complex case. An increase in either increases the branching complexity of the model, with increases in address space having a stronger effect. Although several properties can be checked for this ALU, the results shown are for the one which represents the worst (greatest) performance difference. Overhead costs can still be amortised over this because it is a template that needs to be separately checked for each bit on the data-path.

Due to the absence of standard optimisations, the system began to thrash with an address space of four bits. Nevertheless, the graph shows that the difference in performance closes as the branching complexity of the system increases: as the number of registers increase, the performance penalty begins to drop with increasing datapath width and becomes acceptably small for the larger examples. With two registers, the BDD operations are fast and the penalty is large (1800% in the worst case). With four registers, the penalty is still large but does not increase as the datapath width increases. With eight registers, the BDD operations are consuming a significant amount of resources and the penalty drops as datapath width increases.

This is because the most expensive BDD operation, the relational product, is hit particularly hard by any increase in branching complexity, whereas the corresponding operation on the term part of the term-BDD is trivial. This allows the theorem proving component of the program to "catch up" with the BDD component. In the most complex

situation, with an eight bit datapath and three bit address space, the performance penalty is about 30%.

7.3 Concluding Remarks

We have as yet not implemented any of the standard optimisations such as partitioning the transition relation, iterative squaring and others mentioned in Chapter 3. All these would speed up the BDD component and increase the performance difference. On the other hand, our test bed is a toy example and the expectation is that even with all these optimisations, the theorem proving component, which does not scale as badly as BDDs for larger examples, will catch up when the program is run for harder examples.

When is 30% an acceptable performance penalty? The question is subjective and there is no quantitative answer. In conversations with senior formal verification engineers at Intel Corporation and what used to be Compaq Corporation's Alpha division (now with Intel), we have received the general impression that industrial practitioners would not consider the advantages of a proof-driven implementation worthwhile if the system caused more than a 100% performance penalty compared to alternative solutions, under any circumstances. In many situations, particularly in model checking runs expected to terminate within twelve hours (so they could be run overnight), even a below 100% penalty may not be acceptable.

Getting encouraging results with a single class of example does not provide conclusive evidence about the performance penalty for proof-driven model checking. Further benchmarking is required, with other kinds of examples such as asynchronous circuits and with improved implementations of the model checker which use state space reductions and lower-level optimisations.

Chapter 8

Related work

At the end of each chapter we have briefly discussed research in technologies closely related to the work presented. The main contribution of this work is not a model checking or theorem proving technology however, but rather an approach to how the two might be integrated. In this chapter we discuss other approaches to combining model checking and theorem proving, this time emphasising the integration aspects. We give an overview, followed by a more detailed look at three systems that are closely related to our approach.

8.1 Overview

The work closest to ours in spirit is the VOSS system [166], in which a data-type of BDDs was added to a lazy functional language, to enable easy functional programming of BDD based algorithms. This system was later interfaced to HOL [98] to enable results from VOSS to be imported into HOL. This work is discussed in detail in §8.2.

8.1.1 Model Checkers as Oracles

There have been several somewhat similar attempts to use model checkers as oracles for theorem provers, with varying degrees of integration:

- In Kurshan et al [107], compositional reasoning is supported by splitting proofs into model checkable pieces (checked in COSPAN [106] and recombined in the TLP theorem prover (a proof environment for the Temporal Logic of Actions [109])). Integration is via input and output files.

- The first significant non-file-based integration of symbolic model checking with theorem proving is achieved in 1995 in Dingle et al [57] and PVS [154]. The latter is particularly relevant as it involves a formalisation of temporal logic inside the theorem prover to facilitate importing the result of model checking back into the theorem prover. The model checker itself is implemented as an atomic proof rule in the proof system of the theorem prover. This approach has been extended to include abstraction and abstraction refinement for model checking [77, 148, 161].

- At about the same time another project formalised a theory of I/O automata [119] in ISABELLE and constructed an environment for model checking specifications mod-

elled as I/O automata using the STEP system [19] and a model checker for L_μ [80] as external oracles. A comprehensive account can be found in Müller et al [137].

- In Schneider et al [165], theories of LTL and ω-automata (automata over infinite words) are formalised in HOL together with theorems to enable formal translation of LTL specifications to ω-automata. Infrastructure was provided to enable the use of the SMV model checker as an external oracle.

- A recent Ph.D. thesis by Susanto [181] uses CADENCE SMV and ACL2 for verifying components using model checking and symbolic execution respectively, and imports the results into HOL where they may be used as lemmas for more abstract theorems.

In the work described so far, the central theme has been to exploit the automation of model checking technology by constructing the main proof in a theorem prover and using model checkers as oracles (the model checker may use the theorem prover's decision procedures to construct its abstractions).

Our work differs from these in three aspects. First, they formalise only the theory (and sometimes not even that), whereas we formalise the executable algorithm as well. Second, because of the above, whereas they must trust the result of the oracle, we have a high assurance of soundness because more of the work is backed up by mechanised fully-expansive proof. Third, we have a fine-grained integration in the sense that the user has full access to both the theorem proving and model checking tools.

The price we pay for the extra proof and flexibility is in reduced performance and increased development effort. The former appears to be acceptable so far (see Chapter 7). The latter is hard to quantify but we feel that having to prove each step is offset by the relatively little time spent debugging the tool (thanks to the LCF-style architecture of HOL).

8.1.2 Theorem Provers as Organizers

Other systems take a more model checking oriented approach in which the primary task of the theorem prover is to decompose the properties being checked, and also do case-splitting and model transformation.

CADENCE SMV (or just SMV) [126, 127] implements lightweight theorem proving on top of a model checker, primarily to aid in abstraction and compositional reasoning. Recent work has focused more on instrumenting SAT solvers to help with symbolic model checking [129].

In CVC [180] and its successor CVC LITE [12], the focus is on a complete decision procedure for quantifier-free first order formulas in which atomic propositions range over decidable fragments of more expressive logics such as Presburger arithmetic, with decision procedures (including an integrated SAT solver) for decidable logics cooperating in a variant of the Nelson-Oppen style [142]. Although there is no direct connection with our work, we feel that our approach of instrumenting decision procedures with proof could be fruitfully extended to this kind of system.

A more diverse approach is taken in the STEP system [19]. Models can be input in an expressive language. The user can then hierarchically decompose the proof into subgoals that can be discharged using integrated symbolic and explicit-state model checkers or decision procedures. Alternatively, a subgoal can be considered in an integrated interactive

prover that supports most standard theorem proving machinery. This system is oriented towards verification of reactive systems rather than general purpose mechanised proof, and it is not fully expansive. Though it would be possible to replicate its abilities in our framework, it would involve considerable effort.

On a more general level, Berezin [14] describes a Gentzen-style calculus in which judgements carry the model as well as hypothesis and entailments. Unlike our similar but more specific calculus, the elements of the judgements can be instantiated with the user's choice of logic and models. The calculus is sound, so that as long as the user's implementations of the rules of inference of the calculus are sound, the result is guaranteed correct. Of course, this leaves soundness entirely up to the implementer. But the generality of the idea is appealing and we hope at some point to create a similar framework for our own work.

8.1.3 Verifying the Model Checker

Our model checker's results are correct by construction, since both the theory and the implementation are proved correct in HOL. The only danger is that of a bug in HOL, but the probability of this is very low, as explained in Appendix A.

Other efforts have also been made to verify model checkers. The first approach is to formalise and verify the theoretical justification for some model checking algorithm. This has been done for partial order reductions using HOL [31]. The authors cite the "not uncommon" occurrence of incorrect proofs in model checking literature and the relatively better soundness of HOL as motivating factors. They do not, however, present any way of checking the correctness of their implementation of the theory.

Using proof to check the correctness of the implementation of a model checker has also been addressed previously [140]. The authors use the correspondence of L_μ model checking with infinite parity games [58] to generate a proof of the property being checked. This proof is generated during the model checking run.

This approach is more efficient than our fully-expansive one. However, the validity of the generated proof needs to be checked separately. The authors mention the possibility of using a SAT solver to perform this check efficiently. This shifts the burden of tool correctness to the SAT solver,[1] and leaves it at that.

An alternative is to prove the model checking theory in a constructive logic and extract a model checker from the proof. This has been done for CTL* [178]. This is for an explicit state model checker however, so a direct comparison with out symbolic model checker is not possible. Whereas the approach is elegant, executable algorithms extracted from constructive proofs have had a history of efficiency issues [145, 163].

8.1.4 High-level Integration

Our work achieves a tight integration of model checking and theorem proving, but the technology available for use is limited to what we have implemented. Other technologies

[1] A SAT solver is used to check the validity of a formula f by attempting to find a satisfying assignment for the negation of f. A negative result then means success. However, a negative result cannot be verified efficiently with a theorem prover or any other technique. Thus, we must rely on the SAT solver being sound.

cannot be added without re-implementing them fully-expansively in our framework, other than as oracles, which goes against our general approach.

Projects like the Prosper toolkit [130] and the MathWeb Software Bus [194] address this problem by describing plug-in APIs that allow results to be exchanged between live sessions of different tools, on the same machine or over a network connection. This gives the flexibility of being able to use the right tool for the job (e.g. [181]).

Since the focus is on integrating existing technology, it is difficult to directly compare this approach to integration with ours, which also keeps the development of new technology in mind. The one obvious trade-off is between coarse and fine integration (discussed further in the next section).

For the interested reader, a fairly comprehensive survey of techniques combining model checking and theorem proving (as opposed to the work mentioned here that focuses on techniques *for* combining model checking and theorem proving) is available [186].

8.2 HOL-Voss, VossProver and ThmTac

Checking circuits using switch-level simulation has been around since the early 1980s [24]. In normal simulation, every node in the circuit was assigned a true/false boolean value and node values were checked against given input and output values by directly simulating the operation of the circuit. In symbolic simulation, node values could instead be considered boolean variables, which gave more expressivity to the kind of assertion that could be checked. For instance, instead of checking for two nodes A and B that $A = 0 \iff B = 1 \wedge A = 1 \iff B = 0$, one could simply check that $A \iff \neg B$.

Expressive power was further increased by symbolic trajectory evaluation (STE) in which the symbolic values of nodes at different points in time can be compared. Thus, for an inverter, one could check that the output at time $t + 1$ was the inverse of the input at time t, and so on. Another innovation in STE is the use of "don't care" values to reduce the search space. In effect, STE is a special case of temporal model checking [192].

The VOSS system [166] is an implementation of STE, done in 1990-91. It was implemented in a lazy functional language fl, which borrowed heavily from Edinburgh ML. One distinguishing feature was that booleans were represented internally by BDDs. This allowed native implementation of BDD based algorithms such as STE.

In HOL-VOSS [98], VOSS was interfaced to the HOL theorem prover. The assertion language of VOSS was formalised in HOL and a proved assertion returned as a HOL theorem. The actual checking was done external to HOL and was wrapped up in a HOL derived inference rule that relied on an oracle (i.e. VOSS). This allowed positive results from VOSS to be manipulated within HOL. The authors cited mathematical rigour and access to HOL's powerful logic as the driving factors.

However, they discovered that in the HOL-VOSS approach the access to the model checker was too coarse [3]. In case of a failure, the user could not access the model checker's debugging and analysis facilities. This was unacceptable since model checking rarely returns a positive result the very first time and a considerable amount of manual interaction with the model checker is required in the debugging phase before running the checker again. What was required was a seamless interface that allowed the user to move to the model checker from the theorem prover and back again as required.

This requirement led to the implementation of the VOSSPROVER tool [3, 85]. This was a lightweight theorem prover built on top of VOSS, again implemented in fl. The system was architected in a manner similar to HOL: it was LCF-style, with a higher order logic as the object language. The system had core and derived rules of inference in the style of HOL. Standard rules of inference were supplemented by special rules of inference for STE calls.

Though VOSSPROVER was an improvement on HOL-VOSS, the translation of specifications between VOSSPROVER's logic and fl was awkward and often led to duplication of effort. Also, the logic was not expressive enough to be really useful. This led to the development of THMTAC, a system similar to VOSSPROVER but implemented in Lifted fl, a language in which fl was embedded into itself. Thus Lifted fl programs could evaluate fl expressions directly instead of having to translate back and forth, achieving the goal of efficiently unifying the model checker and theorem prover's specification languages.

A major difference between this setup and our system is that it implements the STE-related rules of inference in the core as trusted code, whereas we implement our rules for L_μ model checking as untrusted derived rules in HOL.

The advantage we gain is a finer-grained integration which is thus more flexible and in keeping with our overall aim of developing a platform for rigorously programming algorithmic verification techniques. We lose in the area of performance. It is not possible to meaningfully quantify the tradeoff, but we feel that recent advances in model checking and theorem proving technology, and indeed in the speed of computation, should offset our loss somewhat.

THMTAC and further improvements were made after the primary authors joined the industry. The FORTE system which was developed at Intel and which is based on Lifted fl has recently been made available to the public. Preliminary investigation of the code and documentation shows that it supports significantly improved infrastructure for constructing and manipulating models and for debugging, in keeping with the pragmatic approach of the authors.

The Intel implementation of FORTE has recently been moved from Lifted fl to the reFLect language [79] which provides strongly-typed reflection capabilities. Another group at Intel is developing a set of verification tools based around the forspec specification language [9]. This work is not public domain.

Systems verified using the tools above include the IA-32 instruction length decoder [3], and two floating-point adder/substractor circuits from Intel processors [2].

8.3 Model Checking in PVS

The PVS theorem prover [149] was interfaced to a model checker for L_μ [154] in 1995. Since then, several improvements such as automatic abstraction [161] have been made.

This work is significant because it is the first high-profile integration of an existing and widely used theorem prover with model checking technology. Unlike the authors of VOSSPROVER, the architects of this work did not have the luxury of building a theorem proving environment tailor-made to support their model checker. However the object language of PVS is a classical higher order logic (in fact the type system is more powerful than HOL's, with native support for dependant and predicate subtypes). The power of this logic makes the integration task easier.

The design philosophy of PVS is to support mechanical reasoning using very efficient decision procedures. The architecture is not LCF-style though it has the concept of using high-level inferences to build proof strategies. Work proceeds by applying a relatively small number of powerful rules of inference to transform the goal.

This leads to a high level of automation and ease of use, which is no mean achievement for an industrial strength theorem prover. The tradeoff is a relatively higher incidence of soundness issues [78] and lack of programmability by the user. PVS does have a user accessible API, but it is poorly documented and infrequently used [102].

The integration formalises L_μ in PVS, and interfaces this formalisation to a model checker for L_μ [97]. The temporal logics CTL and fairCTL are also formalised, along with their translations into L_μ. This allows users to specify assertions in any of these logics. The goal is then automatically translated into a form the model checker can use.

The model itself is defined by an initial state and a next state predicate in the PVS logic. The call to the model checker is wrapped within an atomic proof rule called `model-check` that returns either the proved goal as a theorem, or unproved goals corresponding to non-satisfying initial states of the model, or it may just simplify the boolean fragments of the goal.

Much like our work, the authors readily admit [154] that this is not an advance in model checking technology. Rather, the power comes from being able to combine model checked goals in a rigorous manner within the theorem prover, and access to an expressive language with a solid proof system.

The expectation is that certain goals are better suited to model checking (thus saving the human prover time) and that goals too big to model check can conceivably be decomposed in the theorem prover, model checked separately and recombined in a mathematically sound manner.

Later improvements [161] added an automatic abstraction framework that allows the model checker to attack larger state spaces. This improvement was an influential development in abstraction for model checking. However, the integration uses the same approach of using an atomic proof rule (called `abstract-and-mc`) and the following remarks apply to it as well.

The approach is similar to ours in that the assertion language is L_μ, other temporal logics are supported and a positive result from the model checker can be seamlessly incoporated in the theorem prover logic. It is different in that it is not LCF-style. This provides the advantage of speed and the flexibility of replacing the underlying model checker without too much effort.

It has the disadvantage that, due to the one-shot nature of the integration, the user is stuck in the theorem proving world with no access to the model checker's debugging facilities; the very disadvantage which led the creators of HOL-VOSS to abandon it in favour of VOSSPROVER.

Recent infrastructural improvements have resulted in the Symbolic Analysis Laboratory (SAL), which combines model checking, abstraction, deduction and automatic invariant generation with PVS using an intermediate modelling language [168].

Verifications that used the PVS model checker interface include a data-link protocol used by Philips Corporation [161] and a cache coherence protocol [162].

8.4 The Symbolic Model Prover

The VOSSPROVER and PVS experiments suggest a list of desirables in a good integration of model checking and theorem proving, roughly in decending order of importance:

1. Speed.
2. Fine-grained integration.
3. Mathematically rigorous interface.
4. Modular architecture.

These issues are addressed in the Symbolic Model Prover (SYMP) framework developed by Berezin [14] in his PhD thesis in 2002. The core of the work is a Gentzen-style sequent calculus in which each judgement

$$M; \Gamma \Rightarrow \Delta$$

carries a model M in addition to the usual antecedents Γ and consequents Δ. The proof system contains the usual inference rules for classical higher order logic and additional rules that transform the model part of the judgement.

This is further generalised by implementing a system which takes a data structure (representing a sequent) and an associated proof system as parameters and generates an Emacs environment for proof management. This allows users to work with custom sequents and proof systems adapted to their particular problem domain. Since the basic Gentzen calculus is sound, as long as the additional rules of inference are sound the entire system is sound. This system is called a *theorem prover generator*. This usage is somewhat misleading since the greatest effort in developing a theorem prover goes into developing the proof system, a task that SYMP leaves to the user. Nevertheless, this scheme gives the user a very modular and flexible working environment.

Berezin implemented in SML several of the more useful proof rules for the domain of integrating classical higher order logic with model checking as an example of this modularity. For example, integration with a model checker can be represented by the rule

$$\frac{MC(M, \bigwedge \Gamma \rightarrow \bigvee \Delta) = \mathbf{true}}{M; \Gamma \Rightarrow \Delta} MC$$

where $MC(M, \phi)$ is a function that checks whether M satisfies ϕ.

It should be noted that unlike PVS, there is no need to wrap the entire model checking call in one rule. For instance, abstraction and decomposition can be represented by separate rules that transform the model part of the judgement only. Since the model is part of the judgement, it can be represented efficiently. Thus there is no loss of compositionality and no efficiency issues, whereas with PVS a rule that transformed only the formulas representing the model could well cause an exponential blowup in the size of the formulas.

At the same time, the system retains VOSSPROVER's advantage of representing decision procedure calls by rules of inference, resulting in a mathematically rigorous interface.

The only drawback of this system is that it is perhaps too general and a considerable amount of effort is required to achieve a fine-grained integration. Of course, it is always

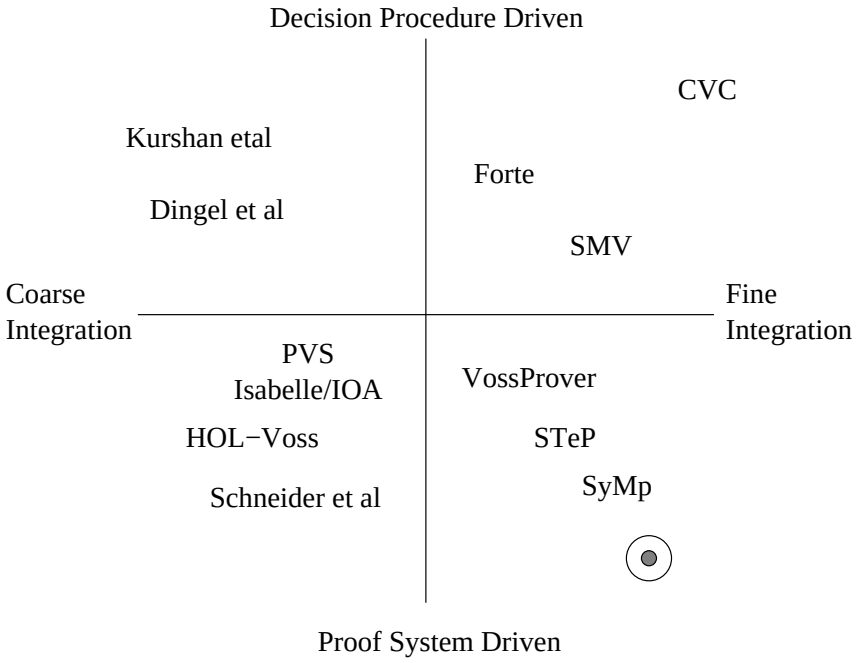

Figure 8.1: Approaches to Integration

possible to simply wrap existing tools in inference rules, but then we run across the integration problems encountered by HOL-VOSS and PVS, which interfere with the usual model checking work flow. This could be the reason why development on SYMP has been more or less dormant since the completion of Berezin's thesis: the last public release of SYMP was in June 2001, and though Berezin's thesis notes a future project to port the Analytica [42] system to SYMP, the latest paper on Analytica [40] has no mention of this.

Our system can be considered as a special case of SYMP in which the proof system is classical higher order logic and the domain is model checking. Our implementation is much more fine grained than the one implemented as a case study in Berezin [14] and we also have access to a powerful theorem proving environment. We conjecture that were the case study integration as fine grained as ours, it would have similar performance.

SYMP was used for security protocol analysis using the Athena approach [177] and verification of C programs in the *Reedpipe/CProver* proof system [14].

8.5 Conclusion

Approaches to integrating model checking and theorem proving could be thought of as falling along a two dimensional graph as in Fig. 8.1. Along the horizontal axis we have granularity of integration, i.e. we move from coarse to fine integration. Along the vertical axis we show whether the proof system or the decision procedures guide the overall effort. The dotted circle shows where we believe our system to lie. This is of course no more than a very rough guide.

Here by coarse integration we mean that the two techniques interact only at a very high level of abstraction, e.g. by passing parameters and results via files or by wrapping the model checker in an atomic proof rule. In fine-grained integration the two domains

can interact with each other at the desirable level of abstraction at any stage of execution. As we move from coarse to fine integration, the system becomes more flexible and easier to instrument and manipulate, but the development effort required also increases.

Proof system driven systems typically give priority to rigour, at some cost to automation and performance. Decision procedure driven systems tend to make the reverse tradeoff. Development effort is high in both extremes, in the former case because of the amount of formal proof involved, in the latter because the tight integration between procedures often requires users to hack the code itself.

We see that chronologically, integrations have become more fine grained. This is partly because of better development infrastructures and partly because of the demand for flexible and adaptable systems. The performance penalty is offset by new technology, both in algorithms and in hardware.

There is no such trend to be seen in whether the proof system or the decision procedures drive the tool. This is because the tradeoffs involved (e.g. expressivity vs speed) are more theoretical and cannot be resolved simply through better engineering. There is as such no happy middle, as different tasks require different strategies. Our own system focuses on providing HOL users with a sound programming platform for symbolic model checking and related techniques. We believe it achieves our goal of close integration without unacceptable performance deterioration.

Chapter 9

Summary

In this chapter we summarise the work done. We also discuss its limitations and directions for the future.

9.1 Work Done

We have developed a formal theory of the modal μ-calculus L_μ in the HOL theorem prover and used it to write a model checker that justifies every step of the execution by proof in HOL.

To reduce the theorem proving overhead, the primitive steps of the model checking are carried out outside HOL in a high-performance BDD engine. In this sense, we extend the fully expansive approach to symbolic model checking: as long as the primitive BDD operations are sound, the whole system is sound (subject to the usual qualifications about the underlying operating environment and hardware).

We have demonstrated that the resulting performance penalty is within acceptable bounds. Additionally, we have formalised and implemented basic and advanced model checking techniques to exercise the theory. Thus we have have shown the feasibility of using a fully-expansive theorem prover as a platform for securely programming new theories and tools for verification.

The code for our work consists of just over 7000 lines of Moscow ML, of which just under half are HOL proof scripts for 265 theorems. These are the offline theorems. Most of the rest of the code is devoted to runtime proof.

Formalised Theories We have formalised the following mathematical theories in HOL:

- The syntax and semantics of L_μ, up to the existence of fixed-points assuming the (abstracted) model is finite (§2.4.1). We note that the lack of support for variable binding in higher order abstract syntax in HOL made this task difficult. Also, much of the formalisation had to be done in non-obvious ways so that the subsequent formalisation of model checking algorithms could be executed efficiently.
- The syntax and semantics of CTL and a characterisation of the fixed-point operators (§4.2).
- An embedding of CTL into L_μ (§4.3). An unexpected occurrence in this embedding is the use of Hilbert's selection operator to extend possibly finite computation paths

to infinite ones. This is needed because the standard semantics of CTL assume infinite paths but there is no such constraint in the standard semantics for L_μ.

Formalised Algorithms The following algorithms have been formalised with some effort towards minimising runtime theorem proving overhead:

- A symbolic model checking algorithm for L_μ (§2.4.2).
- Caching (§3.2) and alternation depth optimisations (§3.3).
- Counterexample generation lifted to L_μ (§5.2.2).
- Functional abstraction by constructing equivalence classes over logical relations between atomic propositions (§5.2.1 and §5.3.1).
- Counterexample detection using a SAT solver (§5.2.3; the SAT solver was integrated elsewhere [73]).
- Counterexample-guided abstraction refinement (§5.2.4).

Formalised Systems The following systems were formalised for the case studies:

- The specification and an implementation of the Advanced Microcontroller Bus Architecture from ARM Limited was formalised for the primary case study (Chapter 6). The implementation was done by us.
- A model of a well-known pedagogical three-stage pipelined arithmetic and logic unit was formalised for the performance study (Chapter 7).

9.2 Limitations

9.2.1 Theoretical Issues

The following theoretical aspects of the work deserve further consideration:

- The formalisation has been carried out in classical higher order logic (HOL). Though we gain much in the way of a considerable amount of existing formal theory and theorem proving support, there are trade-offs:
  - Though HOL is sufficient to express most formalisms, it lacks a satisfactory treatment of foundational aspects of set theory.

    In Lazic et al [113] for instance, a non-standard model of set theory is used to give a direct construction of operational models of concurrency in the presence of unbounded nondeterminism. This may prove troublesome to formalise in the current context. Admittedly, such occurrences are rare in the literature.
  - The non-constructive nature of HOL means that we cannot extract executable programs from the proofs without considerable effort; effectively, not without contructivising the theory.

- Our formalisation of the theory keeps explicit track of α-equivalence using a rudimentary form of substitution (see Definitions 2.10 and 2.11 in Chapter 2). This was because of the lack of native support for variable binding in higher order abstract syntax in HOL. A recent formalisation in HOL [146] of some axioms for α-conversion [70] may provide a cleaner solution to this problem. However, we cannot at the moment conjecture whether this will increase or decrease performance.

- A more powerful type system, and PVS-like predicate subtypes in particular, would be very helpful in reducing clutter and complexity in the formal proofs. First steps towards this have been taken [93].

9.2.2 Practical Shortcomings

Several standard model checking optimisations have not yet been implemented. Without these, it is not possible to meaningfully compare the performance of our system against more mature ones. For the moment we can only compare the system against a non-formal version of itself. This type of relative comparison is unlikely to convince industrial practitioners who are usually interested in absolute performance figures.

As is evident from Chapter 8, much of current research is focused on enabling cooperation between various techniques. Though in theory we can implement any such technique in HOL, the absence of a general framework gives any implementation an ad hoc nature.

All we have is a philosophy: do everything fully-expansively for better assurance of soundness, closer integration, and scriptability, and exploit the asymmetric cost of proof checking vs. proof search whenever possible for efficiency. Though we have taken first steps, a SYMP-style framework [14] that embodies this philosophy with an emphasis on combining technologies would be desirable. Whether or not this would be over-engineering depends on the eventual domain of use.

The approach of embedding model checking algorithms in a theorem prover carries a clear performance penalty. There are benchmarking issues about what kind of problems this approach is best suited to. There are also engineering issues about how to improve the performance without abandoning the fully-expansive approach. And finally there are usage issues about just what is considered an acceptable penalty in a given situation. All these will need to be addressed at some point.

9.3 Future Directions

Immediate goals include:

- Improving the performance of the model checking component of the work by improving the existing code and adding standard optimisations.

- Improving the usability of the system by adding to the debugging and failure-analysis capabilities, and adding support for assume-guarantee style reasoning.

- Formally proving the soundness of the BDD calculus. We believe the calculus to be sound. However, a formal proof is desirable.

- Implementing abstraction over models where atomic propositions can be over any logic that is decidable by the theorem prover. There are at least two candidate techniques for this [108, 161]. This would enable model checking of infinite state systems.
- Adding support for other assertion languages such as CTL* [53] and PSL/Sugar [75].

In the long term, there are several ideas worth considering:

- Efficiency.
  - A scheme for speeding up proofs in a fully-expansive theorem prover has been described [20]. The idea is to delay the logical justification of a theorem but provide the structure immediately. This could be extended to our system where proofs could be separated into operations on BDDs and the theorem proving overhead. These could be handed out to separate processors and the theorem proving part "lazyfied". Theoretically, this would all but eliminate the performance penalty at the cost of an additional processor. In practical terms, this seems a sensible trade-off.
  - The approach so far has been to formalise logics in strictly textbook fashion. Often a non-formal implementation (conducted for learning purposes) has been retrofitted with formal theory. This has left room for several local optimisations.
  - The caching algorithm can easily be extended to use algebraic simplification of L_μ formulas. It would be interesting to see if such aggressive caching at the term level improves performance beyond what we get by caching BDDs only.
  - The optimisations mentioned in §3.1 can be implemented.
- Expressiveness.
  - Data independence techniques developed in [111] provide a powerful abstraction ability assuming a small set of network invariants hold. Originally developed for CSP [88], the theory has recently been given a semantics-independent treatment [112] and could prove a useful addition to the system.
  - Game-semantic models of functional languages with control operators and higher order references can be model checked [147]. Since HOL has built-in support for functional programming [138, 174], integrating this ability would achieve a far deeper integration of model checking and theorem proving than what we have achieved so far.
- Infrastructure. We need a framework in which model checking oriented technologies can be added in a clean modular fashion. Possible approaches to follow include [14, 180].

Appendix A

The HOL theorem prover

This appendix provides a quick overview of higher order logic (HOL) and the HOL theorem prover. For a more detailed and formal treatment, see the HOL System Description [175].

In the late 19^{th} century, dissatisfaction with the foundations of mathematics led to the treatment of logic as a formal mathematical discipline [189]. Higher order logic in the context of our work refers to Church's simple theory of types [32] together with Robin Milner's type polymorphism [132].

Types are interpreted as standard sets. This was considered sufficient for the purposes of hardware verification [71], which was the first use of the logic. Types can be type variables $\alpha, \beta, \ldots$ (occuring (unlike Church) in the object language as single polymorphic terms), compound types (e.g. pairs, lists and records), and function types. Atomic types can be considered as the zero-arity case of compound types.

Terms denote elements of the sets denoted by the term's type. The term structure is that of the typed λ-calculus with distinguished constants. The formula syntax is that of predicate calculus with equality.

The deductive system is based on natural deduction, consisting of eight rules of inference (reflexivity, β-conversion, modus ponens etc.) and five axioms (Hilbert's choice, exluded middle etc.). Judgements forming the root of derivation trees in this system are called *theorems*. *Definitions* are created by adding new constants in a manner that cannot introduce inconsistency.

The HOL-4 (or just HOL) theorem prover is a Moscow ML [167] implementation of this deductive system (the original version was implemented by M. J. C. Gordon and T. F. Melham in an old version of ML [76]). Theorem proving in HOL is usually done in an interactive session, by setting up a goal and then issuing commands that apply the rules of inference to the goal. A goal that can be reduced to the axioms or ground rules is a theorem.

Using only the core inference rules would be tedious. HOL provides procedures called *tactics* that sequentially apply several rules of inference to a goal to transform it in some desired manner (e.g. case splits, resolution etc.). *Tacticals* are higher level procedures that allow the user to combine tactics to further aid in automation. HOL tactics include decision procedures, simplifiers and rewrite systems.

HOL also provides facilities for writing complicated definitions without resorting to the primitive rules [174].

Informally, a *theory* is a collection of definitions and theorems. HOL has theories for

higher order logic, natural numbers, records, pairs, integer and real arithmetic, sets,[1] probability, temporal logics and automata, bit-vectors, strings and algebra among others. Related theories together with any appropriate decision procedures are packaged in *libraries*.

An important aspect of the HOL architecture is that whereas terms in the logic can be freely constructed, theorems can be constructed using the core axioms and inference rules only, i.e. by proof.[2]. Thus all tactics and tacticals can ultimately be unfolded into applications of the core rules. This reduces the size of trusted code to the small kernel of core rules. If the core rules are sound,[3] then all of HOL is sound. This is called the *LCF-style* or *fully-expansive* approach to implementing a deduction system.

Certain decision procedures are inefficient to implement in this way. HOL allows the user to import the result of tools external to the logic as theorems. However, these theorems are tagged to indicate that they were not derived fully-expansively.

A very useful side-effect of the LCF-style approach is that developers can be granted full access to HOL (except the kernel) without any soundness worries. Since the meta-language Moscow ML is a full-blown functional programming language, this gives HOL virtually unlimited and easy scriptability. This is ideal for users wishing to program procedures which require access to a powerful proof system. Correctness by construction comes as an extra bonus.

HOL has been used in several verifications. Recent examples include microprocessors [64] and number theoretic algorithms [95].

[1] It shold be noted that HOL sets are not ZF sets [173]. A set in HOL is simply a predicate $P : \alpha \to \texttt{bool}$ and $x \in P$ is equivalent to $P(x)$.

[2] HOL does provide a function `mk_thm` for converting arbitrary terms to theorems, but none of our proofs or the HOL theories they rely on use this. We know this because theorems constructed via `mk_thm` are tagged as such.

[3] HOL's core rules of inference have been proved sound on paper ([76], Chapter 16).

Appendix B

Formalised version of theorem 4.10

This appendix contains the complete proof that the standard syntactic embedding of CTL into L_μ preserves semantics. We include it because we were unable to find a formal treatment in the literature of this commonly cited result. The account given here closely follows the formal HOL proof. We will use subscripts to distinguish between the components of M and $\mathcal{T}M$.

Theorem B.1

$$\forall M f. [\![f]\!]_M = [\![\mathcal{T}(f)]\!]_{\mathcal{T}M} \bot$$

Proof By induction on the definition of f.

- $f \equiv True$. Trivial by D4.4,D4.8, D4.9.
- $f \equiv p$.

$$\begin{array}{rl} & [\![p]\!]_M \\ = & \{s | p \in L_M(s)\} \quad \text{by D4.4} \\ = & \{s | \mathcal{T}(p) \in L_{\mathcal{T}M}(s)\} \quad \text{by D4.8, D4.9} \\ = & [\![\mathcal{T}(p)]\!]_{\mathcal{T}M} \bot \quad \text{by D2.3} \end{array}$$

- $f \equiv \neg f'$.

$$\begin{array}{rl} & [\![\neg f']\!]_M \\ = & S \backslash [\![f']\!]_M \quad \text{by D4.4 and set theory} \\ = & S \backslash [\![\mathcal{T}(f')]\!]_{\mathcal{T}M} \bot \quad \text{by the IH} \\ = & [\![\mathcal{T}(\neg f')]\!]_{\mathcal{T}M} \bot \quad \text{by D2.3, D4.8} \end{array}$$

- $f \equiv f' \wedge f''$.

$$\begin{array}{rl} & [\![f' \wedge f'']\!]_M \\ = & [\![f']\!]_M \cap [\![f'']\!]_M \quad \text{by D4.4 and set theory} \\ = & [\![\mathcal{T}(f')]\!]_{\mathcal{T}M} \bot \cap [\![\mathcal{T}(f')]\!]_{\mathcal{T}M} \bot \quad \text{by the IH} \\ = & [\![\mathcal{T}(f' \wedge f'')]\!]_{\mathcal{T}M} \bot \quad \text{by D2.3, D4.8} \end{array}$$

- $f \equiv \mathbf{EX}f'$. For any state $s \in S_M$,

$$\begin{array}{rl} & s \in [\![\mathbf{EX}f']\!]_M \\ \iff & s \models_M \mathbf{EX}f' \quad \text{by definition of} [\![-]\!]_M \\ \iff & \exists \pi. \mathit{PATH}\ M \pi s \wedge \pi_1 \models_M f' \quad \text{by D4.4} \\ \iff & \exists \pi. \mathit{PATH}\ M \pi s \wedge \pi_1 \in [\![f']\!]_M \quad \text{by definition of} [\![-]\!]_M \\ \\ \iff & \exists \pi. \mathit{PATH}\ M \pi s \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by the IH} \\ \iff & \exists \pi. R_M(s, \pi_1) \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by D4.2} \\ \iff & \exists \pi. s \dot{\rightarrow} \pi_1 \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \ \text{by definition of} \rightarrow \end{array}$$

  Now define our existential witness π by

$$\begin{array}{rcl} \pi_0 & = & s \\ \pi_(n+1) & = & \text{if}\,(n = 0)\,\text{then}\,\pi_1\,\text{else}\,\varepsilon r. R_M(\pi_n, r) \end{array}$$

  where ε is Hilbert's selection operator. Then simplifying and continuing,

$$\begin{array}{rl} \iff & \exists s'. s \dot{\rightarrow} s' \wedge s' \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by definition of} \pi \\ \iff & s \in \{s | \exists s'. s \dot{\rightarrow} s' \wedge s' \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot\} \ \text{by defn of} \in \\ \iff & s \in [\![\langle . \rangle \mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by D2.3} \\ \iff & s \in [\![\mathcal{T}(\mathbf{EX}f')]\!]_{\mathcal{T}M}\bot \quad \text{by D4.8} \end{array}$$

  and we have the required result by extensionality.

- $f \equiv \mathbf{EG}f'$. Define

$$\tau(W) = [\![\mathcal{T}(f) \wedge \langle . \rangle Q]\!]_{\mathcal{T}M} \bot [Q \leftarrow W]$$

  and we have

  - $\subseteq$ direction.

$$\begin{array}{rl} & [\![\mathbf{EG}f]\!]_M \subseteq [\![\mathcal{T}(\mathbf{EG}f)]\!]_{\mathcal{T}M}\bot \\ \iff & [\![\mathbf{EG}f]\!]_M \subseteq \bigcap_n \tau^n S_{\mathcal{T}M} \quad \text{by P2.5,D4.8,D2.3,D4.9} \\ \iff & \forall n. [\![\mathbf{EG}f]\!]_M \subseteq \tau^n S_{\mathcal{T}M} \quad \text{by set theory} \end{array}$$

    Then induction on n gives

    * $n \equiv 0$. Immediate by D2.3,D4.9,P2.5.
    * $n \equiv n' + 1$. Consider the "outer" IH,

$$\begin{array}{rl} & [\![\mathbf{EG}f]\!]_M \subseteq \tau^{n'} S_{\mathcal{T}M} \\ \Rightarrow & \tau([\![\mathbf{EG}f]\!]_M) \subseteq \tau(\tau^{n'} S_{\mathcal{T}M}) \ \text{by P2.4} \\ \iff & [\![\mathbf{EG}f]\!]_M \subseteq \tau^{n'+1} S_{\mathcal{T}M} \ \text{by L4.6,D4.8,D2.3} \end{array}$$

      which is the required result.

– $\supseteq$ direction.

$$\begin{array}{rl} & [\![\mathbf{EG}f]\!]_M \supseteq [\![\mathcal{T}(\mathbf{EG}f)]\!]_{\mathcal{T}M}\bot \\ \iff & [\![\mathbf{EG}f]\!]_M \supseteq \bigcap_n \tau^n S_{\mathcal{T}M} \quad \text{by P2.5,D4.8,D2.3,D4.9} \end{array}$$

Now consider some $s \in \bigcap_n \tau^n S_{\mathcal{T}M}$. By Proposition 2.5,

$$s \in \tau\big(\bigcap_n \tau^n S_{\mathcal{T}M}\big)$$

Suppose π is a path starting at s. Then by the definition of τ, $\pi_1 \in \bigcap_n \tau^n S_{\mathcal{T}M}$. We use the ε operator to pick π_1 for us (this is needed because totality of $R.M$ only tells us that π_1 exists). By repeatedly using ε to pick the appropriate next state on the path, we can construct π such that $\forall i.\pi_i \in \bigcap_n \tau^n S_{\mathcal{T}M}$. But for any s', $s' \in \bigcap_n \tau^n S_{\mathcal{T}M} \Rightarrow s' \models_M f$ by the definition of τ and the outer IH. Thus we have that $\forall i.\pi_i \models_M f$, and we have the required result by Definition 4.4.

- $f \equiv \mathbf{E}[f'\mathbf{U}f'']$. Define

$$\tau(W) = [\![\mathcal{T}(f'') \vee (\mathcal{T}(f') \wedge \langle . \rangle Q)]\!]_{\mathcal{T}M} \bot [Q \leftarrow W]$$

and we have

– $\subseteq$ direction.

$$\begin{array}{rl} & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \subseteq [\![\mathcal{T}(\mathbf{E}[f'\mathbf{U}f''])]\!]_{\mathcal{T}M}\bot \\ \iff & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \subseteq \bigcup_n \tau^n \emptyset \quad \text{by P2.5,D4.8,D2.3,D4.9} \end{array}$$

Now consider some $s \in [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M$. Then by Definition 4.4 we have,

$$\exists \pi.\mathit{PATH}\ M \pi s \wedge \exists k.\pi_k \models_M f'' \wedge \forall j.j < k \Rightarrow \pi_j \models_M f'$$

We proceed by induction on the length $|^k\pi|$ of $^k\pi$.

* $|^k\pi| = 0$. Note that since π is infinite, $|^k\pi| = k$ by the definition of $^k\pi$. So $k = 0$. This implies $s \models_M f''$ by Definition 4.2, i.e. $s \in [\![f'']\!]_M$ and we are done by the definition of τ and the outer IH.
* $|^k\pi| = k' + 1$. We note again that $k = k' + 1$. Consider the path π^1. Then,

$$\begin{array}{rl} & \pi_k \models_M f'' \\ \iff & \pi^1_{k'} \models_M f'' \end{array}$$

and

$$\begin{array}{rl} & \forall j.j < k \Rightarrow \pi_j \models_M f' \\ \iff & \forall j.j + 1 < k \Rightarrow \pi_{j+1} \models_M f' \\ \iff & \forall j.j < k' \Rightarrow \pi^1_j \models_M f' \end{array}$$

So by the IH,

$$\begin{aligned} & \pi_0^1 \in \bigcup_n \tau^n \emptyset \\ \iff \quad & \pi_1 \in \bigcup_n \tau^n \emptyset \\ \iff \quad & s \in \tau\big(\bigcup_n \tau^n \emptyset\big) \quad \because s \models_M f' \text{ and defn of } \tau \end{aligned}$$

and we are done by the outer IH.

– $\supseteq$ direction.

$$\begin{aligned} & [\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M \supseteq [\![\mathcal{T}(\mathbf{E}[f' \mathbf{U} f''])]\!]_{\mathcal{T}M} \bot \\ \iff \quad & [\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M \supseteq \bigcup_n \tau^n \emptyset \text{by P2.5,D4.8,D2.3,D4.9} \\ \iff \quad & \forall n. [\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M \supseteq \tau^n S_{\mathcal{T}M} \text{by set theory} \end{aligned}$$

Then induction on n gives

* $n \equiv 0$. Immediate by D2.3,D4.9,P2.5.
* $n \equiv n' + 1$. Consider the outer IH,

$$\begin{aligned} & [\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M \supseteq \tau^{n'} \emptyset \\ \Rightarrow \quad & \tau([\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M) \supseteq \tau(\tau^{n'} \emptyset) \text{ by P2.4} \\ \iff \quad & [\![\mathbf{E}[f' \mathbf{U} f'']]\!]_M \supseteq \tau^{n'+1} \emptyset \text{ by L4.7,D4.8,D2.3} \end{aligned}$$

which is the required result.

□

Appendix C

Usage example

In this appendix we demonstrate counter-example guided abstraction refinement for a small hand-crafted example, to lend some concreteness to the material in Chapters 2, 3, 4 and 5. The output from HOL has been modified to aid in readability.

We consider a system having a state space over two boolean variables v_0 and v_1, with an initial state predicate

$$INIT(v_0, v_1) \;=\; \neg v_0 \wedge \neg v_1$$

and transition relation predicate

$$\begin{array}{rcl} TRANS((v_0,v_1),(v_0',v_1')) & = & v_0' = \begin{array}{lll} \texttt{case} & v_1 \wedge v_0 & \rightarrow \texttt{T} \\ \mid & v_1 \oplus v_0 & \rightarrow v_0 \\ \mid & \texttt{T} & \rightarrow \texttt{F} \end{array} \\ & & \wedge \\ & & v_1' = \texttt{case}\,\texttt{T} \rightarrow \texttt{T} \end{array}$$

We will let $\bar{v}$ and $\bar{v}'$ abbreviate (v_0, v_1) and (v_0', v_1') respectively. Note that there are four concrete states. To avoid clutter we label them as follows $0 = (F, F)$, $1 = (T, F)$, $2 = (F, T)$ and $3 = (T, T)$. Note that 0 is the only initial state.

The first step is to compute the abstraction function. The appropriate function call with $INIT$ and $TRANS$ as arguments returns a term-BDD h whose term part looks like

$$\begin{array}{rllll} & SCC & TRANS & (\neg v_0, \neg v_1)\ \bar{v} = \neg\hat{v}_0 \wedge \neg\hat{v}_1 \\ \wedge & SCC & TRANS & (v_0, \neg v_1)\ \bar{v} = \hat{v}_0 \wedge \neg\hat{v}_1 \\ \wedge & SCC & TRANS & (v_0, v_1)\ \bar{v} = \neg\hat{v}_0 \wedge \hat{v}_1 \end{array}$$

indicating that we have three abstract states from the four concrete ones. The third argument to SCC is present because we are considering the set of states SCC as a predicate here.[1] We can break down this computation along the lines of §5.2.1 :

1. $F = \{v_1 \wedge v_0, v_1 \oplus v_0\} = FC_0$
2. $VC_0 = \{v_0, v_1\}$

[1] See remarks in footnote 1 of Appendix A.

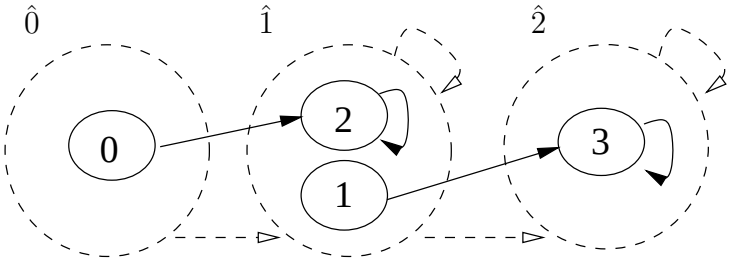

Figure C.1: Concrete and abstracted model

3. It can be shown that

$$\forall \psi \in FC_0.\ 1 \models \psi \iff 2 \models \psi$$

whereas

$$\neg\exists \psi \in FC_0.\ \exists s \neq 3.\ 3 \models \psi \iff s \models \psi$$

and ditto for the concrete state 0. Then h can computed as in §5.3.1.

To avoid clutter we will label the abstract states $\hat{0} = \neg\hat{v}_0 \wedge \neg\hat{v}_1$, $\hat{1} = \hat{v}_0 \wedge \neg\hat{v}_1$ and $\hat{2} = \neg\hat{v}_0 \wedge \hat{v}_1$ and extend the $\bar{v}$ notation to $\bar{\hat{v}}$. Figure C.1 shows how the concrete states (solid circles) are grouped into abstract ones (dashed-circles).

The next step is to build the abstract model. We supply $INIT$, $TRANS$ and h to our system, which returns the abstract model $\hat{M}$:

$$\begin{array}{rcl}
(\hat{AP} & = & \{\hat{v_0}, \hat{v_1}\}, \\
\hat{S} & = & \mathcal{U} : (\texttt{bool} \times \texttt{bool}), \\
\hat{S_0} & = & \lambda\bar{\hat{v}}.\exists\bar{v}.h(\bar{v},\bar{\hat{v}}) \wedge INIT(\bar{v}), \\
\hat{T} & = & \lambda a.\lambda(\bar{\hat{v}},\bar{\hat{v}}').\exists(\bar{v},\bar{v}').h(\bar{v},\bar{\hat{v}}) \wedge h(\bar{v}',\bar{\hat{v}}') \wedge TRANS((v_0,v_1),(v_0',v_1')), \\
\hat{L} & = & \lambda\bar{\hat{v}}\ p\ (\lambda t.\ \texttt{if}\ t = \text{''}v_0\text{''}\ \texttt{then}\ \exists v_1.(\lambda\bar{v}.h(\bar{v},\bar{\hat{v}}))(\texttt{T},v_1) \\
& & \qquad\qquad \texttt{else}\ \exists v_0.(\lambda\bar{v}.h(\bar{v},\bar{\hat{v}}))(v_0,\texttt{T})\ p)
\end{array}$$

This looks similar to Definition 5.1 but several details need explanation:

- S is set to $\mathcal{U}$. See the remarks after Definition 2.9.

- Relational notation is used for the function h. This is because h is not defined directly but rather in terms of the equivalence classes it indirectly induces on the concrete state space. It is thus more natural to treat h as a relation.

- There is an extra λa in front of the definition of $\hat{T}$. Recall that T (or $\hat{T}$) is a set of transition relations indexed by action names. Thus we implement T (or $\hat{T}$) as a function on action names: what follows the λa is indeed a transition relation in the abstract state space. The reason that a is not actually used is because there is only one transition relation in this model, thus action names are not required to pick out which transition relation we want and the default action name "." is always used. In this case, the λa is simply there to preserve type correctness.

- The definition of $\hat{L}$ appears to be completely different from that in Definition 5.1. Intuitively, given an abstract state $\hat{s}$, $\hat{L}$ returns the set of propositions true in the concrete states comprising $\hat{s}$. Note from Definition 2.3 that L (or $\hat{L}$) is eventually used to compute the set $\{s|p \in L(s)\}$ where p is the name (as opposed to the symbolic value) of a proposition.

  Since HOL sets are just predicates, we would like to be able to compute $L(s)(p)$ (or $\hat{L}(\hat{s})(p)$) quickly. We construct the term for $\hat{L}$ in such a way that evaluating $\hat{L}(\hat{s})(p)$ uses the string p to efficiently look up the term representing the set of concrete states in which the value of p is `T`. The term for $\hat{L}$ actually looks like

$$\begin{aligned}\hat{L} \;=\; & \lambda\bar{\hat{v}}\ p\ (\lambda t.\ \texttt{let}\ x = \ \texttt{explode}\ (\texttt{strcat}\ t\ \text{"!!"})\ \texttt{in}\\ & \texttt{if ord}\ ((\texttt{hd}\ o\ \texttt{tl})x) < 49\ \texttt{then}\ \exists v_1.(\lambda\bar{v}.h(\bar{v},\bar{\hat{v}}))(\texttt{T},v_1)\\ & \texttt{else}\ \exists v_0.(\lambda\bar{v}.h(\bar{v},\bar{\hat{v}}))(v_0,\texttt{T}))\ p)\end{aligned}$$

  The term is in fact a Patricia trie, keyed on proposition names, which are strings. Each name is represented as a list of the ASCII values of the characters forming the string. This enables fast searching by comparing natural number values. The "!!" pads the shorter names to the same length as the longest one, noting that the "!" character has a lower ASCII value than any alphanumeric character allowed in a proposition name. This eases the ordering of names.

  This representation was judged (after some benchmarking) to be faster than binary trees. Balanced trees were not considered because the term is constructed only once and never modified, so we can ensure optimal balance at construction time. Splay-trees were not considered because modifying the term after every look-up was too expensive and the system can be architected to reduce the number of look-ups considerably (no more than one in most cases). The extra λt is required to prevent HOL's call-by-value evaluator from descending unnecessary branches.

  The term for $\hat{T}$ would have been similar had there been more than one action. This representation was chosen because it is quicker to evaluate than the obvious one but retains the standard semantics.

Now that we have our model, we are ready to check properties. For our example, we will attempt to verify the CTL property f given by

$$\mathbf{AX}\,\neg v_0$$

which holds in the concrete system. For this we use the model checking procedure from Definition 4.12 that internally translates the CTL to L_μ and calls the L_μ decision procedure from Definition 2.7.

Passing f and M to the system, we get the term-BDD

$$\rho \quad s \models^{\perp}_{\hat{M}} \mathcal{T}(\mathbf{AX}\,\neg v_0) \mapsto b$$

Internally, the first attempt at verification fails with an abstract counterexample trace $\langle\hat{0},\hat{1},\hat{2}\rangle$ (see §5.2.2) because f does not hold in concrete state 3. However, 3 is not actually a reachable state and attempting to compute a corresponding concrete counterexample

(as in §5.2.3) fails. The longest concrete counterexample trace we can manage is $\langle 0, 2\rangle$ indicating that $\hat{1}$ needs to be refined. This is done as in §5.2.4 and now the property can be verified.

Now we can use theorems 4.10 and 5.3 together with the BddEqMp rule from §2.2 to derive the term-BDD

$$\rho \quad s \models_M \mathbf{AX}\,\neg v_0 \mapsto b$$

Note that we cannot use BddOracleThm to extract the term part as a theorem because the property does not hold in all states but only the reachable ones, so that $b \neq \texttt{TRUE}$. Depending on the kind of property, we are usually interested in knowing either that the property is satisfied by all reachable states, or just by the initial states. In this example, we want the former. So we compute the term-BDD *Reachable* for the set of reachable states as in §4.4.1 and use the BddImp rule to get the term-BDD

$$\rho \quad s \in \mathit{Reachable}(T(.))S_0 \Rightarrow s \models_M \mathbf{AX}\,\neg v_0 \mapsto \texttt{TRUE}$$

and finally use BddOracleThm to derive the theorem

$$\vdash s \in \mathit{Reachable}(T(.))S_0 \Rightarrow s \models_M \mathbf{AX}\,\neg v_0$$

as required.

Bibliography

[1] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In Basant R. Chawla, Randal E. Bryant, Jan M. Rabaey, and M. J. Irwin, editors, *Design Automation Conference (DAC)*, pages 538–541. ACM/IEEE, ACM Press, 1998.

[2] M. D. Aagaard, R. B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representation of boolean constants. In Mary Jane Irwin, editor, *Design Automation Conference*, pages 402–407. ACM Press, June 1999.

[3] M. D. Aagaard, R. B. Jones, and Carl-Johan H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics*, volume 1670 of *LNCS*, pages 323–340. Springer, September 1999.

[4] M. D. Aagaard, Thomas F. Melham, and John W. O'Leary. Xs are for trajectory evaluation, booleans are for theorem proving. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods - CHARME '99*, volume 1703 of *LNCS*, pages 202–218. Springer, September 1999.

[5] S. Agerholm and H. Skjodt. Automating a model checker for recursive modal assertions in HOL. Technical Report 92, Aarhus University, January 1990.

[6] Henrik Reif Andersen. An introduction to binary decision diagrams. Available from `http://www.itu.dk/people/hra/bdd97.ps`, October 1997.

[7] Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In Bryan Ackland, editor, *Design Automation Conference*, pages 725–730. ACM, June 2002.

[8] ARM Limited. *AMBA Specification*, 2.0 edition, 1999. ©ARM Limited. All rights reserved.

[9] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec Temporal Logic: A new temporal property-specification language. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 296–211. Springer, April 2002.

[10] R. Arthan. Formal specification of a proof tool. In S. Prehn and W. J. Toetenel, editors, *Formal Software Development Methods*, volume 551 of *LNCS*, pages 356–370. Springer-Verlag, 1991.

[11] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages*, volume 37(1) of *ACM SIGPLAN Notices*, pages 1–3. ACM, ACM Press, January 2002.

[12] Clark Barrett and Sergey Berezin. A Proof-Producing Boolean Search Engine. In Silvio Ranise and Cesare Tinelli, editors, *CADE-19 Workshop: Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, July 2003. Available from `http://www.loria.fr/~ranise/pdpar03`.

[13] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[14] S. Berezin. *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, Carnegie Mellon University School of Computer Science, 2002. Tool URL : `http://www.cs.cmu.edu/~modelcheck/symp.html`.

[15] E.C. Berkeley. *The Computer Revolution*, pages 175–177. Doubleday and Co., 1962.

[16] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*. Springer, March 1999.

[17] Armin Biere, Edmund M. Clarke, and Yunshan Zhu. Combining local and global model checking. In Alessandro Cimatti and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 23(2). Elsevier, 1999.

[18] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and assertions. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming - CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 589–623. Springer, 1995.

[19] Nikolaj Bjørner, Zohar Manna, Henny Sipma, and Tomas Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253(1):27–60, 2001. Tool URL: `http://www-step.stanford.edu`.

[20] Richard John Boulton. Efficiency in a fully-expansive theorem prover. Technical report, University of Cambridge Computer Laboratory, 1994.

[21] R. Boyer and J. Moore. *A Computational Logic*. Academic Press, 1979.

[22] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In Richard C. Smith, editor, *Design Automation Conference*, pages 40–45. ACM/IEEE, IEEE Computer Society Press, 1990.

[23] A. Browne, Edmund M. Clarke, Somesh Jha, David E. Long, and Wilfredo R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. *Theoretical Computer Science*, 178(1-2), 1997.

[24] R. E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, 1984.

[25] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[26] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[27] R. E. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer Aided Verification*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.

[28] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In A. Richard Newton, editor, *Proceedings of the ACM Design Automation Conference*. ACM, June 1991.

[29] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In Richard C. Smith, editor, *Proceedings of the ACM Design Automation Conference*. ACM/IEEE, IEEE Computer Society Press, June 1990.

[30] S. Campos, E.M. Clarke, W. Marrero, and M. Minea. Verifying the Performance of the PCI Local Bus using Symbolic Techniques. In Andreas Kuehlmann, editor, *Proceedings of the IEEE International Conference on Computer Design (ICCD '95)*, Austin, Texas, October 1995.

[31] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 241–257. Springer, March 1996.

[32] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[33] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveriand Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *LNCS*. Springer, March 2002.

[34] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[35] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. Technical Report CMU-CS-92-206, Carnegie Mellon University, October 1992.

[36] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[37] E. M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification - (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[38] E. M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[39] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. of Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.

[40] E. M. Clarke, Michael Kohlhase, Joel Ouaknine, and Klaus Sutner. System description: Analytica 2. In *CALCULEMUS*, number LIP6 2003/010 in Technical Reports, Rome, September 2003. Laboratoire D'Informatique de Paris 6.

[41] E. M. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–642, 1996.

[42] E. M. Clarke and Xudong Zhao. Analytica-A theorem prover in Mathematica. In Deepak Kapur, editor, *Automated Deduction - CADE'92*, volume 607 of *LNCS*, pages 761–763. Springer, June 15-18 1992.

[43] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[44] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(9):725–747, September 1990.

[45] Common Criteria Version 2.1. `http://csrc.nist.gov/cc/`, August 1999.

[46] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. Tool URL : `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

[47] John Cooley. Trip report: Design automation conference 2003. `http://www.deepchip.com/posts/dac03.html`, January 2004.

[48] D. C. Cooper. Theorem-proving in arithmetic without multiplication. *Machine Intell.*, 7:91–100, 1972.

[49] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Oddifredi, editor, *Logic and Computer Science*. Academic Press, 1990.

[50] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines using boolean functional vectors. In *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[51] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines using symbolic execution. In Joseph Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[52] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.

[53] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994.

[54] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*. Springer, November 2002.

[55] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[56] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, July 2003.

[57] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abtraction, assumption-commitment style reasoning and theorem proving. In *Proceedings of the 1995 Workshop on Computer Aided Verification*, volume 939 of *LNCS*, pages 54–69. Springer, 1995.

[58] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science*, pages 368–377. IEEE Computer Society Press, 1991. Extended abstract.

[59] E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[60] E. A. Emerson and A. Prasad Sistla. Symmetry and model checking. In Costas Courcoubetis, editor, *Computer Aided Verification - CAV'93*, volume 697 of *LNCS*, pages 463–478. Springer, 1993.

[61] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In G. Longo, editor, *Proceedings of the 14th Logic in Computer Science Conference*, pages 193–202. IEEE, IEEE Computer Society Press, 1999.

[62] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of Symposia in Appl. Math.*, volume 19, pages 19–32. American Mathematical Society, 1967.

[63] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR 2 User Manual*, 1992-2004. `http://www.fsel.com/fdr2_manual.html`.

[64] A.C.J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, 2002.

[65] N. Francez. *The Analysis of Cyclic Programs*. PhD thesis, Weizmann Institute of Science, 1976.

[66] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002. Special issue in honour of Rod Burstall.

[67] Stephen J. Garland, John V. Guttag, and James J. Horning. An overview of Larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 329–348. Springer-Verlag, July 1993. Tool URL: `http://www.sds.lcs.mit.edu/spd/larch/index.html`.

[68] Amit Goel and William R. Lee. Formal verification of an IBM CoreConnect processor local bus arbiter core. In Giovanni De Micheli, editor, *37th Conference on Design Automation (DAC 2000)*. ACM, June 2000.

[69] E. Goldberg and Y. Novikov. Berkmin: a fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe - DATE*, pages 142–149. IEEE Computer Society, IEEE Press, March 2002.

[70] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.

[71] M. J. C. Gordon. From LCF to HOL: a short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language and interaction: essays in honour of Robin Milner*, pages 169–185. MIT Press, 2000.

[72] M. J. C. Gordon. `HolBddLib`. Hol98 (*Kananaskis* release) documentation, 2001.

[73] M. J. C. Gordon. `HolSatLib` documentation. `http://www.cl.cam.ac.uk/users/mjcg/HolSatLib/HolSatLib.ps`, 2002.

[74] M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.

[75] M. J. C. Gordon. Validating the PSL/Sugar semantics using automated reasoning. *Formal Aspects of Computing*, 15(4):406 – 421, December 2003.

[76] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.

[77] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.

[78] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479, pages 123–142, Canberra, Australia, 1998. Springer-Verlag.

[79] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Oxford University, 2003.

[80] Tobias Hamberger. Integrating theorem proving and model checking in Isabelle/IOA. Technical Report TUM-I, Technische Universität München, 1999.

[81] J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.

[82] J. Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 221–234. Springer, 1996.

[83] J. Harrison. Formal verification at Intel. In P. G. Kolaitis, editor, *18th Annual IEEE Symposium on Logic in Computer Science*, pages 45–53. IEEE Computer Society, 2003.

[84] Klaus Havelund, Michael Lowry, and John Penix. Formal analysis of a space craft controller using SPIN. *IEEE Trans. on Software Engineering*, 27(8):749–765, August 2001.

[85] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, 1995.

[86] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.

[87] C. A. R. Hoare. An axiomatic basis for programming. *Communications of the ACM*, 12(10):576–580, 1969.

[88] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[89] The HOL-4 Proof Tool. Tool URL `http://hol.sf.net`, 2003.

[90] G. J. Holzmann and D. Peled. The state of SPIN. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV'96: 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 385–389. Springer, 1996.

[91] Gerard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 7.2. Technical Report RT-0256, INRIA, February 2002.

[92] Warren Hunt. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994.

[93] J. E. Hurd. Predicate subtyping with predicate sets. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *LNCS*, pages 265–280. Springer, September 2001.

[94] J. E. Hurd. Fast normalization in the HOL theorem prover. In Toby Walsh, editor, *Ninth Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*, Imperial College, London, UK, APR 2002. The Society for the Study of Artificial Intelligence and Simulation of Behaviour. An extended abstract.

[95] J. E. Hurd. Verification of the Miller-Rabin probabilistic primality test. *Journal of Logic and Algebraic Programming*, 50(1-2):3–21, May-August 2003.

[96] Intel Corporation. *Statistical Analysis of Floating Point Flaw*, November 1994. Available from `http://support.intel.com/support/processors/pentium/fdiv/wp/`.

[97] G. L. J. M. Janssen. ROBDD software, October 1993. Department of Electrical Engineering, Eindhoven University of Technology.

[98] Jeffrey J. Joyce and Carl-Johan H. Seger. The HOL-Voss system : Model checking inside a general-purpose theorem prover. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 185–198. Springer, 1993.

[99] J. A. Kalman. *Automated Reasoning with Otter*. Rinton Press, 2001.

[100] Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 489 – 507. Springer, 1988.

[101] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[102] Joseph R. Kiniry and Sam Owre. Improving the PVS user interface. In David Aspinall and Christoph Luth, editors, *User Interfaces for Theorem Provers*, number 189 in Technical Report, pages 101–122. Institut fur Informatik, September 2003.

[103] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In Katrin Seyr Georg Gottlob, Etienne Grandjean, editor, *Computer Science Logic, CSL*, number 1584 in LNCS. Springer, August 1998.

[104] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[105] S. A. Kripke. Semantical considerations in modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[106] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1994.

[107] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer Aided Verification*, volume 697 of *LNCS*, pages 166–180. Springer, June 1993.

[108] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer-Aided Verification*, volume 2725 of *LNCS*. Springer, 2003.

[109] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2002.

[110] Ken Friis Larsen. Personal communication, January 2004.

[111] R. S. Lazic and D. Novak. A unifying approach to data-independence. In Catuscia Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *LNCS*, pages 581–595. Springer, August 2000.

[112] R. S. Lazic and D. Novak. On a semantic definition of data independence. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 226–24. Springer, June 2003.

[113] R. S. Lazic and A.W. Roscoe. On transition systems and non-well-founded sets. *Annals of the New York Academy of Sciences*, 806:238–264, December 1996.

[114] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(6):18–41, July 1993.

[115] J. L. Lions. Ariane 5 flight 501 failure report. Technical Report 33-1996, European Space Agency, 1996.

[116] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University School of Computer Science, 1993.

[117] D. E. Long. The design of a cache-friendly BDD library. In *International Conference on Computer-Aided Design*, pages 639–645. ACM and IEEE Computer Society, 1998.

[118] Donald W. Loveland. Mechanical theorem-proving by model elimination. *JACM*, 15(2):236–251, 1968.

[119] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[120] Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.

[121] Panagiotis Manolios. *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 12, pages 93–111. Kluwer Academic Publishers, June 2000.

[122] J. P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[123] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology and Philosophy of Science*, pages 153–75. North-Holland, 1982.

[124] J. McCarthy. *Computer Programming and Formal Systems*, chapter : A Basis for a Mathematical Theory of Computation, pages 33–70. North-Holland, 1967.

[125] William McCune. *Otter 3.3 Reference Manual*. Argonne National Laboratory, August 2003. Tech. Memo. 263.

[126] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[127] K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 1999.

[128] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[129] K. L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer-Aided Verification*, volume 2725 of *LNCS*, pages 1–13. Springer, July 2003.

[130] T. F. Melham. PROSPER - An investigation into software architecture for embedded proof engines. In Alessandro Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *LNCS*, pages 193–206. Springer, 2002.

[131] Steven P. Miller, David A. Greve, and Mandayam K. Srivas. Formal verification of the AAMP5 and AAMP-FV microcode. In *Proceedings of the Third AMAST Workshop on Real-Time Systems*, 1996.

[132] A. R. G. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[133] A. R. G. Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.

[134] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[135] F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from `http://www.dcs.ed.ac.uk/home/cwb/`.

[136] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[137] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universitat Munchen, 1998.

[138] Olaf Müller and Konrad Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.

[139] M. Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993.

[140] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*. Springer, July 2001.

[141] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966.

[142] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[143] Malcolm Newey. A Z specification of the AMBA high-performance bus. Draft, January 2004.

[144] Jørn-Lind Nielsen. BuDDy - A Binary Decision Diagram Package. Technical report, Department of Information Technology, Technical University of Denmark, 1999. `http://www.itu.dk/research/buddy`.

[145] Aleksey Nogin. Writing constructive proofs yielding efficient extracted programs. In Didier Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2000.

[146] Michael Norrish. Mechanising Hankin and Barendregt using the Gordon-Melham axioms. In *Proceedings of the Merlin 2003 Workshop*, 2003. To appear in 2004.

[147] C.-H. Luke Ong. Model checking Algol-like languages using game semantics. In Manindra Agrawal and Anil Seth, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *LNCS*, pages 33–36, 2002.

[148] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Thomas A. Henzinger Rajeev Alur, editor, *CAV'96: 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 411–414. Springer, July 1996.

[149] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, jun 1992. Tool URL :`http://pvs.csl.sri.com`.

[150] Abelardo Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems*. PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.

[151] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 772–773. Springer, 1988.

[152] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In E. M. Clarke and R. P. Kurshan, editors, *DIMACS Workshop on Computer Aided Verification*, pages 293–320, Providence, RI, 1990. American Mathematical Society, ACM Press.

[153] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[154] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In Pierre Wolper, editor, *Proceedings of Computer Aided Verification*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

[155] A. Riazanov and A. Voronkov. Vampire. In Harald Ganzinger, editor, *Automated Deduction (CADE)*, volume 1632 of *LNCS*, pages 292–296. Springer, May 1999. `http://www.cs.man.ac.uk/~riazanoa/Vampire/`.

[156] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[157] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[158] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *Design, Automation and Test in Europe*, pages 10828–10833, Munich, Germany, March 2003. IEEE Computer Society.

[159] David M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In Warren

A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 3–36. Springer, November 2000.

[160] Mark Saaltink and Irwin Meisels. *The Z/EVES Reference Manual*. ORA, Canada, September 1997. Technical Report TR-97-5493-03d.

[161] H. Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *LNCS*, pages 377–396. Springer, July 2000.

[162] H. Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag.

[163] J. Sasaki. *Extracting Efficient Code From Constructive Proofs*. PhD thesis, Cornell Univeristy, 1986.

[164] Julien Schmaltz and Dominique Borrione. Validation of a parameterized bus architecture using ACL2. In Warren Hunt Jr., Matt Kaufmann, and J. Moore, editors, *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, Boulder CO, USA, July 2003.

[165] K. Schneider and D. Hoffmann. A HOL conversion for translating linear time temporal logic to omega-automata. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 255–272. Springer, September 1999.

[166] C-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report UBC-TR-93-45, The University of British Columbia, December 1993.

[167] Peter Sestoft. Moscow ML. `http://www.dina.dk/~sestoft/mosml.html`, 2003.

[168] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In Catuscia Palamidessi, editor, *Concurrency Theory (CONCUR)*, volume 1877 of *LNCS*, pages 1–16. Springer, 2000.

[169] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 108–125. Springer, November 2000.

[170] M. Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522, pages 82–99. Springer-Verlag, Berlin, 1998.

[171] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26(2):351–360, 1979.

[172] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[173] T. Skolem. Some remarks on axiomatised set theory. In van Heijenoort [189], pages 290–301.

[174] Konrad Slind. Function definition in higher-order logic. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 381–397. Springer, 1996.

[175] Konrad Slind and Michael Norrish. *The HOL System Description*. University of Cambridge Computer Laboratory Automated Reasoning Group, 3rd edition, June 2002. Available from `http://hol.sf.net`.

[176] Fabio Somenzi. Tool URL `http://vlsi.colorado.edu/~fabio/CUDD/`, 2003.

[177] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[178] C. Sprenger. *Deductive Local Model Checking*. PhD thesis, Computer Networking Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.

[179] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[180] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *LNCS*, pages 500–504. Springer, July 2002.

[181] Kong Woei Susanto. *A Verification Platform for System on Chip*. PhD thesis, Department of Computing Science, University of Glasgow, UK, 2004. Private copy.

[182] T. Tammet. Towards efficient subsumption. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction (CADE)*, volume 1421 of *LNCS*, pages 427–441. Springer Verlag, July 1998.

[183] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[184] A. M. Turing. On computable numbers, with an application to the *entscheidungsproblem*. *Proc. Lond. Math. Soc.*, 42(2):230–265, 1936.

[185] A. M. Turing. Checking a large routine. *EDSAC Inaugural Conference*, 1949. Reprinted with corrections and annotations in "An early program proof by Alan Turing", by L. Morris and C. B. Jones, Annals of the History of Computing, 6 (2) pp.129-143 (1984).

[186] T. E. Uribe. Combinations of model checking and theorem proving. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of the Third Intl. Workshop on Frontiers of Combining Systems*, volume 1794 of *LNCS*, pages 151–170. Springer-Verlag, March 2000.

[187] U.S.-Canada Power System Outage Task Force. *Interim Report: Causes of the August 14th Blackout in the United States and Canada*, November 2003. Available from `https://reports.energy.gov/`.

[188] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[189] J. van Heijenoort, editor. *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1967.

[190] C. Weidenbach. SPASS: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.

[191] G. Winskel. A note on model checking in the modal ν-calculus. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 761 – 772. Springer, 1989.

[192] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 70–87. Springer, 2002.

[193] I. Zakiuddin, N. Moffat, C. O'Halloran, and P.Ryan. Chasing events to certify a critical system. Technical report, UK Defence Evaluation and Research Agency, 1998.

[194] Jürgen Zimmer and Michael Kohlhase. System description: The MathWeb software bus for distributed mathematical reasoning. In Andrei Voronkov, editor, *18th International Conference on Automated Deduction*, number 2392 in LNAI, pages 139–143. Springer Verlag, 2002.

Notation Index