VERIFYING SAFETY PROPERTIES OF LUSTRE PROGRAMS: AN SMT-BASED APPROACH

by

George Edward Hagen

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2008

Thesis Supervisor: Associate Professor Cesare Tinelli

ABSTRACT

An important problem in hardware and software design is ensuring a designed system is error-free. Even small errors in a computer system can have disastrous consequences to a project, sometimes costing large amounts of money to correct, or even leading to unexpected and catastrophic system failure.

There are a number of steps one can take to eliminate as many errors as possible. We focus on a set of techniques known as formal methods that are used in computer science to help ensure correct system behavior. In order to minimize the potential for human error and to reduce the time and expertise needed, we seek to use techniques that are highly automatable. We focus on one such approach, an inductive variation of model checking that can be used to verify formally the invariance of properties or produce counterexamples.

One class of systems of particular interest for verification are reactive systems. This is a class of systems that continuously react to their environment in a timely manner. Reactive systems are pervasive in everyday life, ranging from simple thermostats to the controls of nuclear power plants. As a representative language to describe these systems, we look at an established specification and programming language, Lustre.

We have developed a set of techniques based on inductive reasoning and Satisfiability Modulo Theories (SMT) that are automatically able to prove invariant properties of systems described in Lustre. These techniques involve the translation

of a Lustre program and property into formulas of a suitable logic, and then the application of k-induction with improvements such as path compression and abstraction/refinement. This process can be used to prove a property invariant or to provide a concrete counterexample for it that can aid in correcting errors. While these techniques individually have been applied to solve similar problems, we refine and combine them in a novel way to deal effectively with Lustre-based systems with the aid of automated off-the-shelf SMT reasoners. We have implemented these techniques in a new system, Kind, and can experimentally show this is an improvement over the current state of the art.

Abstract Approved: _____
Thesis Supervisor

_____
Title and Department

_____
Date

VERIFYING SAFETY PROPERTIES OF LUSTRE PROGRAMS: AN SMT-BASED APPROACH

by

George Edward Hagen

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2008

Thesis Supervisor: Associate Professor Cesare Tinelli

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

George Edward Hagen

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2008 graduation.

Thesis Committee: _____
Cesare Tinelli, Thesis Supervisor

_____
Hantao Zhang

_____
Jim Cremer

_____
Ted Herman

_____
Steven Miller

_____
Aaron Stump

_____
Jon Kuhl

ACKNOWLEDGEMENTS

I would like to thank my advisor, committee members, parents, and friends for all their support.

ABSTRACT

An important problem in hardware and software design is ensuring a designed system is error-free. Even small errors in a computer system can have disastrous consequences to a project, sometimes costing large amounts of money to correct, or even leading to unexpected and catastrophic system failure.

There are a number of steps one can take to eliminate as many errors as possible. We focus on a set of techniques known as formal methods that are used in computer science to help ensure correct system behavior. In order to minimize the potential for human error and to reduce the time and expertise needed, we seek to use techniques that are highly automatable. We focus on one such approach, an inductive variation of model checking that can be used to verify formally the invariance of properties or produce counterexamples.

One class of systems of particular interest for verification are reactive systems. This is a class of systems that continuously react to their environment in a timely manner. Reactive systems are pervasive in everyday life, ranging from simple thermostats to the controls of nuclear power plants. As a representative language to describe these systems, we look at an established specification and programming language, Lustre.

We have developed a set of techniques based on inductive reasoning and Satisfiability Modulo Theories (SMT) that are automatically able to prove invariant properties of systems described in Lustre. These techniques involve the translation

of a Lustre program and property into formulas of a suitable logic, and then the application of k-induction with improvements such as path compression and abstraction/refinement. This process can be used to prove a property invariant or to provide a concrete counterexample for it that can aid in correcting errors. While these techniques individually have been applied to solve similar problems, we refine and combine them in a novel way to deal effectively with Lustre-based systems with the aid of automated off-the-shelf SMT reasoners. We have implemented these techniques in a new system, Kind, and can experimentally show this is an improvement over the current state of the art.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1 MOTIVATION 1

1.1 Introduction 1
1.2 Formal Methods 2
1.3 Reactive Systems 4
1.4 About This Thesis 6

2 VERIFICATION BACKGROUND 7

2.1 Introduction 7
2.1.1 Definitions 9
2.2 Deductive Verification 11
2.3 Model Checking 12
2.3.1 Temporal Logics 12
2.3.2 ω-Automata 15
2.3.3 Symbolic Model Checking 17
2.4 Bounded Model Checking 18
2.4.1 Completeness 22
2.4.2 Optimizations 25
2.4.3 Limitations 26
2.5 Abstraction 26
2.5.1 Predicate Abstraction 27
2.5.2 CEGAR 27
2.6 k-Induction 28
2.7 Satisfiability Modulo Theories 31
2.7.1 Boolean Satisfiability 31
2.7.2 BMC-specific Optimizations 34
2.7.3 Satisfiability Modulo Theories 36
2.7.4 Common Theories 38
2.8 Summary 38

3 CHECKING SAFETY PROPERTIES OF SYNCHRONOUS DATAFLOW LANGUAGES 40

3.1 Introduction . . . 40
3.2 Reactive Languages . . . 40
3.3 Lustre . . . 41
3.3.1 Recent Versions . . . 46
3.4 Translating Lustre into $\mathcal{IL}$ . . . 47
3.4.1 Example . . . 51
3.5 Checking Safety Properties in Lustre . . . 54
3.6 current and when . . . 57
3.6.1 Clock Inference . . . 61
3.6.2 Relaxed Translation . . . 64
3.7 Summary . . . 69

4 K-INDUCTION OVER LUSTRE . . . 70

4.1 Introduction . . . 70
4.1.1 Related Work . . . 70
4.2 Background . . . 71
4.3 Basic Inductive Procedure . . . 72
4.3.1 Implementation Note: SMT Solver Features . . . 75
4.3.2 Implementation Note: Memory depth . . . 77
4.4 Additional Extensions . . . 77
4.4.1 Quantified Path Restrictions . . . 77
4.4.2 Static Analysis . . . 79
4.4.3 Skipping Steps . . . 84
4.5 Summary . . . 85

5 ALGORITHM EXTENSIONS . . . 86

5.1 Introduction . . . 86
5.1.1 Related Work . . . 86
5.2 Path Compression . . . 87
5.2.1 Implementation . . . 94
5.2.2 when and Path Compression . . . 95
5.3 Abstraction of Lustre Programs . . . 95
5.3.1 Structural Abstraction . . . 96
5.3.2 Basic Procedure with Abstraction/Refinement . . . 99
5.3.3 Combining Abstractions and Path Compression . . . 102
5.3.4 Implementation: Refinement Strategies . . . 103
5.4 Summary . . . 109

6 IMPLEMENTATION AND EXPERIMENTS . . . 110

6.1 Introduction . . . 110
6.2 The Kind System . . . 110

6.3 Experimental Results 113
6.3.1 Abstraction vs. No Abstraction 119
6.3.2 Other Configurations 121
6.3.3 Kind vs. Other Lustre checkers 123
6.3.4 Kind vs. SAL 127
6.4 Summary 128

7 FUTURE WORK 130

REFERENCES 133

LIST OF TABLES

Table

6.1 Kind: abstraction vs. non-abstraction, invalid problems 114

6.2 Kind: abstraction vs. non-abstraction, valid problems 115

6.3 Kind vs. other systems, invalid problems 115

6.4 Kind vs. other systems, valid problems 116

6.5 Kind configurations, invalid problems 121

6.6 Kind configurations, valid problems 122

LIST OF FIGURES

Figure

2.1 LTL semantics defined over suffixes of infinite paths 13

2.2 Bounded path semantics 19

2.3 Basic DPLL algorithm 33

3.1 Example of temporal operators in Lustre 45

3.2 Example Lustre program 46

3.3 Basic Lustre translation semantics 50

3.4 Lustre counter example code 52

3.5 Lustre counter inlined code 53

3.6 Translated Lustre counter example 54

3.7 Fibonacci example 55

3.8 Hidden counter example 58

3.9 Hidden counter example, incorrect translation 59

3.10 Clock inference rules 63

3.11 Relaxed Lustre translation semantics 65

3.12 Hidden counter example, with clock labels 67

3.13 Hidden counter example, corrected translation 68

4.1 Base k-induction algorithm 74

4.2 Lustre counter with aggressively inlined code 83

5.1 Path compression 87

5.2 Base k-induction algorithm with path compression 93

5.3 Structural abstraction of a node 97

5.4 Base k-induction algorithm with abstraction 100

5.5 Path refinement example 106

6.1 Runtimes for Kind with and without abstraction, on hard invalid problems. 118

6.2 Runtimes for Kind with and without abstraction, on hard valid problems. 118

6.3 Kind vs. Rantanplan and Luke on valid and invalid problems. 124

6.4 Kind vs. SAL, respectively on invalid and valid problems. 125

CHAPTER 1
MOTIVATION

1.1 Introduction

When engineers build a computer-based system, be it hardware or software, it is highly desirable to ensure that it will reliably work as intended, with no errors or undesirable side effects. Seemingly trivial errors can have deleterious consequences to a project, sometimes costing large amounts of money to correct, or, if uncaught, even leading to unexpected, complete system failure. The Denver International Airport, for example, suffered from an improperly designed baggage system that was delayed for nearly a year, with efforts to correct it supposedly costing more than the initial cost of the project; to this day the automated luggage system is still not used as intended [82]. The Ariane 5 rocket, flight 501, was destroyed shortly after liftoff, along with approximately $500 million in payload due to a change in specifications that resulted in a simple overflow error in its programming [30, 58]. A software error also resulted in the loss of the NASA Mars Climate Orbiter in 1999 due to improperly converting measurement scales between the Orbiter and the terrestrial control station [59, 73]. Other errors can be represented as vulnerabilities to supposedly secure systems, possibly resulting in expensive or reputation-damaging data loss. Such examples clearly illustrate some of the potential dangers and costs of *not* catching errors during a system's development.

There are a number of additional steps one can take to eliminate as many

errors as possible. We focus on a set of techniques known as *formal methods* that are used in computer science to help ensure correct system behavior.

1.2 Formal Methods

The use of formal methods involve applying a variety of rigorous, logically sound techniques to ensure correct behavior of a system.

A project in development will nearly always have problems at some point. It has been shown that detecting and addressing these problems early in the development cycle can provide significant overall cost savings. While it may seem cheaper or quicker to patch errors after a product has shipped, this is rarely the case — for example the Pentium FDIV led to Intel's open offer to replace affected processors [85]. Formal methods can be applied at nearly any stage in the development cycle, from initial specification to final testing, and as such have additional chances to catch bugs earlier.

For the purposes of this thesis, we are interested in *software* systems, and more specifically in producing *correct* code, or code that conforms to a certain set of specifications. These specifications are represented as a set of *properties* that a program's behavior must conform to. There are a number of formal method techniques used to help ensure correct code is produced, ranging from specification guidelines to annotations that may help ensure a fragment of code behaves properly, to detecting bugs and proving that properties hold.

In order to minimize the potential for human error and to reduce the needed

time and expertise involved, we seek to use formal method techniques that can be (mostly) automated. Two such techniques are formal verification and the detection and production of counterexamples.

The first such group of techniques, that of *verification*, involve mathematically proving a system does or does not exhibit certain desired properties. The second, the production of *counterexamples*, can provide a concrete example of the behavior of a system that leads to a failure, something that can be extremely useful in correcting errors.

Note that there are a number of ways to achieve these goals, with formal methods only being one possible solution. Another common, and orthogonal, approach is traditional testing, where the examiner attempts to generate input/output pairs that lead and demonstrate statistically that the system will behave as expected. This can also produce counterexamples, but generally is unable to prove the correctness of a system. Testing is only able to look at a finite number of specific input/output pairs, and so effort is made to try to choose those likely to produce faults; the choices are then used to actually run the system and the results observed. In the event of a failure, the given input is a counterexample. Formal methods, on the other hand, determine counterexamples as a byproduct of the verification process, as witnesses that a property is not verified. Furthermore they are generally based on a static analysis of the code using symbolic reasoning instead of direct processing of concrete inputs.

1.3 Reactive Systems

One area of particular interest is that of *reactive systems*. This general classification involves systems that run continuously, processing input and typically reacting to their environment in a timely manner. *Embedded systems* are one example that are ubiquitous in modern life and are typically reactive in nature. These range anywhere from simple thermostats to sensor nets and complex health systems. Many reactive systems are safety-critical, where even minor errors are unacceptable. For instance, health-related systems, cars' anti-lock breaks, airplane flight control systems, and controls for nuclear power plants are reactive systems where failure could be disastrous.

Reactive systems are often specified and implemented via languages specifically designed for the task, as opposed to more general computer languages. These languages are often designed to remain simple and to enable easier verification.

The problem of system verification is a difficult and ongoing one, which we intend to address. As it is vitally important to prove that safety-critical systems behave properly we focus on reactive systems in particular, and attempt to prove properties about them that fall into the category of safety properties (invariants). As an example, consider a controller for a typical set of traffic lights. One safety property might be that at least one colored light must be on at all times. Another might be that a green light and a red light cannot both be on at the same time. These are generally more applicable to reactive systems than another major category of properties, fairness properties (stating some event will eventually occur), as designers of reactive

systems often have hard time limits associated with any properties they wish to prove – for example, if an obstacle is detected, the brakes must engage within 0.5 seconds; some indefinite time in the future is not sufficient. As a representative language to describe these systems, we look at an established specification and programming language, Lustre.

Another problem associated with verification (and many other subsets of formal methods) is that of accessibility. It often requires a fair amount of experience to use the various formal methods techniques, and possibly a large degree of user interaction as well. So we focus on techniques that are highly automatable, specifically an inductive variation of a technique called *model checking*.

We have developed a set of techniques based on inductive reasoning that are able to automatically prove safety properties for systems described in Lustre. This includes (1) the translation of the system from the Lustre language into a suitable logic $\mathcal{IL}$, and then the application of k-induction with (2) path compression and (3) abstraction / refinement to prove the invariance of a property or to provide concrete counterexamples that can aid developers in correcting errors. While these techniques individually have been applied to solve similar problems, we refine and combine them to effectively deal with Lustre-based systems. We can show these are improvements, as we have implemented these ideas in a system called Kind that outperforms other, similar systems in these tasks. With Kind, the user needs only to provide a Lustre program and the property to be checked; no other user interaction is required.

1.4 About This Thesis

In the remainder of this thesis, we will be looking at the Lustre language as a specific example of a synchronous language used to describe reactive systems, and focusing on a particular subset of formal methods known as model checking. We have synthesized a number of existing techniques as an overall approach that advances the state of the art in the verification of Lustre programs.

Chapter 2 covers some background in the field, Chapter 3 describes the language Lustre as well as a translation into the logic we shall be reasoning with, Chapters 4 and 5 explain the basic algorithm we are utilizing, as well as a number of variations, and Chapter 6 describes some of the experimental results we have achieved while exploring these issues, and Chapter 7 details some potential avenues of extending this research.

CHAPTER 2
VERIFICATION BACKGROUND

2.1 Introduction

System verification can take many forms. One such option is to provide *deductive* proofs that a system behaves in a certain way, possibly with the aid of an automated theorem prover. Unfortunately, due to the undecidability of the logics involved, such tools often require experienced human input in order to perform effectively, either to encode background knowledge that the theorem prover may not have access to or be able to deduce (such as system invariants), or else manually direct the search of the theorem prover towards more promising areas.

Another option is that of (finite state) *model checking*. This highly automatable technique involves building a model of the system and a model of a user-supplied specification and ensuring the system model complies to the specification. There is still some hands-on work involved in encoding system and spec in a manner that a model checking tool can understand, but after that the process is completely automatic. Assuming the problem will fit within memory and time constraints, the checker will produce either confirmation that the model conforms to the specification or else evidence of a discrepancy, possibly a counterexample that can be useful in eliminating system errors.

A new variant of this technique is inspired by recent advances in tools with the ability to efficiently solve propositional satisfiability problems (or SAT, the "classic"

NP-complete example). *Bounded model checking* is a method involving checking potential executions of the model in an incremental fashion against the negation of a specification by encoding them as propositional satisfiability formulas. If the model's executions conform to the specification property up to some bounded number of steps, the resulting propositional formula will be unsatisfiable. If this is the case then the bound can be increased and a longer run examined. If the formula proves to be satisfiable, then a concrete counterexample can be extracted from the resulting logical model, providing a trace of system states leading to the error.

These model checking variants can also benefit from the use of *abstraction techniques*, which intelligently simplify the problem to reduce its complexity. This can take the form of abstracting away unimportant features or replacing complex subproblems with more manageable versions.

A fourth option is to use *inductive* reasoning to prove a system conforms to a specification. This can be seen as an extension to bounded model checking, and keeps many of its strengths while also addressing at least one of its primary weaknesses. While bounded model checking excels at providing short examples of system problems, it can have difficulty proving that a system really does conform to a specification; inductive approaches, on the other hand, can often provide such assurance.

The advances in SAT that inspired bounded model checking and have been useful in checking systems inductively can also be lifted to a more powerful form of solvers based on *Satisfiability Modulo Theories* (SMT). With these solvers it is possible

to use more natural translations for systems, have fewer limitations on specifications, and often still have significant performance gains over previous tools.

In the rest of this chapter we will introduce some basic concepts, and review a number of model checking techniques. This will include a brief background on temporal logic, several flavors of model checking, and some basic ideas on the use of abstraction. We will finish with a brief overview of some of the advances in the underlying SAT and SMT approaches used in some of these techniques.

2.1.1 Definitions

Formally, a base logic includes some notion of atoms, with formulas defined in terms of Boolean connections of atoms. This base logic definition can therefore include basic propositional logic, where atoms are simple propositions, as well as more complex logics, such as first order logic, where the atoms include predicates and quantified formulas. Satisfiability, entailment, and validity are defined in terms of the structure of the base logic. A formula ϕ is *satisfiable* (SAT) if there exists an assignment of values (*interpretation*) such that ϕ evaluates to true. Such an assignment is called a *model*. A formula ϕ *entails* ψ ($\phi \models \psi$) if for every interpretation in which ϕ evaluates to true, ψ also evaluates to true. A formula ϕ is *valid* if it evaluates to true for all possible interpretations.

A *system* here refers specifically to a computational artifact, either a software program or logical hardware circuit. The particular configuration of an active system at a given instant in time, for example the values of a program's variables, is a *state*

of the system. Depending on the cardinality of the states in a system, it may be termed *finite state* (for example digital systems with bounded values) or *infinite state* (such as continuous or analog systems with unbounded values).

To reason about real-world systems it is generally necessary to produce an abstract model that distills their essences into a form that is easier to manipulate. A real system is modeled mathmatically by a *transition system* defined as a relation over states. This is formally represented as a *Kripke structure*, which can be seen as a simple form of automaton. Given a set of atomic propositions P in the base logic, a Kripke structure $M = (S, I, T, L)$ is a structure with a set of states S, with an *initial set of states* $I \subseteq S$, a *translation relation* $T \subseteq S \times S$, and a set of *state labels* $L : S \to 2^P$. To simplify reasoning about the system, it is assumed that each state in S has at least one successor defined in T. If the original system would reach a state with no outwards transitions, the Kripke structure instead has a transition from that state to itself.

A *path* is a sequence of states in M that obey the transition relation T. If the first state in a path is an initial state (belonging to I), that path is called *initialized*. If no initialized path reaches some state s then that state is *unreachable*, and can be safely discarded.

Properties indicate some constraint on the system state or execution. Properties are generally expressed as boolean combinations of predicates; a property *holds* in a state if it is satisfied by that state, for some formal notion of satisfiability in a state.

The properties we wish to verify often fall into one of two categories: safety properties and liveness properties. A *safety property* of a system is an invariant, something that holds in every reachable state of the system. These are generally of the notion that "nothing bad will ever happen" (e.g. a null pointer will never be accessed). A *liveness property* is one that will eventually hold in any execution. These are often of the notion that "something good will eventually happen" (e.g. the process will terminate correctly). Both of these can be checked with model checking techniques, though for the types of systems we will be concerned with, safety properties are usually more significant.

A set of desired properties is often grouped as a *specification* for the system. A system satisfies its specification if every possible execution of the system satisfies the specification.

2.2 Deductive Verification

There are several methods of proving the validity of properties of systems. One might translate the system M and property P into first- (or higher) order formulas and use something such as a resolution-based theorem prover to interactively prove the property holds. Assuming the system definition is translated automatically into an appropriate formula, this allows the user to express the properties in question in a formal yet still somewhat natural way. There is no limitation on the data types used to describe the system, provided they can be described by the logic in question.

This technique is perfect for theorem provers, which are designed to prove

the validity of formulas. However, if there is an execution where the property does *not* hold, then the formula being checked will be invalid, and it is useful to have a counterexample when trying to correct errors. Additionally, it may be difficult or impossible to extract a particular incorrect execution from the invalid result. Proofs are often in terms of rules applied, axioms, and (possibly quantified) formulas; the theorem prover may not have an internal concept of a program execution and may not be able to provide meaningful concrete values in the event of an invalid result.

In addition to the difficulty of extracting incorrect execution examples, deductive tools, especially ones utilizing the more powerful high-order logics, may require extensive user interaction to deal with the infeasibility of the proof search. User expertise may be a significant factor in the efficient use of these tools.

2.3 Model Checking

The primary alternative to the deductive approach is to attempt to *dis*prove a property through the technique of *model checking*, which is formally stated on some kind of temporal logic.

2.3.1 Temporal Logics

Temporal logics are examples of modal logics: versions of some base logic system such as first order logic with additional interpreted operator symbols, in this case dealing with time. These logics are used to represent the behavior of discrete sequential systems in a formal and concise manner, especially to describe specifications. Some examples of the modal operators commonly used include *next* (**X**, alternately

$$
\begin{array}{lll}
\pi \models p & \text{iff} & p \in L(\pi(0)) \\
\pi \models \neg f & \text{iff} & \pi \not\models f \\
\pi \models f \wedge g & \text{iff} & \pi \models f \wedge \pi \models g \\
\pi \models f \vee g & \text{iff} & \pi \models f \vee \pi \models g \\
\pi \models \mathbf{X}\ f & \text{iff} & \pi \models f \\
\pi \models \mathbf{G}\ f & \text{iff} & \text{for all } i \geq 0, \pi_i \models f \\
\pi \models \mathbf{F}\ f & \text{iff} & \text{for some } i \geq 0, \pi_i \models f \\
\pi \models f\ \mathbf{U}\ g & \text{iff} & \text{for some } i \geq 0, \pi_i \models g \text{ and for all } j, 0 \leq j \leq i, \pi_j \models f \\
\pi \models f\ \mathbf{R}\ g & \text{iff} & \pi_i \models g \text{ if for all } j < i, \pi_j \not\models f
\end{array}
$$

Figure 2.1: LTL semantics defined over suffixes of infinite paths [11]. π is path $\pi = s_0, s_1, s_2, ...$ π_i is the suffix of π starting at s_i (so π_0 is π itself) and $L(\pi(0))$ is the set of property labels associated with the first state in path π. Note that the operators **F** and **G** are duals, or $\neg\mathbf{F}\ f \equiv \mathbf{G}\neg f$, similarly with **U** and **R**.

$\bigcirc$), which informally means a property holds in the next state; *always* (**G** or $\Box$), meaning a property holds in the current state and all future states; *eventually* (**F**, or $\diamond$), meaning that a property will hold in some future state; and the binary operators *until* (a **U** b) indicating that property a holds in all states up to one where b holds, and *release* (a **R** b), where b holds until the first point at which a holds. Although there are a number of temporal logics (including computation tree logic — CTL and its more expressive extension CTL* [38]), *linear temporal logic* (LTL) is a particularly popular logic for defining system properties in sequential (or interleaving sequential) systems. This is generally built on top of propositional or first order logic, with the addition of the modal operators mentioned above.

LTL takes the view that the execution of a system can be represented by sequences of states, which are called *execution paths*, or just *paths*. In a path, time is represented abstractly and discretely, with each transition between states representing

a new unit of time. LTL formulas therefore speak of properties in sequences of states, or an *evolution* of states. An LTL formula can be a formula in the base logic, or one including Boolean connectives or modal operators. Semantically a notion of satisfaction in LTL is given as a satisfaction relation $\models$ between paths and LTL formulas. The relation $\models$ is defined inductively over the suffixes of infinite paths, as seen in Figure 2.1 [11].

Linear temporal logic takes the view that transitions are deterministic, and is often used to express models of systems that use an interleaving view of concurrency. A related temporal logic, computation tree logic, takes a branching view of time — one where there are several possible futures. This can be more appropriate for expressing specifications dealing with nondeterminism, such as the existence of a path where a property holds.

CTL provides a notion of satisfaction as a relation $\models$ between systems (as opposed to LTL paths) and CTL formulas. CTL is formed of *state formulas*, and *path formulas*, concerning properties of each, respectively. Modal operators only appear in path formulas, and are, in essence, explicitly *quantified* versions of the LTL modal operators, meaning each modal operator either applies to all possible paths starting from a state ($\mathbf{AG}, \mathbf{AF}, \ldots$) or at least one possible path starting from a state ($\mathbf{EG}, \mathbf{EF}, \ldots$). For example, for a transition system M, $M \models \mathbf{AG}f$ iff f holds on all states of all reachable paths of M, and $M \models \mathbf{EG}f$ iff f holds on all states of some reachable path of M. The duals of CTL operators are as in LTL, but also incorporate the opposite quantification, so $\neg\mathbf{EF}f \equiv \mathbf{AG}\neg f$. LTL formulas can be seen to be

implicitly universally quantified over paths.

In bounded model checking (Section 2.4) we generally we wish to check properties universally (to ensure that a safety property holds for all executions), so in CTL we actually check if $M \models \mathbf{EF}\neg f$. If the Boolean translation of this is not satisfiable then we know the property $M \models \mathbf{AG}f$ holds. If we can find a witness path with a state satisfying $\neg f$, we have a counterexample to the property.

Both LTL and CTL are fragments of *CTL**, which may have freely nested **A** and **E** quantifiers over both paths and states. A comparison of the expressiveness of CTL*, CTL, and LTL appears in [26]. CTL* itself is a fragment of the μ-Calculus, a particularly expressive, but not very human-readable, logic that expresses systems in terms of fixpoints — see [68].

2.3.2 ω-Automata

An alternative way of expressing finite-state systems and their specifications is through the use of infinite-word automata, or *ω-automata*. An automaton's *language* is the set of all words it accepts. (As we are dealing with reactive systems, ω-automata are again used to avoid the complexity of dealing with termination issues, and are very similar to Kripke structures). In essence an accepted word for such an automaton corresponds directly to a reachable path in LTL.

One specific class of ω-automata are *Büchi automata*. A Büchi automaton $B = (\Sigma, S, I, T, L, F)$ consists of a *finite alphabet* Σ, a finite set of *states* S, a set I of starting or *initial states*, a *transition relation* T between pairs of states, a *labeling* L of

states to the alphabet, and a set F of *accepting states*. A path for a Büchi automaton can be represented as a word over Σ. An infinite word is *accepted*, representing a legal execution, if it encounters an accepting state infinitely many times. In other respects, Büchi automata are similar to other automata. There is a translation algorithm from LTL specifications to Büchi automata such that a word is in the specification — a path is allowed — iff it is accepted by the automaton (presented on pp 156-164 of [66]). This process is used in a number of flavors of model checking.

For example in [77], Vardi and Wolper introduce an exponential-time model-checking algorithm that checks a system against an LTL specification by combining automata. The idea is that both the system and the negation of the specification are represented as Büchi automata, the first accepting infinite words that represents the system's possible executions, the second accepting words that describe paths that satisfy the (negated) specification. These two automata are then intersected and the resulting automaton is examined. If it only accepts the empty language, then the system satisfies the specification, otherwise there is a discrepancy. This *automata-theoretic* model verification is the basis for Bell Lab's Spin system [52], one that specializes in asynchronous programs. The automata-theoretic approach explicitly necessitates the enumeration of all states in the system, a task that can cause a problem's complexity to quickly become unmanageable.

2.3.3 Symbolic Model Checking

Most present-day model checkers do not build and check automata, but rather they encode the transition systems symbolically through canonical representations of Boolean formulas called *binary decision diagrams* (BDDs). A BDD is a directed, acyclic graph where each non-leaf node represents a variable, with its outward edges representing the assignment of truth values to that variable. All redundant nodes are eliminated during construction of the graph. There are two possible leaf nodes, with values of either $true$ or $false$. BDDs can be merged or modified fairly efficiently, and the test for satisfiability is trivial: a BDD is satisfiable iff the $true$ node is part of the graph. If this is the case, a linear traversal of the graph from said leaf to the root provides a partial model of the formula encoded by the BDD. The symbolic state encoding and use of BDDs quickly enabled model checkers to handle systems orders of magnitude larger than they could using the basic automata- theoretic approach, encoding 10^{20} states or more [15, 16].

In this sort of symbolic model checker, the states are encoded as vectors of Boolean variables, and their transitions are encoded as a BDD. In [27] Clarke, Emerson, and Sistla describe a linear-time process that explicitly searches the state-space and compares it against a CTL specification. This is expanded in symbolic model checking to use the BDD representations of the inputs, which are then iterated over through a series of fixpoint operations, also represented by BDDs (a version using LTL is presented in Chapter 6 of [23]). While this method is quite useful, it does still need to encode and manipulate the entire state space, so an exponential state

explosion is still a potential problem. It is therefore desirable to look for techniques that do not require the exploration of the entire system.

2.4 Bounded Model Checking

The basic idea behind *bounded model checking* (BMC) [11] is quite simple: check the validity of the executions of a finite-state system incrementally by looking at executions of increasing length. We examine a system's executions in this manner until either an error is detected or we reach a certain limit on the execution length. This process is completed through *unwinding* the transition system, or calculating its paths by following the system's transition relation a certain number of steps (say, k steps). We then check to see if an arbitrary property holds in the unwound paths; if the property does hold, then we can increase k to $k + 1$ and repeat the process. If the property does not hold, then we can extract a counterexample.

Previously we had looked at systems represented as Kripke structures where all paths are of infinite length, but the paths we examine in BMC are a *finite prefix* of these infinite paths. Symbolic model checking encodes an entire system into a BDD, yet bounded model checking only examines finite paths in this system. How is it possible to reconcile these two?

First note that Kripke structures utilizing a base Boolean logic all have finitely many states, but their paths are infinitely long. Therefore all paths in a Boolean logic Kripke structure must contain cycles; each path must revisit some state or states infinitely many times. When defining the semantics of a *bounded model*, it is

$\pi \models_k^i p$ iff $p \in L(\pi(i))$
$\pi \models_k^i \neg p$ iff $p \notin L(\pi(i))$
$\pi \models_k^i f \wedge g$ iff $\pi \models_k^i f \wedge \pi \models_k^i g$
$\pi \models_k^i f \vee g$ iff $\pi \models_k^i f \vee \pi \models_k^i g$
$\pi \models_k^i \mathbf{G}\ f$ is $false$ if there is no cycle within bound $[i,k]$, otherwise iff $\pi \models_i \mathbf{G}\ f$
$\pi \models_k^i \mathbf{F}\ f$ iff $\exists j, i \leq j \leq k$ and $\pi \models_k^j f$
$\pi \models_k^i \mathbf{X}\ f$ iff $i < k$ and $\pi \models_k^{i+1} f$
$\pi \models_k^i f\ \mathbf{U}\ g$ iff $\exists j, i \leq j \leq k . \pi \models_k^j g$ and $\forall n, i \leq n < j . \pi \models_k^n f$
$\pi \models_k^i f\ \mathbf{R}\ g$ iff $\exists j, i \leq j \leq k . \pi \models_k^j f$ and $\forall n, i \leq n < j . \pi \models_k^n g$

Figure 2.2: Bounded path semantics [9]. All formulas are assumed to begin in negated normal form, with negations having been pushed in to only apply to atoms. $\pi \models_k^i f$ means that, for an LTL formula f and a path segment π_i starting at position i and with bound k ($k \geq 0$), $\pi_i \models f$. $L(\pi(i))$ indicates the labels for the ith state on path π.

important to note if the prefix of a path π contains a cycle. If it does, then this prefix effectively holds the same information as the (infinite) unbounded path defined by some LTL formula. In this case then, for an LTL formula f, $\pi_i \models f$ iff $\pi_i \models_k f$ (f holds on the suffix of path π starting at i iff f holds on the segment of path π starting at i and ending with bound k).

If it is not the case that π contains a loop within bound k, then we need to define new *bounded semantics*; see Figure 2.2 [11, 9]. In the absence of a cycle, note that **G** and **F** are no longer duals, nor are **U** and **R**. Also note that **G** formulas are conservatively considered false. However, if the bound k is sufficiently large, then $\pi \models f$ (under the unbounded semantics) iff $\exists k$ such that $\pi \models_k f$ (under the bounded semantics).

So given these semantics, it is possible to translate both a Kripke structure M and safety property p at a given bound k from LTL into a formula Φ_k in the base

logic that is satisfiable iff f holds under the k-bounded semantics of M. Φ_k is the conjunction of three subformulas: one representing the unwound structure M, one determining the presence of loops, and one representing the unwound LTL formula p. Traditionally this is done using a Boolean base logic and the *negation* of the property, allowing the question of whether the property holds in the model to be treated as a satisfiability problem: if Φ_k is satisfiable, then there is a point within bound k where p does not hold for M — a counterexample.

The Kripke structure M is unwound into the formula $[M]_k$ from its initial states by:

$$[M]_k := I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right)$$

with $I(s_0)$ being the representation of the initial state, and $T(s_i, s_{i+1})$ being the transition relation from state s_i to s_{i+1}.

The looping condition predicate ${}_jL_k$ is true iff there is a transition from state s_k to state s_j. The loop condition L_k is true iff there exists a transition from state s_k to state s_j, with $j \leq k$:

$$L_k := \bigvee_{j=0}^{k} {}_jL_k$$

The property p can similarly be translated into a base-logic formula ϕ_i. To preserve soundness of the process, properties of the form $\mathbf{G}f$ are considered *false* if the bound does not contain a complete loop or if the translated formula should refer to a position i outside of bound k.

The final formula then is:

$$\Phi_k := [M]_k \wedge \left((\neg L_k \wedge \phi_k) \vee \bigvee_{j=0}^{k} ({}_j L_k \wedge \phi_k) \right)$$

The above is generally used to check safety properties — or, more accurately, their negations.

For liveness properties, the following translation can be used [9]:

$$\Lambda_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \rightarrow \bigvee_{i=0}^{k} p(s_i)$$

where $M \models \mathbf{F}p$ iff there exists some k such that the formula Λ_k is valid. So for our purposes if we have such a k, we can verify that $\neg\Lambda_k$ is unsatisfiable (unlike the above we are searching for an *absence* of witnesses).

A similar technique can be used for *equivalence checking* [10], where two systems are compared. In this case we would unwind the first system and the negation of the second, with satisfiable models representing the discrepancies. Such a technique is used in the CBMC program to verify Verilog hardware designs against a C prototype [25].

An alternative to the above syntactic translation harkens back to earlier model checking approaches — the translation of the problem into a Büchi automaton. Clark, *et al.* call this the *semantic translation* [24]. In basic model checking the system and negation of the property are translated into Büchi automata, which are then combined. If the resulting automaton accepts an empty language, then the property holds. The semantic translation version of BMC does the exact same thing, but then

it checks the resulting automaton for any accepting loops (meaning the automaton accepts some non-empty words) by unrolling it. Therefore, it is possible to check the automaton's language incrementally, without necessarily encoding its entire state-space.

In practice, it has been found that the symbolic model checking and bounded model checking approaches to model checking can complement each other well — for Boolean logics, each can solve problems that the other cannot. BMC can handle larger systems and it can be guaranteed to find the shortest counterexamples, or executions where a property does not hold, but its efficiency degrades as the bound increases. BDD-based approaches tend to do better at finding "deep" bugs, especially, and are complete by construction — given adequate resources they are guaranteed to terminate, though the state space explosion limits their usefulness in large problems.

2.4.1 Completeness

It is important to stress that bounded model checking is generally used to find bugs in a system, and it is desirable to find counterexamples with the shortest possible run path. When looking for a witness, the basic BMC algorithm begins with a bound of $k = 0$ and performs the satisfiability check. If no counterexample is found, it then increases k and repeats until it does find a counterexample. If no witness exists (meaning $M \not\models \mathbf{E}p$, in CTL*) then the process may not terminate. So in general we only have a semi-decision procedure.

Therefore it is common practice to run the BMC algorithm while incrementally

increasing the bound of k until some arbitrary maximum upper bound is reached. While this process is indeed often good for finding errors in the system (and for otherwise undecidable problems, being able to arbitrarily stop the process can be seen as a plus), unless the maximum upper bound is sufficiently large, this technique is not sufficient to verify that the system satisfies the property. For complete verification it is necessary to establish a *completeness threshold* that ensures the bound k is sufficient to include all applicable paths.

Trivially for finite Boolean systems, the completeness threshold is at most 2^n, with n being the number of state variables. This estimate is often, however, overly large, and does not take into account any unreachable states. Therefore it is desirable to compute tighter bounds for a problem. When checking for the existence of properties that are not nested (for example, $\mathbf{EG}p$ or $\mathbf{EF}p$ in CTL) it is necessary to visit all reachable states, the minimum number of steps being the *reachability diameter* of the model, or the longest path in the set of minimal paths to any reachable state. Essentially this amounts to unwinding the model and checking to see at what point no new states have been reached. This can be calculated via appropriate graph algorithms, or by evaluating certain quantified Boolean formulas [56, 9]. The *recurrence diameter* for reachability is a more easily computed (over) approximation of the reachability diameter, where we instead calculate the longest loop-free path [11]. Both of these calculations are generated via inductive calculations in [70]; we provide a simplified approximation of this process in Section 5.2.

When checking a liveness property, one of the form $\mathbf{F}p$, there is also the pos-

sibility of incompleteness. As above, it is possible to attempt to calculate a maximal bound for k, but here we may be beneficial to use a different technique. In this case we want to know if either $\mathbf{F}p$ holds or if its negation holds. Because one of these must be true, we can simply interleave the two checks, stopping when we get a favorable result from either.[1]

In [2], the authors present a termination check for fairness constraints (of the form $\mathbf{FG}\neg p$). In the event that the property is unsatisfiable at bound k, it is possible to check a certain set of conditions. Once they become unsatisfiable then the property is known to hold. First check

$$\left(\bigwedge_{i=0}^{k+1} T(s_i, s_{i+1})\right) \wedge \left(\bigwedge_{0\leq i<j\leq k+1} (s_i \neq s_j)\right) \wedge p(s_{k+1}) \wedge \bigwedge_{i=0}^{k} \neg p(s_i)$$

Once this becomes unsatisfiable (at some bound m), the solver then begins to check

$$\left(\bigwedge_{i=0}^{k+1} T(s_i, s_{i+1})\right) \wedge \left(\bigwedge_{0\leq i<j\leq k+1} (s_i \neq s_j)\right) \wedge \neg p(s_k) \wedge p(s_{k+1})$$

starting at $k = m$. One this also becomes unsatisfiable, it continues to increase k, checking

$$I(s_0) \wedge \left(\bigwedge_{i=0}^{k} T(s_i, s_{i+1})\right) \wedge \left(\bigwedge_{0\leq i<j\leq k+1} (s_i \neq s_j)\right) \wedge p(s_k)$$

When the third has become unsatisfiable, at some n, it is possible to compute a bound $k' \leq m + n - 1$ beyond which $M \models \mathbf{FG}\neg p$ will never hold, thereby giving us

[1]It may be simpler to determine that the property never holds than that it eventually holds, so this method may not be as beneficial for the safety case.

a concrete stopping point.

For LTL formulas translated into empty Büchi automata the completeness threshold is somewhat easier to compute. Let l be the longest loop-free path from the initial state, and r_1 be the reachability diameter of the automaton and r_2 be the reachability diameter of the automaton from its initial state. The completeness threshold in this case is $\min(l + 1, r_1 + r_2)$. In a fully-connected automaton, for example, the completeness threshold would be 2, no matter how many states there were. In general, however, complete BMC from LTL is a doubly exponential algorithm — k may be exponential in the number of state variables, and for each k there is a SAT problem which itself may be exponential. This is the case even though there are singly-exponential model checking algorithms for LTL [24].

2.4.2 Optimizations

One optimization in traditional model checking is the *cone of influence* reduction. In this optimization variables representing state are linked in a dependency graph with roots being the variables representing the property to be checked. Any variables representing states not connected to this graph are said to be outside of the cone of influence, and as such will not be impacted by the property; they can be safely removed. In [12] the authors define a *bounded cone of influence* (BCOI) reduction, a variation of the cone of influence that is incremental with the bound depth k.

Other optimizations often involve specializations to the satisfiability solver that is used to check the BMC formula Φ (see Section 2.7.2).

2.4.3 Limitations

While it has desirable features, such as the ability to provide concrete counterexamples and high degrees of possible automation, model checking has a number of limitations. One of the most significant is that model checking is limited to systems utilizing finite data types. Both bounded and unbounded model checking can only verify systems via exploration of the entire state space, something generally not possible with infinite data types. Similarly, properties themselves must be propositional in nature, a limitation in expressiveness. Additionally, the translation from the base problem into a Boolean formula (for symbolic and bounded model checking) can be rather complex, requiring some form of unique encoding of all states, with the final formula Φ often being difficult for a human to interpret.

2.5 Abstraction

To some extent, abstraction in some form is used with all verification techniques. At the very least the system will be represented as some form of mathematical construct that models (and generally simplifies) the actual system structure and behavior, especially with physical systems. Certain model checking techniques, however, rely more extensively on abstraction.

In these cases, algorithms generally follow an abstraction/refinement paradigm, where first a *conservative*[2] abstraction of the system is generated and used for verification, and if it fails, it is refined, or made less abstract. This process is repeated

[2]In the sense that the abstraction contains a superset of the executions of the original system

as necessary, either working from a single abstract model that is constantly being refined, or generating new abstract models and refining them in succession.

2.5.1 Predicate Abstraction

Predicate abstraction, at its heart, is an attempt to simplify the systems that are sent to a model checker, either to reduce the workload on the model checker or to check more elaborate systems. Most model checkers are propositional in nature, so it is necessary to reduce more complicated systems to that format. Aspects of the system to be abstracted are replaced by predicates in the formula sent to the (Boolean) model checker, while the interpretation of these predicates is typically handled by a theorem prover. So a new problem for abstraction-based systems is that the model checker may make assignments to the predicates that are not reachable, something that must be caught by the theorem prover. The theorem prover is often also used for determining new abstraction predicates that may be added in the verification process.

2.5.2 CEGAR

Counterexample-guided abstraction and refinement (CEGAR) is a general method of refining abstractions based on analysis of inconsistencies between the abstract and concrete models of the system. It follows the general abstraction/refinement paradigm, but in this case, if the abstract model indicates the presence of a counterexample that is inconsistent with the concrete model (for example, it includes unreachable states), then it should be possible to refine the abstraction so as to eliminate that counterexample. Typically this is done by analyzing the conflict generated

from the conjunction of the potential abstract counterexample and the concrete model [28]. One could simply refine the variables mentioned, or perform some deeper analysis in the hope of ruling out additional, similar false counterexamples in a single step [51]. It is also possible that a spurious counterexample may trigger a more elaborate process, such as re-formulating the entire abstraction [45].

An alternative form of abstraction refinement comes from using so-called *Craig interpolants* [61, 62]. While the intention is similar to that of CEGAR, a different method of analysis, based on proofs of inconsistency, is performed to determine the refined formula, requiring a solver that supports interpolant generation.

2.6 k-Induction

In comparison with deductive approachs, model checking includes some advantages and disadvantages. Primary among its advantages are that it is a fully automatable process, and the generation of counterexamples is usually an immediate result of its process. Additionally it is often faster than deductive approaches, as it tends to represent systems with more limited logics.

As an alternative to deductive verification and model checking, it is also possible to take an inductive approach. As we are dealing with formulas describing a system M, let $T(S, S')$ be a formula encoding the transition relation of M such that $T(S, S')$ holds for two states S and S' iff S' is a successor of S in the system. Let T_i denote $T(S_{i-1}, S_i)$. Similarly let $P(S_i)$ be a formula encoding a property such that $P(S_i)$ is true iff the property holds for a state S_i, and let P_i denote $P(S_i)$. Let $I(S_i)$

be a formula that is true iff state S_i is initial, and I_0 denote $I(S_0)$. Given this, it is rather straightforward to perform temporal induction against the unrolling of paths of length k. As a base case, simply first take I_0 and prove the property is satisfied by the initial states:

$$I_0 \models P_0$$

If this base case holds true, take the assumption that P holds for some state S_n and prove it holds for the next state as well:

$$T_{n+1} \wedge P_n \models P_{n+1}$$

If this inductive step case holds as well, then the property holds for all reachable states, indicating its validity.

If the base case does not hold, then it should be possible to extract a counterexample from the invalidity proof. If the step case does not hold, then we cannot conclude anything about the property in question.

Unfortunately, this last possibility often arises when attempting to verify real systems. It is possible to strengthen the invariant being checked, however, allowing more definitive answers. This often requires human intervention, but in many cases it is possible to use a fully automatable method of strengthening called *k-induction* [70, 13, 34, 37, 1].

K-induction strengthens the formulas in question by looking at progressively larger windows of the system execution paths. One begins as with standard induction above, but if an inconclusive result is obtained, the formulas are extended and induction is attempted again, increasing an index value k.

The base case for k-induction is

$$I_0 \wedge T_1 \wedge \cdots \wedge T_k \models P_0 \wedge \cdots \wedge P_k$$

while the strengthened inductive step case is

$$T_{n+1} \wedge \cdots \wedge T_{n+k+1} \wedge P_n \wedge \cdots \wedge P_{n+k} \models P_{n+k+1}$$

where $k \geq 0$. Again, an invalidity in the base case indicates a reachable counterexample originating from the initial state. An entailment in both the base case and inductive step indicates that P holds in all reachable states, a successful verification. An invalidity in the inductive step may only indicate a need to strengthen, so k is incremented and the base and step cases are again checked. In the case of finite systems, this algorithm can be made complete with the addition of path compression (see Section 5.2), as we will eventually reach a k that encompasses all reachable states of the system.

This method does combine some of the strengths of both deductive verification and model checking. It is fully automated and can use fast SAT-based techniques. Also the base case checks of k-induction are essentially bounded model checking, with the accompanying advantages. But with the induction check it is possible to actually prove a property valid for a system before exploring its entire state space, something not possible in standard BMC.

2.7 Satisfiability Modulo Theories

2.7.1 Boolean Satisfiability

When the base logic is propositional, model checking, BMC, and k-induction can all be reduced to determining the satisfiability of a single Boolean formula. A number of highly effective techniques have been developed to solve SAT problems, many based on improvements on a simple branching search algorithm.

The SAT problem is, given a Boolean formula f including variables $V = v_1, .., v_n$, to find an assignment of truth values to the variables that satisfy the formula, or establish that no such assignment exists. BDDs can be used to solve such problems, as in symbolic model checking, but typically less memory-intensive techniques are employed. Systems that use such techniques are called *satisfiability solvers* (*SAT solvers*), and fall into one of two categories: incomplete or complete. Incomplete solvers generally use local or probabilistic search methods, and can be extremely fast, though they cannot prove unsatisfiability. Complete solvers are therefore more desirable for verification purposes; these systems generally involve some sort of backtrackable search.

Almost all modern complete solvers perform this search on conjunctive normal form formulas (CNF) using a variation of the DPLL algorithm by Davis, Putnam, Logemann, and Loveland [32, 31]. This algorithm is based on case splitting, and essentially traverses the semantic tree of possible variable assignments in a depth-first fashion (with backtracking), pruning certain branches that falsify the formula. An example of the basic algorithm can be seen in Figure 2.3. The algorithm views

the formula as a set of disjunctions of literals (*clauses*), and keeps track of a pool of unassigned literals in the formula, choosing one in each iteration of the loop, and removing that literal and its negation (if present) from the pool. When this pool is empty, the algorithm has made a satisfying assignment for all literals in the formula, and terminates. Each such decision splits the search space into two regions, one branch where the literal is asserted, and one where its negation is asserted. Once a literal is chosen, it is asserted true and this change is propagated through the rest of the formula. Any clauses containing the literal are marked true and discarded, and the *negation* of the literal is removed from all remaining clauses. If a clause becomes empty of all literals due to Boolean propagation, then the algorithm has detected a *conflict* in the literal assignment, and must *backtrack* one or more steps and choose a branching assignment opposite from the original choice. If no backtracking is possible, then the formula is unsatisfiable. The original DPLL algorithm includes optimizations to eliminate always-true clauses (*tautology elimination*) and unnecessary branches in the search if a clause contains only one literal (*Boolean constraint propagation*) or if a literal appears in the formula, but its negation does not (*pure literal elimination*).

Advances in SAT checkers have greatly increased their ability to solve large problems quickly. Four major improvements include optimized Boolean constraint propagation algorithms [86, 63], learning [60], non-chronological backtracking [60, 6], and improved heuristics for deciding on which literals to split [63].

Boolean constraint propagation is the detection and propagation of forced variable assignments. If a clause has a single literal of undetermined value, that unit

```
while true do
    if (out of literals) return SAT;
    choose literal;
    assert literal;
    if (propagation = CONFLICT)
        analyze conflict
        if (can backtrack) backtrack decision(s)
        else return UNSAT
done
```

Figure 2.3: Basic DPLL algorithm.

literal must be satisfied for the formula to be satisfiable — only one case needs to be explored. The process of detecting new unit literals has been made much more efficient through the use of *watched literals*, a method for detecting new unit clauses with minimal work (originally in SATO [86] and modified in Chaff [63]). There have also been a number of *decision heuristics* developed to choose what literals to assert, always with the tradeoff of effectiveness against computation time (the "pure literal" rule from the original DPLL can be seen as one of these); variations of one developed for Chaff [63] have recently become popular. Some of the most recent fast systems have also included variant decision strategies with heuristics to select between them dynamically [36]. *Learning* and *non-chronological backtracking* (both demonstrated in GRASP [60]) both involve conflict analysis. When the basic procedure encounters a clause that has been made unsatisfiable, it is necessary to backtrack its last assignment decision and try another option. It is often possible to perform an analysis on this conflict in order to further prune the search space by backtracking further. Learning involves adding new clauses called *lemmas* that are logical consequences of

the input formula that prevent a repetition of a bad assignment in the search.

2.7.2 BMC-specific Optimizations

There have also been a number of BMC-specific optimizations to the basic DPLL algorithm.

In a bounded model checking encoding, there is frequently a good deal of structural repetition in the constructed formula, especially from applying the same transitions in different unrolling steps. This symmetry can be exploited when pruning the search space by learning new lemmas, as presented in [74] through *constraint replication*. Basically, because of the structure of formulas generated from subsequent unwindings i and j of a system (with $i < j$), if a conflict is generated with respect to a given unwinding i, then a similar conflict is likely to occur with unwinding j as well. Whenever a lemma is generated from a conflict in step i, this near repetition in the formula allows the solver to immediately add additional versions of the lemma, referring to versions of the conflicting variables in step j as well. There are some limitations to this, however, as the formula is not completely symmetric for each step. The bounded cone of influence (Section 2.4.2) tends to break this symmetry, for one, meaning certain variables that would have been replicated might be outside the cone. A simple solution is to only add replicated clauses when all of their variables are within the BCOI. Another option is to check if the proposed clause is indeed unsatisfiable, and only add it as a lemma if so. Finally the initial and goal properties may also break the symmetry, but these endpoints can be simply marked and their

influence traced, stopping the replication if a marked variable is encountered [39].

Another repetition of work between steps in the BMC unwind/check/unwind/check algorithm is that most of the formula from step i will appear again in step $i+1$, and so much of the conflict information will remain the same. The basic BMC algorithm does not take this into account, and treats each SAT step as independent of all others. Instead, it is possible to *forward* some of the information learned in a previous instance to the next instance in the form of lemmas, saving some repetition of the search. In general, any lemma clauses that would be common to the current and next instances could be immediately applied to the later instance. Detecting these forwardable lemmas can be done by starting with a marked set of clauses (those not based off the property holding at the bound, for example), and then marking new lemmas whenever all of their parent clauses are marked [75]. These lemmas are then forwarded to the next iteration, and are still considered marked.

In [55] the authors provide a more elaborate scheme for forwarding clauses, one called *distillation*. In this case small lemmas that would not otherwise be forwarded are checked against the automatically forced assertions of the new instance. If the proposed clause is already satisfied it is redundant and so discarded, otherwise the *negation* of the clause's literals are asserted (making it unsatisfiable). Conflict analysis will result in a new distilled lemma that is then forwarded. Distilling allows both new clauses to be generated as well as an opportunity to refine the current decision heuristic before the normal search is begun.

There has also been work done on modifying the splitting heuristics for SAT

checkers when used on BMC problems. Again the structured nature of BMC formulas can be exploited. In a BMC formula, any given clause will generally only contain literals that originated in a few adjacent steps in the unwinding process. Therefore if we first choose literals that were all generated at unwinding step i and $i+1$, for example, it is more likely that they will interact than if we chose literals from step i and, say, step $i + 12$. Unfortunately, the usual SAT heuristics do not take this into account. In [74], Strichman mentioned that the usual heuristics (specifically the GRASP [60] default, which attempts to maximize the number of satisfied clauses) tended to choose variables that satisfy distinct sets of clauses. Only after these sets expand do they begin to interact with each other and result in conflicts, necessitating large amounts of backtracking. The presented ideas were to enforce some form of localization on the choices so that splits would tend to modify clauses that would immediately interact and so catch conflicts earlier. Both static orderings and a dynamic window based on such static orderings were tried, with the static ordering performing somewhat better.

2.7.3 Satisfiability Modulo Theories

One area of particular interest is in exploiting these highly efficient satisfiability techniques to solve formulas from non-Boolean domains in the input formulas, an effort known as Satisfiability Modulo Theories (SMT) [71, 69]. In SMT, a SAT engine is combined with efficient decision procedures for one or more theories with domains such as uninterpreted functions and linear arithmetic over the integers, allowing the SMT solver to accept non-propositional (usually quantifier-free) formulas from a frag-

ment of first-order logic. There are two basic approaches to SMT, classified as *eager* or *lazy*. Eager approaches basically encode the non-Boolean information directly into a Boolean formula ahead of time in a satisfiability preserving transformation (UCLID [14] is a primary example), while lazy approaches instead try to find an abstract propositional (partial) model and then cross-check this against the theory.

One such lazy method is using the *DPLL(T)* framework [76, 43]. This allows the inclusion of decision procedures for specific domains into the general DPLL framework, and so the manipulation of richer formulas. In DPLL(T), for example, some theory T is incorporated into the engine in the form of a decision procedure.

When the DPLL solver encounters a theory-interpreted literal (typically non-Boolean, possibly an abstraction for a more complicated formula), the solver queries the theory decision procedure as to the status of the predicate. The decision procedure generally keeps an internal set of theory-entailed literals in the form of a constraint store, and if the literal (or its negation) is in this set, then the variable is treated as if it had been assigned a truth value by the solver and Boolean constraint propagation is performed. If the decision procedure cannot produce a truth value for the literal, then it is treated as just another Boolean literal, and the solver may assign it an arbitrary value. If the solver does assign a value to an interpreted literal then that literal is asserted to the theory, which adds it to its internal constraint store. If the constraint store becomes unsatisfiable, then the theory notifies the solver and the solver must backtrack.

2.7.4 Common Theories

There are a number of popular theories that SMT solvers support, typically ones that can be easily combined using methods based on [64].

Of particular interest to us are theories of tuples, equality with uninterpreted functions, and linear arithmetic over the integers and rationals. These will be of use when proving properties for Lustre programs (see Section 3.3).

The theory of *tuples* includes a tuple constructor operation and projection operation that allows extraction of a term in a given position. *Equality with uninterpreted function symbols* ($\mathcal{EUF}$) is a theory concerning atoms of the form $t_1 = t_2$ where t_i is a functional term of the form a or $F(t_j, \cdots, t_k)$. The theory includes axioms of equality and congruence between functions. It is typically implemented via congruence closure. *Rational linear arithmetic* ($\mathcal{LA}(\mathbb{Q})$) is a theory involving atoms of the form $a_0 \times x_0 + \cdots + a_i \times x_i \bowtie b$, where a_i, b are rational constants and $\bowtie$ can be any of $\{=, <, >, =<, >=, \neq\}$. Decision procedures for this theory are typically implemented as variants of well-known linear programming techniques such as the simplex method or Fourier-Motzkin elimination. *Integer linear arithmetic* ($\mathcal{LA}(\mathbb{Z})$) is of a similar form, but coefficients and variables are limited to integers.

2.8 Summary

This chapter covered a number of background areas useful in putting this work in context. It included some general definitions and background on a number of versions of automated system verification, including several flavors of model checking,

such as symbolic model checking and bounded model checking. It also included a brief overview of deductive and inductive approaches and compared general strengths and weaknesses of these approaches. This chapter also introduced the concept of model abstraction and refinement in the context of model checking. Finally it gave some details of the evolution of SAT to SMT, as well as some common theories supported in SMT that will be useful later in this dissertation.

CHAPTER 3
CHECKING SAFETY PROPERTIES OF SYNCHRONOUS DATAFLOW LANGUAGES

3.1 Introduction

This chapter includes an overview of reactive programming and specification languages, used to describe systems designed to quickly and continuously react to their environments. In particular we will focus on the Lustre language. Recall that we are interested in verification of safety properties. We verify these properties using a logic-based approach where we translate the system and properties into formulas and then reason about them using automated deduction technologies. To do this we have devised a translation that takes an *idealized* form of Lustre into a suitable first-order logic $\mathcal{IL}$ that can model Lustre at a useful level of abstraction. A distinguishing feature for this logic is that it lends itself to reasoning with Satisfiability Modulo Theories techniques (see Section 2.7.3).

3.2 Reactive Languages

When specifying or implementing reactive systems, it is often convenient to use languages specifically designed for these tasks instead of attempting to modify traditional languages. *Reactive languages* are designed for situations where there is arbitrary, constantly changing input and a desire to quickly respond to that data. As such, they can be ideal for modeling or designing embedded systems.

Synchronous programming languages are a class of reactive languages based on

a theory of synchronous time, where the system and its environment are considered to both view time with respect to some (abstract) universal clock. In order to simplify reasoning about such systems, outputs are usually considered to be calculated instantly [7]. Such languages are designed to describe "real-time systems", or ones that must react quickly to a dynamic environment. Examples include Esterel [8], an imperative-style synchronous language and Lustre [17, 46], a dataflow synchronous language.

Dataflow languages are intrinsically different from traditional imperative programming languages in that they focus on data rather than control. They have no direct concept of stateful variables or commands as in an imperative language, and instead represent all data as infinite sequences of values called *streams* $(s_0, s_1, \ldots)$. Streams are related functionally, with changes in one impacting others immediately in parallel. Analogous to a system's state would be what we call an *instantaneous configuration* of its streams, the set of stream values at a given point in time. A simple example of a dataflow system would be a thermostat: it continuously takes in an analog value (temperature) and returns a Boolean value (turning a heater on or off).

3.3 Lustre

Lustre [17, 46, 48] is a simple synchronous dataflow language designed to model reactive systems through a series of equational definitions. It was designed to be used both as a programming language and as a specification language; as a result

its current incarnation is used primarily as an intermediate language of the commercial SCADE® development suite created by ESTEREL Technologies, and used extensively in development of a number of commercial products, including avionics systems designed by the Airbus corporation and Rockwell Collins, among others. Its rigorous design and simplicity lend themselves well to model checking and verification, especially of safety properties.

Lustre combines each data stream with an associated *clock* as a means of discretizing time. The overall system is considered to have a default clock that represents the smallest time span the system is able to distinguish, with additional, coarser-grained, user-defined clocks. Therefore the overall system may have different subsections that react to inputs at different frequencies. User-defined clocks are represented simply as Boolean streams. At each clock tick, the system is considered to evaluate all streams, so all values are considered stable for any actual time spent in the *instant* between ticks. A stream *position* can be used to indicate a specific value of a stream in a given instant, indexed by its clock tick. A stream at position 0 is in its *initial configuration*. Positions prior to this have no defined stream value.

Variables in Lustre are used to represent individual streams. Variables are typed, with basic types including streams of *real* numbers, *integers*, and *Booleans*. Additonally, basic Lustre contains a composite *tuple* type, allowing streams representing arbitrary groupings of other streams.

Lustre programs and subprograms are expressed in terms of *nodes*. Nodes directly model subsystems in a modular fashion, with an externally visible set of

inputs and outputs. In a functional sense, a node can be seen as a mapping of a finite set of input streams (in the form of a tuple) to a finite set of output streams (also expressed as a tuple). At each instant i, the node takes in the values of its input streams and returns the values of its output streams. The *top node* is the main node of the program, the one that would interface with the outside world and never be called by another node.

Nodes have a limited form of memory that allows reference to their stream values from a finite number of previous instants; how far back is determined statically through use of specific clock operators in the program.

The body of each node is a set of *stream definition equations* of the form $x = t$, where x is a local or output stream variable and t is a Lustre expression containing variables denoting node-specific input, output, or local streams. For the most part, the semantics of Lustre's operators are a direct lifting of the base operators to streams. For example, if $x = (x_0, x_1, \ldots)$ and $y = (y_0, y_1, \ldots)$ are integer streams, then $x + y$ is the stream consisting of the values $(x_0 + y_0, x_1 + y_1, \ldots)$. Arithmetic operators include plus $(+)$, minus $(-)$, division $(/)$, multiplication $(*)$, integer division (div), and modulus (mod), and excepting the last two, are overloaded for both the integer and real types. Boolean operators include and, or, and not. Relational operators include $=$, $<$, $>$, $\geq$, and $\leq$, defined for both integer and real streams. Additionally there is a conditional if-then-else operator. *Constants* are streams with a fixed value at all instants, and can include numerals or the true or false streams.

Additionally there are four temporal operators that involve a stream's clock:

previous (pre), followed-by ($\rightarrow$), when, and current. For any instant i pre(x) returns the value of x at instant $i-1$, and functions as a delay, or basic memory. pres may be nested to arbitrary depth, though the pre of a stream's initial value is *undefined*. The followed-by arrow allows the assignment of initial values to streams. At instant 0, $x \rightarrow y$ returns the value of x at instant 0. For any subsequent instant $i > 0$, it returns the corresponding value of stream y. The when operator allows a stream to be sampled according to a user-defined clock y. The stream x when y returns the current value of x in an instant for which y is true. If y is *false*, then the stream x when y is undefined. A node instance's clock is dictated by the clock of its inputs, so sampling the parameter of a node can be used to simulate subsystems operating at a slower clock[1]. The current operator interpolates a value from a stream with a slower clock, allowing it to be read by a stream with a faster clock. The when and current operators are generally composed together. See Figure 3.1 for an example utilizing these operators. Note that some stream expressions with unguarded temporal operators are not well defined.

A well-behaved Lustre program will only perform calculations on streams when they are defined. This may be achieved, for example, through the use of $\rightarrow$ operators to guard pre expressions, and current operators to guard when expressions.

An example program can be seen in Figure 3.2. In this program the node `reset_timer` takes a Boolean `reset` as input and returns a Boolean `alarm`. It contains

---

[1]If a node's inputs have different clocks, the *fastest* is the base clock for that instance, and any other clocks must be explicitly included as inputs. In this case the formal node header must include this information, such as `node N (a:bool; x:int when a) returns (y:int)...`, with N's default clock being the clock of a, while x's clock is a. [46] Similarly, outputs must include any clocks that may be slower than the rest of the node.

	global clock										
stream	0	1	2	3	4	5	6	7	8	9	...
c	F	F	T	F	F	T	F	F	T	F	...
x	1	2	3	4	5	6	7	8	9	0	...
$y =$ pre x	-	1	2	3	4	5	6	7	8	9	...
$z = x \rightarrow y$	1	1	2	3	4	5	6	7	8	9	...
$w = z$ when c	-	-	2	-	-	5	-	-	8	-	...
$v =$ current w	-	-	2	2	2	5	5	5	8	8	...

Figure 3.1: Example of temporal operators in Lustre. The stream c is a Boolean input clock, x is an integer input stream, all others are local integer streams. A dash (-) indicates an undefined value at that position.

a local integer variable `time` to keep track of the time passed.

Lustre does not support recursive node calls (actually more recent versions *do*, in a very limited way, see Section 3.3.1). Additionally, stream equations must be well-founded: a stream cannot depend on its own current value. Let $\mathbf{R}$ be the binary relation between streams x, y in a single same node such that $x \ \mathbf{R} \ y$ iff y occurs in the definition of x not within a pre. Given that this is a finite relation, the equations are well founded if $\mathbf{R}$ does not contain any cycles. This can be expanded to multi-node programs by first inlining any node calls (see Section 3.4).

Lustre also allows functions to be imported from a host language. These function calls are treated as "black box" node calls within Lustre itself.

In addition to variable definitions, a Lustre program may contain *assertions* that introduce additional constraints on variables. These boolean expressions are always considered to be true, and generally used to represent known restrictions to inputs. Operationally, assertions are transparent to the user, though they may allow

```
node reset_timer (reset:bool) returns (alarm:bool);
  var time: int;
let
  time = 1 -> if reset then
                1
              else
                if pre(time) = 10 then
                  1
                else
                  pre(time) + 1;
  alarm = (time = 10);
tel
```

Figure 3.2: Example Lustre program. This is a simple timer with a reset. Its return value represents signaling an alarm; this happens every 10 clock ticks, unless a reset is input.

certain compiler optimizations, such as skipping certain tests. As Lustre is also a specification language, they primarily act as assumptions on the program behavior (and so weakening any properties verified).

3.3.1 Recent Versions

The publicly available Version 4 of Lustre [49] has added several refinements to the language to aid in use. These are generally syntactic sugar and do not add significant power or flexibility to the core language. New operations are supported including xor, <>, and the "onetrue" # operator. This # operator returns true if exactly one of a tuple of Boolean streams is true. Also, Version 4 adds explicit type conversions between values of simple data types, for example bool(0) gives the value false. Also added are arrays and a limited form of recursion.

Arrays are implemented in terms of single-type tuples. Though they do not

add any power to the language, they allow the user to perform identical operations or node calls on all elements of slices of an array in a single expression, as opposed to duplicating the operations to access all elements of a tuple. Additionally a limited form of recursive calls is allowed, though the semantics of this is merely unrolling the recursion into a (finite) sequence of normal node calls.

The SCADE version of Lustre [67, 79] also includes extensions such as record data types, a case statement, and the fby and condact operators. fby initializes a variable over a specified number of clock ticks. Condact combines initialization, current, and when operators in a single command to create safely guarded expressions with slower clocks.

3.4 Translating Lustre into $\mathcal{IL}$

In this section we describe a translation of Lustre programs into first-order logic formulas. We have developed this translation along the lines of Anders Franzén's work [40] on the Rantanplan system (described in Section 6.3.3) and of Michael Whalen, et al. [84, 83]. The main differences are that [40] translates into a formulation of Boolean and integer terms, while we translate into a logic $\mathcal{IL}$, with support of reals and other data types. And while [84, 83] also supports Lustre data types other than Booleans and integers, it focuses on translations tailored to a two-state formulation.

Lustre is a real-world programming language, so its int and real data types actually represent machine-bounded integer and floating point numbers. There are currently no efficient reasoners for such data types, but there *are* ones for unbounded

integers and infinite-precision rational numbers, at least under linear arithmetic. As a result, we will actually work with an *idealized* version of Lustre, using these latter data types. This version of Lustre can be modeled with a typed, first-order logic $\mathcal{IL}$ containing uninterpreted function symbols, tuples, integers, and rationals. A distinct advantage of $\mathcal{IL}$ is that, when used with quantifier-free formulas and linear numerical terms, it allows the use of efficient SMT-based decision procedures. The use of these decision procedures, however, does present some limitations on $\mathcal{IL}$ beyond that of actual Lustre, in that $\mathcal{IL}$ is unable to model behaviors involving nonlinear numeric calculations, as well as errors due to overflow or underflow.

In $\mathcal{IL}$, we can represent a Lustre stream of type τ simply as an uninterpreted function from $\mathbb{N}$ to τ, mapping a temporal position to the stream value's type, possibly involving conditional if-then-else (*ite*) expressions. These functions are applied by use of the homomorphic translation function tr, described in Figure 3.3. Note that the numeric position n is interpreted *with respect to a program's global clock* — streams with slower clocks need to be dealt with specially. Therefore Lustre terms utilizing arithmetic or Boolean operations can be mapped to $\mathcal{IL}$ terms with the corresponding operations, and Lustre equational definitions can be represented as universally quantified equations of stream values. For example, retaining the Lustre stream names in the corresponding uninterpreted functions to more clearly show the origins of the formulas, the Boolean stream equation $a = b \ \mathsf{or} \ c$ is translated as $\forall n : \mathbb{N}.a(n) = b(n) \vee c(n)$, and the integer stream equation $x = y + z$ becomes $\forall n : \mathbb{N}.x(n) = y(n) + z(n)$. Most temporal operators also present straightforward

semantics as shown in Figure 3.3. The when operator can present special problems, however, so its translation is limited to certain cases.

Let N be any single-node Lustre program with stream variables

$$\overline{v} = \langle x_1, \ldots, x_l, y_1, \ldots, y_m \rangle$$

where $x_1, \ldots, x_l$ are N's input variables and $y_1, \ldots, y_m$ are N's non-input (i.e., local and output) variables. The semantics of N is completely captured in $\mathcal{IL}$ by the universal quantification over the integer variable n of the following system of equations

$$\Delta(n) = \begin{cases} y_1(n) & = & t_1[\overline{v}(n), \overline{v}(n-1), \ldots, \overline{v}(n-d)] \\ \vdots & \vdots & \vdots \\ y_q(n) & = & t_q[\overline{v}(n), \overline{v}(n-1), \ldots, \overline{v}(n-d)] \end{cases}$$

where each $y_i(n) = t_i$ is the translation of the expression defining y_i in N and d is the nesting depth of pre operators in N.

Generally, multi-node programs can be modeled the same way by first recursively in-lining in the main node all calls to sub-nodes as shown below and then translating the main node as explained earlier.

Let $N = A \cup \{y = t[f(\overline{a})]\}$ be a Lustre node body containing a call to another node f with arguments $\overline{a}$. Let node f be defined by body B, with output stream o (possibly a tuple) and formal input streams $\overline{v}$. Let B' be a *fresh copy* of B where all streams have been renamed with new symbols, with o' and $\overline{v}'$ being the renamed versions of o and $\overline{v}$, respectively. Before translating to $\mathcal{IL}$, we first inline $f(\overline{a})$ into N by rewriting N as follows:

$$A \cup \{y = t[f(\overline{a})]\} \longrightarrow A \cup \{y = t[f(\overline{a})/o']\} \cup \{B'[\overline{v}'/\overline{a}]\}$$

constant streams:	
$\mathsf{tr}(c,\ n)$	c
variables:	
$\mathsf{tr}(x,\ n)$	$x(n)$
unary operators:	
$\mathsf{tr}(-e,\ n)$	$-\mathsf{tr}(e,\ n)$
$\mathsf{tr}((\mathsf{not}\ e,\ n)$	$\neg\mathsf{tr}(e,\ n)$
binary operators ($\oplus = \{+, -, *, /, \mathsf{div}, \mathsf{mod}\}$):	
$\mathsf{tr}(e_1 \oplus e_2,\ n)$	$\mathsf{tr}(e_1,\ n) \oplus \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1\ \mathsf{or}\ e_2,\ n)$	$\mathsf{tr}(e_1,\ n) \vee \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1\ \mathsf{and}\ e_2,\ n)$	$\mathsf{tr}(e_1,\ n) \wedge \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1\ \mathsf{xor}\ e_2,\ n)$	$\mathsf{tr}(e_1,\ n)\ \mathsf{xor}\ \mathsf{tr}(e_2,\ n)$
binary relations ($\bowtie = \{=, <, >\}$):	
$\mathsf{tr}(e_1 \bowtie e_2,\ n)$	$\mathsf{tr}(e_1, n) \bowtie \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1 <> e_2,\ n)$	$\mathsf{tr}(e_1, n) \neq \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1 <= e_2,\ n)$	$\mathsf{tr}(e_1, n) \leq \mathsf{tr}(e_2,\ n)$
$\mathsf{tr}(e_1 >= e_2,\ n)$	$\mathsf{tr}(e_1, n) \geq \mathsf{tr}(e_2,\ n)$
if-then-else:	
$\mathsf{tr}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,\ n)$	$ite(\mathsf{tr}(e_1,\ n), \mathsf{tr}(e_2,\ n), \mathsf{tr}(e_3,\ n)))$
tuples:	
$\mathsf{tr}(\langle e_0, \ldots, e_i\rangle,\ n)$	$\langle \mathsf{tr}(e_0,\ n), \ldots, \mathsf{tr}(e_i,\ n)\rangle$
temporal operators:	
$\mathsf{tr}(\mathsf{pre}\ e,\ n)$	$\mathsf{tr}(e,\ (n-1))$
$\mathsf{tr}(e_1\ \rightarrow\ e_2,\ n)$	$ite(n = 0, \mathsf{tr}(e_1,\ 0), \mathsf{tr}(e_2,\ n))$
$\mathsf{tr}(\mathsf{current}(e_1\ \mathsf{when}\ e_2),\ n)$	$ite(\mathsf{tr}(e_2,\ n), \mathsf{tr}(e_1,\ n), \mathsf{tr}(e_1,\ (n-1)))$
stream definition:	
$\mathsf{tr}(e_1 = e_2,\ n)$	$\mathsf{tr}(e_1,\ n) = \mathsf{tr}(e_2,\ n)$

Figure 3.3: Basic Lustre translation semantics of the translation function tr for a program N. In all cases, Lustre streams of type τ become uninterpreted functions in $\mathcal{IL}$ of type $\mathbb{N} \rightarrow \tau$ with the same symbol. Note constant streams may be represented directly as constant values for all positions and do not need to be translated into uninterpreted function equivalents.

where $t[f(\overline{a})]$ is a term containing $f(\overline{a})$ as a subterm, and $t[f(\overline{a})/o']$ is the result of substituting an occurrence of $f(\overline{a})$ with o' in term t (and similarly for $B'[\overline{v}'/\overline{a}]$). We repeat this process on the new version of N until all calls have been inlined.

Note that assert statements in Lustre are Boolean expressions interpreted as additional constraints on the streams in the expression. When proving a property of a node, they can be considered as assumptions for the property, and as such weaken it. Typically these assertions are added to constrain some input values to a certain range or behavior. asserts may be included in our translation by adding their translated formulas to Δ. However, if they are not invariant they may introduce unsoundness to the overall verification process.

3.4.1 Example

As an example of the translation process into $\mathcal{IL}$, we will consider a simple Lustre program that compares two implementations of a 2-bit counter: a low-level Boolean implementation and a higher-level implementation using integers. The Lustre code can be found in Figure 3.4.

The `greycounter` node internally repeats the sequence $ab = \{00, 01, 11, 10, 00, \ldots\}$ indefinitely, while the `integercounter` node repeats the sequence $time = \{0, 1, 2, 3, 0, \ldots\}$. In both cases the counter returns a boolean value that is true iff the counter is in its third step and the input `x` is true. The top node `test` is an example of a synchronous observer. So we wish to verify the safety property that both implementations have the same observable behavior, i.e. that the stream `OK` is

```
node greycounter (x:bool) returns (out:bool);
var a,b:bool;
let
  a = false -> not pre(b);
  b = false -> pre(a);
  out = x and a and b;
tel

node integercounter (x:bool) returns (out:bool);
var time: int;
let
  time = 0 -> if pre(time) = 3 then 0
              else pre time + 1;
  out = x and (time = 2);
tel

node test (x:bool) returns (OK:bool);
let
  OK = (greycounter(x) = integercounter(x));
tel
```

Figure 3.4: Lustre counter example code. Each node returns a true signal if $x =$ true when the counter is in its third state. The node `test` returns the value of the safety property to be proven.

```
node test (x:bool) returns (OK:bool);
var ga,gb:bool;
    itime: int;
let
  ga = false -> not pre(gb);
  gb = false -> pre(ga);
  gout = x and ga and gb;
  itime = 0 -> if pre(itime) = 3 then 0
            else pre itime + 1;
  iout = x and (itime = 2);
  OK = (gout = iout);
tel
```

Figure 3.5: Lustre counter inlined code. The *greycounter* streams have been renamed with a prepended g, while the *integercounter* streams have been renamed with a prepended i.

always true.

First we inline the `greycounter` and `integercounter` nodes into `test`. For example the `greycounter` node becomes a set of new stream definitions that are renamed versions of those in `greycounter`, in this case all having a prepended `g`. Next, any formal parameter streams are renamed to correspond to the actual node call parameters (here the newly renamed `gx` is replaced by `x`). Finally, the node call itself is rewritten as the (fresh instance's) output stream, here `gout`. The full result of this rewriting for both nodes can be seen in Figure 3.5.

Each stream definition equation is translated according to the basic translation

$$\Delta(n) = \left\{ \begin{array}{l} ga(n) = ite(n = 0, \mathsf{false}, \neg gb(n-1)) \\ gb(n) = ite(n = 0, \mathsf{false}, ga(n-1)) \\ gout(n) = x(n) \wedge ga(n) \wedge gb(n) \\ itime(n) = ite(n = 0, 0, ite(itime(n-1) = 3, 0, itime(n-1) + 1)) \\ iout(n) = x(n) \wedge (itime(n) = 2) \\ OK(n) = (gout(n) \Leftrightarrow iout(n)) \end{array} \right\}$$

Figure 3.6: Translated Lustre counter example.

semantics of tr from Figure 3.3:

$$\mathsf{tr}(ga = \mathsf{false} \rightarrow \mathsf{not}\ \mathsf{pre}(gb); , n) \Longrightarrow$$

$$\mathsf{tr}(ga, n) = \mathsf{tr}(\mathsf{false} \rightarrow \mathsf{not}\ \mathsf{pre}(gb), n) \Longrightarrow$$

$$ga(n) = ite(n = 0, \mathsf{tr}(\mathsf{false}, n), \mathsf{tr}(\mathsf{not}\ \mathsf{pre}(gb), n)) \Longrightarrow$$

$$ga(n) = ite(n = 0, \mathsf{false}, \neg \mathsf{tr}(pre(gb), n)) \Longrightarrow$$

$$ga(n) = ite(n = 0, \mathsf{false}, \neg \mathsf{tr}(gb, n-1)) \Longrightarrow$$

$$ga(n) = ite(n = 0, \mathsf{false}, \neg gb(n-1))$$

The other definition equations are translated similarly into Δ, as seen in Figure 3.6.

A second example with a greater nesting of pre, along with its translation, is shown in Figure 3.7.

3.5 Checking Safety Properties in Lustre

Given a tuple of streams, we call any well-typed tuple of values for these streams at a certain instant an *instantaneous configuration*. Note that at any particular instant the variables $\overline{v}$ in a Lustre program N denote precisely one instantaneous configuration. Effectively, then, a Lustre program can be seen as a system of constraints over instantaneous configurations.

```
node fib (max:int) returns (y:int);
let
  y = if ((1 -> pre y) < max) then
        1 -> pre(y + (0->pre y))
      else
        pre y;
tel
```

(a)

$$\Delta(n) = \left\{ y(n) = \begin{array}{l} ite(ite(n=0,1,y(n-1)) > max(n), \\ \quad ite(n=0,1,y(n-1) + \underline{ite(n=1,0,y(n-2))}), \\ \quad y(n-1)) \end{array} \right\}$$

(b)

Figure 3.7: Fibonacci example. Lustre code (a) and translated equation system (b). This node returns the Fibonacci sequence up to a user-defined maximum; after it exceeds that maximum ($max \geq 0$), it constantly returns the last sequence number generated. Note how the doubly nested ($0 \rightarrow \mathsf{pre}\ y$) is further offset in the translation (underlined).

Each program has a certain amount of memory associated with it, a *depth* $d \geq 0$ that is the maximum nesting depth of pre operators in the program, or the maximum number of instantaneous configurations that need be considered to fully calculate the current configuration. This notation for $\Delta(n)$ is conceptually consistent with the fact that the value of $y_i(n)$ is a function of (some of the values in) the configurations at instants $n, n-1, \ldots, n-d$. Note that it is possible to reduce any program to an equivalent one with a depth at most 1 by fully flattening all terms involving pre. This will be used later to simplify the explanation of our algorithms. Also note that under this assumption, $\Delta(n)$ can be seen as a translation relation between consecutive configurations on a legal trace.

For a program N, a *trace* is a tuple $\overline{s} = \langle s_1, \ldots, s_{l+m} \rangle$ of streams of the same

type as N's variables $\overline{v}$. A *path* of length q is a finite sequence of q consecutive configurations on a trace. The possible input/output behaviors of N are exactly the traces $\overline{s}$ that satisfy the formula $\forall n : \mathbb{N}.\Delta(n)$ when $\overline{v}$ is interpreted as $\overline{s}$. We call such traces *legal traces*. A configuration is *reachable* if it occurs on some legal trace. It is *initial* if it is the first configuration on a legal trace, and all and only configurations that satisfy $\Delta(0)$ are initial configurations. A path is *reachable* if it lays on a legal trace. A path is *initial* when it is reachable and begins with an initial configuration.

As argued in [48], with Lustre programs one is primarily interested in verifying safety properties. The sense is that, since Lustre is used to develop critical reactive systems, we are usually not so much interested in liveness properties, but rather in ensuring that some failure state is never attained. Even if a liveness property P is supposedly desired, it is usually only required in a restricted sense for these systems: knowing if the property P holds *eventually* is often not very useful in a real-world setting. It is much more useful to know if the property will hold within some upper bound. As an example, "we will eventually reach the desired cruising speed" is not as useful as "we will reach the desired cruising speed within at most fifteen minutes." Or, as a more extreme example, consider "in the event of a fire, a suppressant will eventually be deployed." Given such hard time limits it is rather simple to re-conceive P as a safety property.

More precisely, we verify properties that are invariant in the following sense: a property P of paths is *invariant (for N)* if it holds for all reachable paths.

We will consider only *quantifier-free properties*, that is properties that can be

expressed by a quantifier-free formula in $\mathcal{IL}$. While this is a restriction, in common practice safety properties can generally be expressed as quantifier-free formulas.

Further, we can reduce checking a property P of *paths* to the checking of *instantaneous configurations* if P is expressible as a Lustre Boolean expression through the use of a *synchronous observer* [50]. A synchronous observer is a wrapper used to test observable properties of a node N with minimal modification to the node itself; it returns an error signal if the property does not hold, reducing the more complicated property to a single Boolean stream where we need only check if the stream is constantly true.

Checking invariant properties of Lustre programs can be done by lifting and adapting a number of SAT-based model checking techniques to the logic $\mathcal{IL}$. The main one used here is k-induction.

3.6 current and when

Because we are using the *global clock* as the basis of our concept of stream position in our translations into $\mathcal{IL}$ (referenced by the variable n), the semantics of Lustre's current and when operator can cause difficulties, and only specific cases are handled by *tr*. One example that is not supported by *tr* is shown in Figure 3.8. It contains a program with a simple auxiliary node containing a counter that starts at 0 and increments every clock tick, but the second `hiddencounter` instance node is running at a different clock than the main system. Because the parameter of the *h*2*time* `hiddencounter` is at a slower clock, the execution of the node should be

```
node hiddencounter (c:bool) returns (hc:int)
let
  hc = 0 -> pre(hc) + 1;
tel

node basecounter (c:bool) returns (time,h1time,h2time:int)
let
  time = hiddencounter(c);
  h1time = current ((hiddencounter(c)) when c);
  h2time = current (hiddencounter(c when c));
tel
```

(a)

	global clock										
stream	0	1	2	3	4	5	6	7	8	9	...
c	F	F	T	F	F	T	F	F	T	F	...
$time$	0	1	2	3	4	5	6	7	8	9	...
$h1time$	-	-	2	2	2	5	5	5	8	8	...
$h2time$	-	-	0	0	0	1	1	1	2	2	...

(b)

Figure 3.8: Hidden counter example. Lustre code (a) and a sample timing diagram (b). A dash (-) indicates an undefined value at that stream position. In this case the timing diagram shows the program's behavior when properly following the semantics of when. Note that $h1time$ demonstrates the effect of when on a node's output (it continues to operate at the global clock frequency), while $h2time$ demonstrates the effect of when on a node's inputs, putting the node on the slower clock c. All nodes must contain at least one input, precisely to establish their respective clocks.

$$\Delta(n) = \begin{cases} hc_0(n) = ite(n = 0, 0, hc_0(n-1) + 1) \\ time(n) = hc_0(n) \\ hc_1(n) = ite(n = 0, 0, hc_1(n-1) + 1) \\ h1time(n) = ite(c(n), hc_1(n), hc_1(n-1)) \\ hc_2(n) = ite(n = 0, 0, hc_2(n-1) + 1) \\ h2time(n) = hc_2(n) \end{cases}$$

(a)

	global clock (n)										
stream	0	1	2	3	4	5	6	7	8	9	...
$c(n)$	F	F	T	F	F	T	F	F	T	F	...
$hc_0(n)$	0	1	2	3	4	5	6	7	8	9	...
$hc_1(n)$	0	1	2	3	4	5	6	7	8	9	...
$hc_2(n)$	0	1	2	3	4	5	6	7	8	9	...
$time(n)$	0	1	2	3	4	5	6	7	8	9	...
$h1time(n)$	-	-	2	2	2	5	5	5	8	8	...
$h2time(n)$	0	1	2	3	4	5	6	7	8	9	...

(b)

Figure 3.9: Hidden counter example, incorrect translation (a) and example timing diagram (b). A dash (-) indicates an undefined value at that stream position. $hc_2(n)$ is the translation of the inlined call `hc2 = 0 -> pre(hc2) + (c when c)`.

slowed (as opposed to the node executing at the global clock speed, but only sampled at a slower speed).

If we simply inline the node call and apply a similar rule to $\mathsf{tr}(\mathsf{current}(e_1 \text{ } \mathsf{when} \text{ } e_2),\ n)$, such as

$$\mathsf{tr}(e_1 \text{ } \mathsf{when} \text{ } e_2,\ n) \longrightarrow \mathsf{tr}(e_1, ite(\mathsf{tr}(e_2,\ n), n, n-1))$$

then the parameter term c when c disappears in the normal inlining process. As shown in the example Figure 3.9, however, the modeled behavior will be incorrect due to the missing clock information. This problem can appear for any local variable in a node that contains state information. One possible means to relax the constraint on translating current and when (at least for some cases, currently not implemented) is presented below.

The streams involving a when expression can be undefined at certain instants. For the time being we do not choose to directly model these as potential error states, but instead to interpolate a stream's values when it is undefined — this may cause some intermediate values to be defined when they technically should not be, but it will not affect the final output of a well formed Lustre program. The problem with "hidden" state within a node call, however, remains. If the node's parameters are at a slower clock, the node should only be evaluated at that slower clock. It is therefore necessary to preserve additional clock information when inlining function calls. We do so by assigning a *label* λ_i to each stream variable that indicates its clock.

We will accomplish this in two steps: first we infer the clocks of all streams through a unification process, performed along with node inlining, then we will utilize

a modified tr (tr′) to finish the translation.

3.6.1 Clock Inference

First, we rewrite the program into a functionally equivalent program N such that any terms of the form x when y are not subterms of any expression, introducing fresh variables as needed.

Then we assign each stream s_i in a program N a label λ_i of type Boolean stream, containing either a variable placeholder x_i or a Boolean clock stream c_i. As a global constraint, stream variables with the same name should have the same label. We label input streams for the main node with the global, constant true clock c_0. Fresh labels are also generated when we inline node calls.

In order to model the semantics of Lustre clocks, we will need to establish an ordering on the clocks in N. Clock streams are partially ordered by the transitive closure of relation $<_f$, where $c_1 <_f c_2$ (c_1 is faster than c_2) iff the term x when c_1 is part of the (full) definition of c_2. The global clock c_0 is a lowest bound on clocks: $\forall i \geq 0. c_0 <_f c_i$. The *base clock* for a node is the fastest clock of its parameters. For the main node the base clock is c_0. For a subnode, the base clock is the least upper bound of its input streams.

Normally we can infer the clocks of a node's streams from the inputs — a complication arises, however, when a node's inputs do not appear on the right hand side of *any* stream definition in the node. To deal with these *orphaned inputs*, we can create a directed *dependency graph* $G = (E, V)$ of a node's streams, with graph ver-

tices V representing the program node's streams and directed edges $E = \{(v_1, v_2) | v_1$ on LHS of definition for $v_2\}$. If subgraphs do not have an input as a source, we add an edge e_{ij} from the subgraph's source to the fastest input into the node.

Let $N = A \cup \{y = t[f(\overline{a})]\}$ be a Lustre node body containing a call to another node f with *labeled* arguments $\overline{(a)} = \langle (a_0, L_{a_0}), \ldots, (a_j, L_{a_j}) \rangle$. Let node f be defined by body B, with labeled output stream (o, L_o) (possibly a tuple) and formal labeled input streams $\overline{v} = \langle (v_0, L_{v_0}), \ldots, (v_k, L_{v_k}) \rangle$. Let B' be a *fresh copy* of B where all streams have been renamed with new symbols having corresponding new, renamed labels, with $(o', L_{o'})$ and $\overline{v}' = \langle (v_0, L_{v'_0}), \ldots, (v_k, L_{v'_k}) \rangle$ being the renamed versions of o and $\overline{v}$, respectively. We then rewrite as in our previous translation.

Once all nodes have been inlined, we will use a set of rules to infer the clock streams to be associated with each program subterm. These rules are from $(T, U, I) \Longrightarrow (T, U, I)$, where T is the set of subterms of the program that contain some operator (including "=" as both a relation and definition operator), U is a set of equalities between labels, and I is a set of inequality constraints. These rules are found in Figure 3.10, and are applied until T is empty. Lustre clock semantics specify that (almost) all standard Lustre operations must be performed with all operands and result being on the same clock. The two exceptions are when, which has a slower clock for its result (specified by its second operand), and current, which has a faster clock for its result (specifically the "next fastest" clock, which would be the clock of the corresponding when operators). In the event of orphaned inputs, we unify the clock of the fastest input with the root stream of appropriate isolated dependency

Unary operators ($-, \neg, \mathsf{pre}, \ldots$, except $\mathsf{current}$):
$T \cup (op_1((t_1, \lambda_1)), \lambda_0), U, I \Longrightarrow T, U \cup \{\lambda_0 = \lambda_1\}, I$
Binary operators ($+, \mathsf{and}, \rightarrow, =, \ldots$, except when):
$T \cup (op_2((t_1, \lambda_1), (t_2, \lambda_2)), \lambda_0), U, I \Longrightarrow T, U \cup \{\lambda_0 = \lambda_1, \lambda_1 = \lambda_2\}, I$
N-ary operators ($\mathsf{if} - \mathsf{then} - \mathsf{else}, \ldots$):
$T \cup (op_n((t_1, \lambda_1), \ldots, (t_n, \lambda_n)), \lambda_0), E, I \Longrightarrow T, E \cup \{\lambda_0 = \lambda_1, \ldots \lambda_{n-1} = \lambda_n\}, I$
when:
$T \cup ((t_1, \lambda_1)\ \mathsf{when}\ (t_2, \lambda_2)), \lambda_0), U, I \Longrightarrow T, U \cup \{\lambda_0 = t_2, \lambda_1 = \lambda_2\}, I$
$\mathsf{current}^*$:
$T \cup (\mathsf{current}\ (t_1, \lambda_1)), \lambda_0), U, I \Longrightarrow T, U, I \cup \{\lambda_0 <_f \lambda_1\}$

Figure 3.10: Clock inference rules. Given the set of Lustre terms T (initially all subterms in the inlined program that contain an operand, including stream definitions such as $x = y$) and a set of equalities U (initially $U = \{\lambda_{I_0} = c_0, \ldots, \lambda_{I_n} = c_0\}$, the labels of all input streams $I_0, \ldots, I_n$ are equal to the global clock), and inequalities I (initially empty), we apply the above rules until T is empty. Additionally, U should initially contain equalities relating any orphaned input labels with their corresponding stream labels from the dependency graph. In the case of when, we introduce a new concrete clock. In the case of $\mathsf{current}$, we introduce a constraint on the clocks in the program. At this point, we should be able to determine a concrete clock to associate with each stream through standard unification, and the constraints in I should be fulfilled. If there are any clashes in the unification of E or if the constraints in I are not fulfilled, the program is rejected.

graphs. Any clashes in the unification of $E.$ or inconsistencies in I indicate an error.

3.6.2 Relaxed Translation

The idea behind the relaxed translation tr' is to eagerly interpolate all streams, stretching streams with slower clocks so that, once initialized, all streams have a defined value at each instant of the global clock. In the event that a stream would normally be undefined due to sampling with the when operator, the stream instead retains its last known value. The semantics of tr' can be seen in Figure 3.11.

This interpolation will not affect the verification of a safety property P, expressed as a Lustre Boolean stream. If the property contains a term t that consists of a stream operating at a slower clock, then t must be explicitly interpolated at P's clock by a current operator (or else P would be inconsistent). If P operates at a slower clock than one of its terms t (such as $P = t$ when c), we are only interested in P's value when it is sampled. If P held during its last sampled instant, it can be considered to continue to hold while it is being interpolated at the global clock rate.

Note this assumes that clock streams will not themselves be delayed (a subject for future work). Also note that an intermediate stream may still be translated so as to have an undefined value at some instants; as with pre this does not prove to be a problem as long as the undefined value is resolved for all outputs of the node.

This relaxed translation is similar to the basic translation with two notable exceptions: the interpolation of a stream at a slower clock occurs as part of the stream definition, and the initial instant of a slower stream is calculated as part of the $\rightarrow$

constant streams:	
$\mathsf{tr}'((c, c_c),\ n)$	c
variables:	
$\mathsf{tr}'((x, c_x)\ n)$	$x(n)$
unary operators:	
$\mathsf{tr}'((\ominus\ e,\ c_e),\ n)$	$\ominus \mathsf{tr}'(e,\ n)$
binary operators:	
$\mathsf{tr}'((e_1 \oplus e_2,\ c_e),\ n)$	$\mathsf{tr}'(e_1,\ n) \oplus \mathsf{tr}'(e_2,\ n)$;
binary relations:	
$\mathsf{tr}'((e_1 \bowtie e_2,\ c_e),\ n)$	$\mathsf{tr}'(e_1, n) \bowtie \mathsf{tr}'(e_2\ n)$;
if-then-else:	
$\mathsf{tr}'((\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,\ c_e),\ n)$	$ite(\mathsf{tr}'(e_1,\ n), \mathsf{tr}'(e_2,\ n), \mathsf{tr}'(e_3,\ n)))$
tuples:	
$\mathsf{tr}'((\langle e_0, \ldots, e_i\rangle,\ c_e),\ n)$	$\langle \mathsf{tr}'(e_0,\ n), \ldots, \mathsf{tr}'(e_i,\ n)\rangle$;
temporal operators:	
$\mathsf{tr}'((\mathsf{pre}\ e_1,\ c_e),\ n)$	$\mathsf{tr}'(e_1,\ (n-1))$
$\mathsf{tr}'((e_1\ \rightarrow\ e_2,\ c_e),\ n)$	if $c_e = c_0$: $ite(n = 0, \mathsf{tr}(e_1,\ 0), \mathsf{tr}(e_2,\ n))$
$\mathsf{tr}'((e_1\ \rightarrow\ e_2,\ c_e),\ n)$	if $c_e \neq c_0$: Create fresh variables ev_{c_e} and ft_{c_e}: let $ev_{c_e}(n) =$ $ite(n = 0, c_e(n), c_e(n) \vee\ ev_{c_e}(n-1))$ let $ft_{c_e}(n) =$ $ite(n = 0, c_e(n), ev_{c_e}(n) \wedge \neg ev_{c_e}(n-1))$ in: $ite(ft_{c_e}(n), \mathsf{tr}'(e_1,\ n), \mathsf{tr}'(e_2,\ n))$,
$\mathsf{tr}'((\mathsf{current}\ e,\ n), \lambda_e)$	$\mathsf{tr}'((e,\ n)$
$\mathsf{tr}'((e_1\ \mathsf{when}\ e_2, \lambda_e),\ n)$	$\mathsf{tr}'(e_1,\ n)$
stream definition:	
$\mathsf{tr}'((x_1 = e_2,\ c_e),\ n)$	$\mathsf{tr}'((x_1, c_e),\ n) =$ $ite(c_e, \mathsf{tr}'(e_2,\ n), tr'(x_1,\ n-1))$

Figure 3.11: Relaxed Lustre translation semantics of the translation function tr' for a program N. In all cases, Lustre streams of type τ (with associated label λ_i begin resolved to a concrete clock stream c_i in the unification step) become uninterpreted functions in $\mathcal{IL}$ of type $\mathbb{N} \rightarrow \tau$ with the same symbol. In the case where the label corresponds to the global clock c_0, these translations simplify to those found in Figure 3.3. Under this translation, Boolean variables that are clocks (they occur in such a position in a when term) are considered false when undefined. For simplicity of the presentation, we consider definitions of streams that are not tuples (x_1 above).

term translation.

Calculating the interpolation at the stream definition level is necessary to properly capture the interpolated behavior of a slower stream. When operating at the global clock, this reduces to the same translation of definitions as in Figure 3.3.

The $\rightarrow$ operator takes special handling when in the context of a slower clock. It is necessary to determine when the slower clock first is true in order to properly initialize the stream; this initial instant is determined though the inclusion of two additional Boolean variables: ev_i, which is true if e_1 has ever been true, and ft_i, which is true only the first time e_1 is true.

Again, tr' is not applicable in cases when clock streams are themselves delayed — in these cases tr' may interpolate a delayed clock with a true signal, incorrectly interpreting this as an "active" stream when it is actually undefined. [2]

These new translation semantics are used to generate a set of equations Δ, as before. The clock labels for the hidden counter example can be seen in Figure 3.12; a corrected translation can be seen in Figure 3.13.

The translation in Section 3.4 above can be seen as a special case of this, where all streams have been labeled with the global (constant true) clock C_g.

---

[2] To allow for clock streams that are themselves delayed, it would be necessary to modify tr' in such a way that a clock stream c that is delayed is *not* interpolated, but instead is interpreted as true only when c is both defined and has the value true, and false at all other instants. This is further complicated in that c may be used in other calculations as a normal Boolean stream, where it *should* be interpolated when its value is undefined. This requires potentially including two definitions for c in our formulas, depending on if it is being used as a clock or not. It is noteworthy that Lustre clock semantics have proved difficult for programmers as well, eventually leading to the adoption of a restricted current/when construct [47]: condact, where $out = \mathsf{condact}(b, N(x), i)$ is equivalent to $out =$ if b then $N(x$ when $b)$ else $i->$ pre out [79].

$$\Delta(n) = \left\{ \begin{array}{l} time_{c_0} = hc0_{c_1} \\ hc0_{c_1} = 0 -> \mathsf{pre}(hc0_{c_1}) + 1 \\ h1time_{c_0} = \mathsf{current} x1_{c_2} \\ x1_{c_2} = hc1_{c_0}\ when\ c_{c_0} \\ hc1_{c_0} = 0 -> \mathsf{pre}(hc1_{c_0}) + 1 \\ h2time_{c_0} = \mathsf{current}(hc2_{c_4}) \\ x2_{c_4} = c_{c_0}\ \mathsf{when}\ c_{c_0} \\ hc2_{c_4} = 0 -> \mathsf{pre}(hc2_{c_4}) + 1 \end{array} \right\}$$

(a)

$$\Delta(n) = \left\{ \begin{array}{l} time_{c_0} = hc0_{c_0} \\ hc0_{c_0} = 0 -> \mathsf{pre}(hc0_{c_0}) + 1 \\ h1time_{c_0} = \mathsf{current} x1_{c} \\ x1_{c} = hc1_{c_0}\ when\ c_{c_0} \\ hc1_{c_0} = 0 -> \mathsf{pre}(hc1_{c_0}) + 1 \\ h2time_{c_0} = \mathsf{current}(hc2_{c}) \\ x2_{c} = c_{c_0}\ \mathsf{when}\ c_{c_0} \\ hc2_{c} = 0 -> \mathsf{pre}(hc2_{c}) + 1 \end{array} \right\}$$

(b)

Figure 3.12: Hidden counter example, with clock labels on stream variables after inlining (a), and after inferring clocks (b). Labels are subscripts.

$$\Delta(n) = \left\{ \begin{array}{l} time(n) = hc0(n) \\ hc0(n) = ite(n = 0, 0, hc0(n-1) + 1) \\ h1time(n) = x_1(n) \\ x1(n) = ite(c(n), hc1(n), hc1(n-1)) \\ hc1(n) = ite(n = 0, 0, hc1(n-1) + 1) \\ ev_c(n) = ite(n = 0, c(n), c(n) \vee ev_c(n-1)) \\ ft_c(n) = ite(n = 0, c(n), ev_c(n) \wedge \neg ev_c(n-1)) \\ hc2(n) = \left\{ \begin{array}{l} ite(c(n), \\ \quad ite(ft_c(n), \\ \quad\quad 0, \\ \quad\quad hc2(n-1) + 1), \\ \quad hc2(n-1)) \end{array} \right\} \\ x2(n) = ite(c(n), x2(n), x2(n-1)) \\ h2time(n) = hc2(n) \end{array} \right\}$$

(a)

	global clock										
stream	0	1	2	3	4	5	6	7	8	9	...
c	F	F	T	F	F	T	F	F	T	F	...
ev_c	F	F	T	T	T	T	T	T	T	T	...
ft_c	F	F	T	F	F	F	F	F	F	F	...
$hc0$	0	1	2	3	4	5	6	7	8	9	...
$time$	0	1	2	3	4	5	6	7	8	9	...
$hc1$	0	1	2	3	4	5	6	7	8	9	...
$x1$	-	-	2	2	2	5	5	5	8	8	...
$h1time$	-	-	2	2	2	5	5	5	8	8	...
$hc2$	-	-	0	0	0	1	1	1	2	2	...
$x2$	-	-	T	F	F	T	F	F	T	F	...
$h2time$	-	-	0	0	0	1	1	1	2	2	...

(b)

Figure 3.13: Hidden counter example, corrected translation after simplification (a), with sample timing diagram (b). Note $x2$ is technically a clock, so its translation is guarded.

3.7 Summary

This section introduced the language Lustre, and explained its features. It also introduced our translation from Lustre into $\mathcal{IL}$, as well as a novel extension to this translation that handles current and when in a more general case.

CHAPTER 4
K-INDUCTION OVER LUSTRE

4.1 Introduction

In this chapter we introduce our method for verifying safety properties in Lustre, by first describing a basic k-induction algorithm we devised for this process. Additionally we describe several orthogonal improvements on the basic algorithm that we have devised or adapted from the literature.

In the next chapter we will discuss two more extensive modifications to the algorithm: path compression and abstraction.

4.1.1 Related Work

Our work is closest to Franzén's [41] which extends SAT-based k-induction to produce a safety property verifier for Lustre programs with unbounded integers. There, the extension is achieved by adding to the MiniSat SAT solver some simple ILP procedures for handling integer constraints. The focus of [41] was building the extended SAT solver. In contrast, we rely on much more powerful off-the-shelf embeddable SMT solvers that can also handle linear rational arithmetic constraints, and work more on improving the k-induction procedure. Other work has been performed on BMC [42].

Among the additional optimizations, ITE-elimination and inlining are similar to the partial evaluation or inlining commonly used in compilers. Our exploration into quantified path restrictions is based on work by de Mourna, et al. in [34]. Slicing

and the use of the cone of influence to eliminate variables have long been used in software and hardware analysis and model checking (e.g. [81, 57, 80, 23, 12]). These techniques are similar, but we differentiate the two by referring to slicing as a static pre-computation, while the cone of influence reduction is applied dynamically.

4.2 Background

Similar to the general application of k-induction based verification of transition systems (Section 2.6), we can prove P is invariant for N if we succeed in proving the validity of the following two formulas for some $k \geq 1$. For the rest of this chapter and the next, let N be a single-node Lustre program with variables $\overline{x}$, and let $\Delta(n)$ be the equational system modeling N in $\mathcal{IL}$. Let P be a property of N's configurations expressible by a quantifier-free formula $P(n)$ of $\mathcal{IL}$ over $\overline{x}(n)$. If t is any integer term of $\mathcal{IL}$, $P(t)$ is the formula obtained from $P(n)$ by replacing every occurrence of n with t. Similarly for $\Delta(t)$.

$$\Delta(0) \wedge \ldots \wedge \Delta(k) \models_{\mathcal{IL}} P(0) \wedge \ldots \wedge P(k) \tag{4.1}$$

$$\Delta(\mathsf{n}) \wedge \ldots \wedge \Delta(\mathsf{n}+k+1) \wedge P(\mathsf{n}) \wedge \ldots \wedge P(\mathsf{n}+k) \models_{\mathcal{IL}} P(\mathsf{n}+k+1) \tag{4.2}$$

were $\models_{\mathcal{IL}}$ denotes logical entailment in $\mathcal{IL}$ and n is an uninterpreted integer constant.

For the procedures in the remaining sections of this chapter and the next, for simplicity we will assume that any program examined will have a memory depth of at most one. This is possible by flattening the input Lustre program, a simple rewriting of the program where we replace any non-variable expression t within a pre

expression with a fresh variable x, and adding the definition equation $x = t$ to the Lustre program, to completion. Additionally, if the property is also written as part of the program being checked as a synchronous observer, this preprocessing step allows the program to be effectively modeled by a two-state system, a common construct in model checking, such as in [20].

Also these procedures will often deal with counterexamples. Recall that a path $\pi = \overline{v}_0, \ldots, \overline{v}_{k+1}$ is *initial* for N if it is on a legal trace, $k \geq -1$, and $\overline{v}_0$ is an initial configuration. For the rest of this chapter, a *counterexample* is an initial path π where P holds for all $\overline{v}_0, \ldots, \overline{v}_k$, but does not hold for $\overline{v}_{k+1}$.

In this chapter, we will introduce a procedure based on formulas (4.1) and (4.2) that proves invariance for properties of Lustre programs and extends it with several enhancements. We will also provide some additional implementation details.

4.3 Basic Inductive Procedure

Both tests (4.1) and (4.2) can be decided by current SMT solvers that support $\mathcal{IL}$. To verify P we ask the solver to prove both cases for some initial k, retrying with larger k until either the base case (4.1) is proven invalid or else the inductive step case (4.2) is proven valid. In the event of the former, P is not invariant, and provided the solver is able to return models, it is possible to extract a counterexample path from a $\mathcal{IL}$ model of $\Delta(0) \wedge \ldots \wedge \Delta(k) \wedge \neg(P(0) \wedge \ldots \wedge P(k))$. In the latter case, P will have been shown to hold for all reachable configurations, and so is invariant. We will call the above the *basic inductive procedure*. Note that performing this procedure

with just (4.1) and not (4.2) is essentially Bounded Model Checking.

The basic inductive procedure is *sound*, meaning it never falsely claims a property to be invariant. It is not, however, *complete* for all Lustre programs, nor can it be made so in general: for some properties and programs it is possible that the procedure may loop indefinitely, ever increasing k.

Failure to catch a counterexample in 4.1 may result in unsoundness. Failure to prove validity of 4.2 may result in incompleteness. For future reference, we will call comparisons of completeness *accuracy*, a partial ordering over algorithms. A verification algorithm $A1$ is *as* accurate as a verification algorithm $A2$ if the programs $A2$ can correctly verify are a subset of the programs $A1$ can verify. Algorithm $A1$ is *more* accurate as algorithm $A2$ if the programs $A2$ can correctly verify are a proper subset of the programs $A1$ can verify.

Pseudocode for the basic inductive procedure is shown in Figure 4.1. It, along with subsequent versions, uses two global sets of formulas, initially empty, which we call *contexts*: Γ_{base} and Γ_{step}. A formula is *asserted* in context Γ_{base} by adding it to Γ_{base}. For context Γ_{base} and formula ϕ, the *entailment* $\Gamma_{base} \models_{\mathcal{IL}} \phi$ holds iff every model of $\mathcal{IL}$ that satisfies all formulas in Γ_{base} also satisfies ϕ. The function $assert_{base}(\phi)$ adds the formula ϕ to Γ_{base}. $entailed_{base}(\phi)$ tests if $\Gamma_{base} \models_{\mathcal{IL}} \phi$, and if not, then $cex_{base}(\phi)$ returns a counterexample formula A that satisfies Γ_{base} but falsifies ϕ. Specifically, $cex_{base}(\phi)$ returns A such that

(* initialize base *)
$assert_{base}(\Delta(0) \wedge \Delta(1))$;
(* verify initial base *)
if $\neg entailed_{base}(P(0) \wedge P(1))$ then return $cex_{base}(P(0) \wedge P(1))$;
(* initialize step *)
$assert_{step}(\Delta(n) \wedge \Delta(n+1) \wedge \Delta(n+2) \wedge P(n) \wedge P(n+1))$;
$k := 1$;
while true do
    (* INV: $\Delta(0) \wedge \ldots \wedge \Delta(k) \models_{\mathcal{IL}} P(0) \wedge \ldots \wedge P(k)$ ***)**
    (* INV: $\left\{ \begin{array}{c} \Delta(n) \wedge \ldots \wedge \Delta(n+k+1) \wedge \\ P(n) \wedge \ldots \wedge P(n+k) \end{array} \right. \not\models_{\mathcal{IL}} P(n+k+1) \Big\}$ ***)**
    $k := k + 1$;
    (* Base case / BMC *)
    $assert_{base}(\Delta(k))$;
    if $\neg entailed_{base}(P(k))$ then return $cex_{base}(P(k))$;
    (* Step case *)
    $assert_{step}(\Delta(n+k+1) \wedge P(n+k))$;
    if $entailed_{step}(P(n+k+1))$ then return Valid;
done

Figure 4.1: Base k-induction algorithm. P is the property to be checked, Δ is the program, both in $\mathcal{IL}$. Assertions, entailment checks, and counterexample extractions ($cex_x(P_i)$) are with respect to a specific solver, either *base* case or inductive *step* case. Counterexamples are with respect to the negated property P_i. Note this assumes that the program's memory depth is at most 1.

1. A, Γ_{base} is $\mathcal{IL}$ satisfiable
2. $A, \Gamma_{base} \models_{\mathcal{IL}} \neg\phi$

and

$$A = \left\{ \begin{array}{l} \overline{x}(0) = \overline{v}_0 \\ \vdots \\ \overline{x}(k+1) = \overline{v}_{k+1} \end{array} \right\}$$

where for $i = 1, \ldots, k+1$, v_i is a well-typed tuple of values for $\overline{x}$. Notice string $\pi = \overline{v}_0, \ldots, \overline{v}_{k+1}$ is a counterexample in our sense.

Similarly for the *step* functions, applied to context Γ_{step}.

Note that tests (4.1) and (4.2) can be executed by a SMT solver. Also note that (4.1) and (4.2) are independent. Similarly in the pseudocode, all calls associated with context Γ_{base} are independent from all calls associated with Γ_{step}. Therefore the algorithm actually assumes the use of two instances of the SMT solver, one for Γ_{base} and one for Γ_{step}.

4.3.1 Implementation Note: SMT Solver Features

We take advantage of current technologies by implementing these algorithms with solvers that include several features common to lazy SMT solvers (see Section 2.7.3), including being on-line, incremental, and backtrackable, being able to return models and compute unsatisfiable cores, and being able to remember learned lemmas.

An *on-line* solver integrates the decision procedures into the Boolean SAT engine in an active fashion; the status of literals is updated in the decision procedure as literal assignment choices are made, and literals entailed by the theory are returned to the SAT solver, a feature vital to the performance of the lazy SMT approach [43].

Because we are checking a series of formulas that differ primarily in the ad-

dition of constraints between steps, it can be beneficial to use an *incremental* and *backtrackable* solver. Such a solver keeps a notion of a context, as above, in which the validity of a query formula is resolved modulo that context. Additional constraint formulas may be freely *asserted*, or added to the context, and assertions may be retracted as well. This backtracking is necessary to undo assertions that make the context unsatisfiable, as queries on any formula past that point will not provide additional useful information. It can also be used to retract previous assertions that may change or no longer be relevant to the new context. In the given pseudocode we assume that contexts can be saved at arbitrary points and backtracked as needed in a stack-like fashion. We also assume that backtracking will be done as necessary (such as immediately following a non-falsified entailment check), even if it is not explicitly mentioned in the algorithms.

The ability to preserve learned lemmas can also enhance the performance of incremental queries, reducing the possibility of wasting work on re-doing a failed search.

The ability to compute (possibly partial) models for an invalid formula, on the other hand, is necessary to be able to return a useful counterexample to the user. Computing unsatisfiable cores is generally an extension of the solver's conflict analysis, and necessary for our algorithms involving abstraction refinement (see Section 5.3.4).

4.3.2 Implementation Note: Memory depth

The actual implementation does not fully flatten the program with respect to pres. Instead it bases the starting k value on the program's maximum memory depth d. This more accurately preserves the structure of the program as envisioned by the programmer and can also reduce the number of variables appearing in Δ. In this case, the initialization step of the program includes $\Delta(0)$ through $\Delta(d+1)$, and tests the base case for $P(0)$ through $P(d)$. Similarly the step case is initialized through $n+d+1$.

This means that we are actually checking properties over *paths*, not just over *configurations*.

4.4 Additional Extensions

We will now examine some additional extensions to the basic procedure that are generally orthogonal to path compression and abstraction.

4.4.1 Quantified Path Restrictions

In addition to inspiring the path compression in the next chapter, [34] (and in a less general situation, [3]) also presents a strengthening technique that attempts to eliminate from the search those paths that would lead to configurations that do not preserve the property but are unreachable from an initial state. In our terms, these *bad paths* are unreachable paths that invalidate test (4.2). These paths are problematic because they force a strengthening of our hypothesis (by incrementing k) when there is not really a counterexample. An alternative approach is to strengthen the property

in order to proactively exclude any bad paths from the search.

When test (4.2) fails for some value n for k, we postulate that all paths that invalidate (4.2) are, in fact, unreachable, and strengthen P accordingly. If our postulation is true and all such paths are indeed unreachable, then we have reduced the search space. If, however, any of these paths turn out to be reachable, then the strengthened property will fail under test (4.1) at a lower value of k then might normally be needed. Either way, then, this sort of strengthening reduces the amount of search.

Recall under the simplifying assumption of a program N of depth ≤ 1, $\Delta(n)$ stands for the transition relation from configuration $\overline{x}(n-1)$ to configuration $\overline{x}(n)$. Let $T(\overline{x}_{n-1}, \overline{x}_n)$ denote such a transition. A bad path then is one that falsifies the formula

$$T(\overline{x}_{n-1}, \overline{x}_n) \wedge \ldots \wedge T(\overline{x}_{n+k}, \overline{x}_{n+k+1}) \wedge P(\overline{x}_n) \wedge \ldots \wedge P(\overline{x}_{n+k}) \rightarrow P(\overline{x}_{n+k+1})$$

Since we are concerned with *all* paths ending in a failure of the property, this can be simplified to

$$T(\overline{x}_{n-1}, \overline{x}_n) \wedge \ldots \wedge T(\overline{x}_{n+k}, \overline{x}_{n+k+1}) \rightarrow P(\overline{x}_{n+k+1})$$

Therefore, a configuration that starts a bad path will satisfy the formula $\psi(\overline{z})$:

$$\psi(\overline{z}) \iff \exists \overline{z}_n \ldots \overline{z}_{n+k+1}. \, (T(\overline{z}, \overline{z}_n) \wedge \ldots \wedge T(\overline{z}_{n+k}, \overline{z}_{n+k+1}) \wedge \neg P(\overline{z}_{n+k+1}))$$

where each $\overline{z}_i$ is a tuple of fresh variables representing the configuration $\overline{x}_i$ and $\overline{z}$ represents the current configuration. With $\psi(\overline{x})$ it is possible to reach, from configuration $\overline{x}$, a configuration that falsifies P in $k+1$ steps. We do not want this, so we

strengthen $P(\overline{z})$ to be $P(\overline{z}) \wedge \neg\psi(\overline{z})$: now every (reachable) configuration must not only satisfy P, but also not lead to a counterexample in $k+1$ steps.

Notice that this formula has universal quantification, something that is problematic for many SMT solvers. In [34] and [3] this new formula $\psi(\overline{z})$ is rendered tractable via quantifier elimination techniques. To give an idea, due to the equational nature of Δ and so of T, it is possible to immediately simplify ψ, eliminating the quantification on many of the variables using the logical equivalence:

$$\exists x.(\phi(x) \wedge x = t) \iff \phi(t)$$

The quantification in ψ on all local variables can be removed through such rewriting, however notice variables corresponding to input streams will remain quantified. In [34] and [3] the case of formulas with Boolean (or other small-domain) variables is examined as there exist relatively simple methods of completely eliminating quantification on such formulas. In $\mathcal{IL}$, however, formulas may also include terms of unbounded types such as integers and these methods are not generally applicable.

The SMT solvers we use do have some support for quantified formulas, and we tried using them this way. Unfortunately since these solvers are incomplete with quantified formulas, they often returned results of "don't know" or failed to terminate. As a result, we did not pursue this avenue further.

4.4.2 Static Analysis

There are a number of optimizations that can be performed via static analysis of the program N and the property P to be verified.

4.4.2.1 Slicing

Slicing is a means of eliminating variables that are not relevant to the property at hand, slicing away unnecessary information. This is a method of simplification long used in software analysis as a means of focusing only on relevant sections of code [81], and can similarly be applied in hardware to focus on sub-circuits. In our case, slicing can be done with a simple preprocessing step that traces stream variable dependencies through the definitions of Δ . Variables that can be traced back to a variable mentioned in the property are kept, other variables are discarded by removing their definitions from Δ. In general, reducing the number of variables in the problem increases the efficiency of the SMT solvers, sometimes significantly.

We use a simplistic version of slicing where we build Δ incrementally through a preprocessing step that is roughly similar to our method of structural abstraction and refinement (Section 5.3). Δ begins empty; and we then include all definitions of variables that occur in P. We then recursively include definitions for all variables that occur in the right-hand side of included definitions.

Despite using this rather naive approach, even if further analysis would reveal that some included variables are never actually used, this technique proved fast, easy, and effective in experiments.

4.4.2.2 Cone of Influence

Differentiating this from the naive approach obtained under slicing, above, full *cone of influence reduction* [23, 12] is a method often used to eliminate extraneous

variables in symbolic model checking. The same general approach is taken (only include variables that can be traced back from the property), but the formulas involved are determined *dynamically* as the program's transformation relation is unwound. Ideally this would only include variables that are actually relevant for proving the property P.

The tightest possible cone of influence may require dynamic evaluation of certain variables. As we are not interested in explicitly simulating the program in question, we instead utilized a looser cone of influence similar to our ITE elimination, below — depending on the value of k, we include different (increasing number of) definitions in Δ. It is possible to determine the stream variables likely to be relevant to the property by tracing their dependencies, similar to slicing, but also take the depth at which a stream variable is referenced into account (with our slicing, we merely use the presence or absence of a variable in a definition). For example, if the property depends on x, defined as `x = 1 -> pre (1 -> pre y + 1)`, and x was the only stream that depended on y, y would not actually be useful in proving the property until after x had been initialized, at $k \geq 2$. Because Lustre programs are finite, they must have a finite depth d, and so it is possible to determine the stream variables likely to be used in the first d steps, and modify $\Delta(0), \ldots, \Delta(d)$ accordingly.

Trials of this optimization tended to provide a slight degradation to performance in our experiments compared with the static slicing, possibly due to the altered search-space or added overhead — most programs reached full definition within the first one or two k steps.

4.4.2.3 ITE-elimination

It is possible to prune certain portions of the search space by observing the behavior of if-then-else (ITE) expressions in Δ. While in general it is necessary to evaluate both possible branches of an ITE expression, in certain circumstances it is possible to statically determine which branch will be taken. In the case of Lustre programs expressed in $\mathcal{IL}$, the formulas representing guarded pres are encoded as ITE expressions. For the base case of the induction and most instances of the step case it is possible to re-write the formula, eliminating some ITEs.

For example, for programs N with a pre depth of at most 1 (the general case is similar), it is possible to simplify the translation of N into the formulas $\Delta(0)$, $\Delta(i)$, $\Delta(m)$, and $\Delta(m+j)$, where m is a free integer constant standing for a non-negative integer and i and j are concrete numerals $(0, 1, 2, \ldots)$, with $i \neq 0$. Let n be the position meta-variable introduced in the translation. For $\Delta(0)$ we know that $n = 0$ and for $\Delta(i)$ and $\Delta(m+j)$ we know that $n > 0$.

For example, consider a definition like

$$z(n) = \mathsf{if}\ (n = 0)\ \mathsf{then}\ x(0)\ \mathsf{else}\ y(n)$$

If this definition appears in $\Delta(0)$, then we know that $n = 0$, and the expression can be simplified to $z(0) = x(0)$. If this definition appears in $\Delta(i)$, then we know that it can be simplified to $z(i) = y(i)$, and similarly for $\Delta(m+j)$. The remaining case, $\Delta(m)$, we cannot simplify.

The effect of this optimization actually turned out to be highly problem-dependant, as changing the formulas tends to modify the search space. In some

```
node test (x:bool) returns (OK:bool);
var ga,gb:bool;
    itime: int;
let
  ga = false -> not pre(gb);
  gb = false -> pre(ga);
  itime = 0 -> if pre(itime) = 3 then 0
           else pre itime + 1;
  OK = (x and ga and gb) = (x and (itime = 2));
tel
```

Figure 4.2: Lustre counter with aggressively inlined code. Compare with Figure 3.5.

cases it produced some speedup, in others some slowdown. In general, however, the changes in overall times for our benchmark set were often not very dramatic due to the use of this feature.

4.4.2.4 Inlining

Additionally, it is possible to easily inline certain variable definitions in the formula as a preprocessing step, rewriting Δ as necessary. We attempted several degrees of inlining, from none at all, to only inlining definitions consisting of a single variable such as $x = y$, to inlining most definitions that did not depend on previous values.

In theory inlining can reduce the number of variables in the formula, hopefully leading to faster verifications. Our experiments show that having too many variables can degrade performance in the tested SMT solvers. On the other hand, overly aggressive inlining can result in an exponential explosion of the size of definitions in Δ.

An example of more aggressive inlining can be found in Figure 4.2, based off the code from Figure 3.5.

We observed in our experiments that this technique, as expected, is not orthogonal with abstraction, covered in Section 5.3. While only inlining formulas of the form $x = y$ was beneficial even when used with abstraction, more intensive inlining often reduced the effectiveness of abstraction. This is at least partly due to the fact that our abstraction relies on the definitions of variables. Aggressive inlining focuses on removing as many definitions as possible, meaning there is less to be abstracted, and since the remaining definitions are generally more complicated, any refinements are more coarse in nature.

4.4.3 Skipping Steps

The basic k-induction procedure allows k to be increased by arbitrary increments, essentially skipping some step tests. The step test (4.2) and its derivatives, especially, can be computationally complex, meaning that not checking every value k can provide significant efficiency advantages, at least in principle.

While on the surface utilizing large k increments may seem advantageous – fewer steps means less work – there is are tradeoffs involved. In the base case, using a k increment greater than 1 means the procedure is not guaranteed to find the *shortest* possible counterexample. Additionally, the induction step is generally an expensive test, which grows more expensive as k increases. It is possible to overshoot the minimal k value to discover validity; in more severe cases, this attempt to solve a

problem at a larger k may reach the resource limitations of the computer before the problem is solved.

We experimented with be incrementing k by a fixed amount, and began experimenting with increasing k an increasing amount between steps. Either of these can improve the algorithm's performance, depending on the problem in question.

4.5 Summary

This chapter introduced our k-induction algorithm for verifying Lustre programs translated into $\mathcal{IL}$. Additionally it discussed several modifications to this algorithm that we adapted from the literature.

CHAPTER 5
ALGORITHM EXTENSIONS

5.1 Introduction

In this chapter we describe two significant improvements to the algorithm described in Section 4.3, path compression and abstraction, here implemented in terms of our logic $\mathcal{IL}$.

5.1.1 Related Work

Our path compression method of invariant strengthening can be seen as a special case of the one described by de Moura, *et al.* [34]. Additionally Sheeran, et al. describe a similar method of termination checking in [70], though applied to Boolean systems.

A dependency-based abstraction method analogous to ours is described by Chan, *et al.* [18] for the dataflow language RSML. There, however, abstractions appear to be generated statically as a preprocessing step, and not refined dynamically as in our case. Our abstraction method is more similar to the ones developed by Clarke, et al. [29] and Babic and Hu [4], which use a SAT solver as their reasoning engines. Work by Vecchié and Simone[78] addresses an overall similar approach for BDD-based systems based on Esterel, though this appears to be more of a cone of influence calculation. Work by Gupta, *et al.* [45, 44] has also been done with respect to abstraction, primarily in the context of BMC. The main idea of our method however—treating local variables as inputs and refining based on spurious counterexamples—goes back to

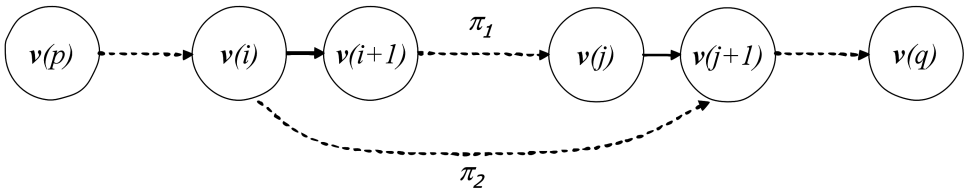

Figure 5.1: Path compression. The path $\pi_2 = \overline{v}(p) \ldots \overline{v}(i)\overline{v}(j+1) \ldots \overline{v}(q)$ is strictly shorter than the *compressible* path $\pi_1 = \overline{v}(p) \ldots \overline{v}(i)\overline{v}(i+1) \ldots \overline{v}(j)\overline{v}(j+1) \ldots \overline{v}(q)$. Assume $\overline{v}(i) =_s \overline{v}(j)$ and π_1 is reachable. Then π_2 is a reachable path.

Kurshan [57] and appears in various forms in several works on hardware verification. Our use of unsatisfiable cores in our refinement heuristic recalls a similar approach by Chauhan, *et al.* [19] for SAT-based model checking.

5.2 Path Compression

One major enhancement on the basic procedure is the use of *path compression* [70, 34], a means of eliminating redundant search. For Lustre programs, path compression is achieved by strengthening the left-hand side of the entailments in (4.1) and (4.2) to eliminate paths that contain repeated configurations (due to a cycle), or more generally, configurations that are *equivalent*, as described below. Our approach is based on [70], lifted to SMT. A similar approach appears to be taken in [22] (and by extension [40]).

To simplify the description of path compression here, we first assume that a program N has been normalized so that pre only applies to stream variables, as mentioned in the previous chapter. Those variables that have a pre applied to them we call *state variables*. The *state* of a configuration $\overline{v}$ is the subtuple of $\overline{v}$ consisting

only of values of $\overline{v}$ corresponding to $\overline{v}$'s state variables.

Recall that it is possible to look at Δ as a transition relation between two configurations, from configurations at instant $n-1$ to configurations at instant n.

Two configurations $\overline{v}_i$ and $\overline{v}_j$ are *equivalent*, if they have the same set of successors in Δ. This notion is interesting to us because it can be used to prune the search space we are exploring. The intuition is that if all configurations in a reachable path satisfy the property P to be proven, and the path contains two equivalent configurations, it can be *compressed*, or represented by a strictly shorter reachable path that also satisfies the property. An example of this can be seen in Figure 5.1.

We could strengthen the left-hand side of (4.2) so that it is satisfied only by paths without equivalent configurations without impacting soundness and completeness. Unfortunately, the full notion of equivalence is not expressible by quantifier-free formulas, and we are limited by tools that do not handle arbitrary quantified formulas well. As a consequence we use a more restrictive notion of equivalence, one based on the configuration's states: let $\overline{z}_i$ be the state of configuration $\overline{v}_i$ and $\overline{z}_j$ be the state of configuration $\overline{v}_j$. We write $\overline{v}_i =_s \overline{v}_j$ iff $\overline{z}_i = \overline{z}_j$. This equivalence relation is expressible by a quantifier-free formula and implies equivalence in the sense above, as expressed in the following lemma.

Lemma 1 *If $\overline{v}_i =_s \overline{v}_j$, then $\overline{v}_i$ and $\overline{v}_j$ are equivalent.*

Proof: Assume $\overline{v}_i =_s \overline{v}_j$. Then we will show $\overline{v}_i$ and $\overline{v}_j$ will have the same set of next configurations. This can be determined by comparing the transition relations $\Delta(i+1)$ and $\Delta(j+1)$. Values in $\overline{v}_i, \overline{v}_j$ fall into three classes: input values, state values, and

other values. Input values are unconstrained in both cases, and may effectively be discounted. The values corresponding to state variables in $\overline{v}_i, \overline{v}_j$ are identical, by the assumption. And all other values depend, ultimately, on either input values or state values. Therefore $\Delta(i+1)$ will have the same set of next configurations as $\Delta(j+1)$.

□

So we strengthen (4.2), saying that different configurations on a path from position n to position $n+k$ do not have the same state, expressed by a formula $C_{n,k}$. We can also strengthen this a little further by saying that no configurations (except possibly the first) have the same state and an initial configuration. More formally, we say that a path is *compressed* if no two configurations in it have the same state and if no configurations (except possibly the first) are initial, otherwise it is *compressible*. For $i < j$, let $\mathit{distinct}(i,j) = (\overline{z}_i \neq \overline{z}_j) \wedge \ldots \wedge (\overline{z}_{j-1} \neq \overline{z}_j)$, and let

$$C_{n,k} = \Delta_0 \wedge \bigwedge_{0 \leq i < j \leq k} (\overline{z}_0 \neq \overline{z}_{n+j} \wedge \mathit{distinct}(n+i, n+j))$$

The formula $C_{n,k}$ is satisfiable only by paths that are compressed from position n to position $n+k$.

Note that by adding $C_{n,k}$ to (4.2) we compress only those paths that satisfy the property, up to (but possibly not including) the path's final configuration.

The new procedure with path compression performs test (4.1), followed by the new version of (4.2), expressed as:

$$\begin{array}{l} \Delta(\mathsf{n}) \wedge \ldots \wedge \Delta(k+1) \wedge \\ P(\mathsf{n}) \wedge \ldots \wedge P(\mathsf{n}+k) \wedge C_{n,k} \end{array} \models_{\mathcal{IL}} P(\mathsf{n}+k+1) \qquad (5.1)$$

This preserves the accuracy and soundness of the basic inductive procedure. Accuracy preservation is trivial, as we are simply adding another constraint to the premise of test (4.2), so if (4.2) holds, then (5.1) holds. In the following we prove that soundness is preserved, as well. We first need to define the notion of a compression: given an initial path π, π' is a *compression* of π if π' is compressed and the configurations in π' also occur in π, in the same order. A compression π' of π is *maximal* if it begins with an initial configuration and ends with the same last configuration of π.

Lemma 2 *Every initial path has a maximal compression.*

Proof: Assume π is an initial path. If π is compressed, then π is its own maximal compression, by definition. If π is not compressed, then there are two possibilities:

Case 1. Compression within π: Assume $\pi = \overline{v}_0, \ldots, \overline{v}_i, \ldots, \overline{v}_j, \ldots, \overline{v}_{q+1}$, with $0 \leq i < j \leq q$, and $\overline{v}_i$ and $\overline{v}_j$ having the same state. Then $\overline{v}_i$ and $\overline{v}_j$ have the same set of next configurations, by Lemma 1. Then $\pi' = \overline{v}_0, \ldots, \overline{v}_i, \overline{v}_{j+1}, \ldots, \overline{v}_{q+1}$ is an initial path strictly shorter than π, with configurations in π' being in the same order as in π.

Case 2. Compression with initial configuration: Let $\pi = \overline{v}_0, \ldots, \overline{v}_i, \ldots, \overline{v}_{q+1}$, with $0 < i \leq q$ and $\overline{v}_i$ has the same state as some initial configuration. Then $\pi' = \overline{v}_i, \ldots, \overline{v}_{q+1}$ is an initial path strictly shorter than π, with configurations in π' being in the same order as in π.

In either case, π' is a shorter path. If we apply the same argument to π', then it is clear that we will eventually obtain a maximal compression of π. □

Lemma 3 *Let π be an initial path and π' a maximal compression of π. Then, if π is a counterexample for program N, so is π'.*

Proof: This is a direct consequence of the definition of a maximal compression. □

Theorem 1 *The extension of the basic induction procedure using formula (4.1) and formula (5.1) is sound.*

Proof: By contradiction: Assume the extended procedure succeeds at step k, meaning (4.1) succeeds for all reachable paths up to length k, and (5.1) succeeds for step k, but there exists a counterexample *pi* (of any length) for N.

Take π and let $\pi' = \overline{v}_0, \ldots, \overline{v}_{q+1}, q \geq 0$ be its maximal compression. For this initial path, P holds for $\overline{v}_0, \ldots, \overline{v}_q$, but does not hold for $\overline{v}_{q+1}$, by Lemma 3.

If $q \leq k$, the entailment in (4.1) does not hold in the current step of the procedure, a contradiction.

If $q > k$, consider the end segment of π' of length k: $\overline{v}_{q+1-k}, \ldots, \overline{v}_{q+1}$. In this segment $\overline{v}_{q+1-k}, \ldots, \overline{v}_q$ satisfy P but $\overline{v}_{q+1}$ does not. This invalidates (5.1), again a contradiction. □

In addition to helping the solver to prove $P(\mathsf{n}+k+1)$, restricting the process to compressed paths makes it a complete k-induction procedure when the length of reachable compressed paths has a (finite) upper bound. Completeness in this case is achieved by checking (4.1) and (5.1) for consecutive values of k starting at 0, with the addition of a check to see if there are any initial (compressed) paths of length $k+1$.

This is done by checking whether the entailment of the formula

$$\Delta(0) \wedge \ldots \wedge \Delta(k+1) \models_{\mathcal{IL}} \neg C_{1,k+1} \tag{5.2}$$

holds, a test analogous to a loop check in bounded model checking. If this *termination check* succeeds in the procedure, then we may conclude that P is invariant for N.

Theorem 2 *If formula (4.1) and formula (5.2) hold for some k, then P is invariant.*

Proof: We will show that if P holds for all initial paths of N up to length k and formula (5.2) holds for N, then P will hold for all legal traces of N. This can be proved by induction on path length.

Let $\pi = \overline{v}_0, \ldots, \overline{v}_q, \overline{v}_{q+1}$, $q \geq -1$ be an initial path. Assume (4.1) and (5.1) hold for π up to the current k.

By (4.1) we know P holds for all paths of length k or less.

Inductive base: If $q < k$ then, by the above, P holds for π.

Inductive step: Assume P holds for all reachable paths of length q, $q \geq k$. We will show it holds for π, with length $q+1$.

By contradiction, instead assume that π is a counterexample where P holds for all $\overline{v}_0, \ldots, \overline{v}_q$, but does not hold for $\overline{v}_{q+1}$, again with $q \geq k$. By the assumptions there are no counterexamples shorter than π.

Because (5.2) holds, we know all paths of length greater than k are compressible. Let π' be a maximal compression of π. As shown in Lemma 2, π' is strictly shorter than π, and it still ends with $\overline{v}_{q+1}$, for which P does not hold, meaning it

(* initialize base *)
$assert_{base}(\Delta(0) \wedge \Delta(1))$;
(* verify initial base *)
if $\neg entailed_{base}(P(0) \wedge P(1))$ then return $cex_{base}(P(0) \wedge P(1))$;
(* initialize step *)
$assert_{step}(\Delta(0) \wedge \Delta(n) \wedge \Delta(n+1) \wedge \Delta(n+2) \wedge P(n) \wedge P(n+1))$;
$k := 1$;
while true do
    begin
        (* INV: $\Delta(0) \wedge \ldots \wedge \Delta(k) \models_{\mathcal{IL}} P(0) \wedge \ldots \wedge P(k)$ ***)**
        (* INV: $\left\{ \begin{array}{ll} \Delta(n+1) \wedge \ldots \wedge \Delta(n+k+1) \wedge & \\ P(n) \wedge \ldots \wedge P(n+k) \wedge C_{n,k} & \not\models_{\mathcal{IL}} P(n+k+1) \end{array} \right\}$ ***)**
        $k := k + 1$;
        $assert_{base}(\Delta(k))$;
        (* Termination check *)
        if $entailed_{base}(\neg(distinct(0,2) \wedge \cdots \wedge distinct(0,k))$ then return Valid;
        (* Base case / BMC *)
        if $\neg entailed_{base}(P(k))$ then return $cex_{base}(P(k))$;
        (* Step case *)
        $assert_{step}(\Delta(n+k+1) \wedge P(n+k))$;
        (* Path compression *)
        $assert_{step}(distinct(n, n+k))$;
        if $entailed_{step}(P(n+k+1))$ then return Valid;
    end

Figure 5.2: Base k-induction algorithm with path compression. For $i < j$, the predicate $distinct(i.j)$ is true if all configurations $\Delta(0)$ and $\Delta(i), \ldots, \Delta(j-1)$ are pairwise distinct (for their stateful streams) with $\Delta(j)$, and false otherwise. Other functions are as in Figure 4.1

is a counterexample. But this is a contradiction, as there are no counterexamples of length q or less. Therefore P must hold for $\overline{v}_{q+1}$, and so for all of π.

□

The pseudocode from Figure (4.1) is extended to include path compression and the termination check in Figure 5.2. In this algorithm $C_{n,k}$ is represented by a combination of $distinct(n, n+k)$ predicates, where $distinct(n, n+k)$ is true iff

$\overline{z}(0) \neq \overline{z}(n+k)$ and $\overline{z}(n) \neq \overline{z}(n+k) \wedge \overline{z}(n+1) \neq \overline{z}(n+k) \wedge \ldots \wedge \overline{z}(n+k-1) \neq \overline{z}(n+k)$, for $i < j$, and N's state variables $\overline{z}$.

5.2.1 Implementation

As mentioned before, the implementation actually makes use of the program's memory depth d. This means that the concept of "state" can be spread out over a number of different consecutive configurations in a trace, specifically the current configuration and up to d prior ones ($d > 0$, if $d = 0$, the program is stateless). $distinct(i, j)$ in cases where $d > 1$ can also refer to state variables with a (negative) offset to n. In this case it is necessary to ensure that $j - i > d$ for N's memory depth d, as prior to that streams may be assigned specific values (due to $\rightarrow$ operators) and so not have a consistent sense of "state"[1].

As concrete examples of *distinct* predicates, the counter comparison example from Section 3.4.1 (Figure 3.6) has a memory depth $d = 1$, with (inlined) state variables ga, gb, and $itime$, all with an offset of 1. In this case $distinct(i, j) = (\langle ga(i), gb(i), itime(i)\rangle \neq \langle ga(j), gb(j), itime(j)\rangle)$. The Fibonacci example from the same section (Figure 3.7), however has a pre nesting depth of 2; it is necessary to include stream variable y for both of its offsets: $distinct(i, j) = (\langle y(i), y(i-1)\rangle \neq \langle y(j), y(j-1)\rangle)$.

[1] In some cases this set of state variables can be reduced. By tracing the dependencies of variables within pre expressions it is sometimes possible to eliminate some redundancies.

5.2.2 when and Path Compression

The above assumes that we only allow a minimal usage of when operators.

We do not perform path compression before the memory depth d, as variables that might otherwise be considered stateful can be assigned arbitrary values through $\rightarrow$ operations for one or more consecutive configurations. The conservative approach allows us a uniform means of comparing states while still guaranteeing soundness. Normally this is trivial to do — count the nesting of pre operators in the program. And the equivalence comparison *distinct* (from Figure 5.2), can be equally trivial: $\overline{s}(i) \neq \overline{s}(j)$.

Unfortunately this d is with respect to the global clock, and (stateful) terms under a when operator do not have a well-defined depth. They might never be executed (and hence never have applicable state), or they might be delayed, so we cannot know ahead of time when they will not longer be guarded by a $\rightarrow$ and should be included in the state equivalence comparison (*distinct* in Figure 5.2).

5.3 Abstraction of Lustre Programs

Abstraction is often used in verification to reduce large or complex verification problems into more manageable ones. One attempts to prove a property P for a program N by proving it for a *conservative* abstraction N' of N. N' simulates N, and accepts at least those behaviors (legal traces) accepted by N, possibly more, and ideally is in some sense simpler than N. For Lustre programs N' may, for instance, consist of a subset of N's equalities. If the property holds for N' then it holds for N

as well, but if the property does not hold for N', then it may be necessary to *refine* N', adding equations to it and bringing it closer to N, and attempt the proof again. We use this basic idea in our verification of Lustre programs.

Specifically, we utilize the same basic process covered in Section 4.3, but with abstracted versions of formulas (4.1) and (4.2):

$$\Delta'(0) \wedge \ldots \wedge \Delta'(k) \models_{\mathcal{IL}} P(0) \wedge \ldots \wedge P(k) \tag{4.1'}$$

and

$$\begin{array}{r} \Delta'(\mathsf{n}) \wedge \ldots \wedge \Delta'(\mathsf{n}+k+1) \wedge \\ P(\mathsf{n}) \wedge \ldots \wedge P(\mathsf{n}+k) \end{array} \models_{\mathcal{IL}} P(\mathsf{n}+k+1) \tag{4.2'}$$

In these abstracted formulas, we use a simplified Δ of N', or Δ'. As before, the formula Δ' defines a set of legal traces (of the abstract version N' of N), but these are guaranteed to include all legal traces defined by the original Δ (and allowed by N). However, Δ' may allow additional traces to Δ. Therefore it is possible that formula (4.1') may be invalid when (4.1) is valid. We call a counterexample $\pi = \overline{v}_0, \ldots, \overline{v}_{k+1}$ for Δ' a *spurious counterexample* for N if it is not also a counterexample for Δ, similarly for (4.2') and (4.2). In the event we discover a spurious counterexample, we recognize that we need to bring Δ' closer to Δ. To do so, we re-try that step of the procedure with a newly refined Δ'.

5.3.1 Structural Abstraction

There are a number of ways to perform abstraction; we will use a form of abstraction often used in hardware analysis, *structural abstraction* [57, 18, 4], as a means of leveraging the structure of a program as a template for the abstraction.

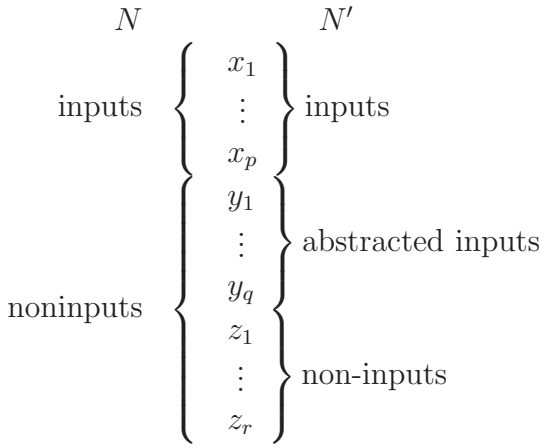

Figure 5.3: Structural abstraction of stream variables. This shows an initial abstraction N' of a program N. $\overline{x} = x_0, \ldots, x_p$ are input variables of N. $\overline{z} = z_0, \ldots, z_r$ are non-input variables in the property. $\overline{y} = y_0, \ldots, y_r$ are all remaining non-input variables of N. Initially $\overline{z}$ are defined in the abstraction, while $\overline{y}$ are treated as inputs.

In structural abstraction, certain non-input streams are left undefined and treated as inputs. The abstraction is refined by adding back definitions for some of these streams. Because a single node Lustre program can be expressed as a set of stream variable definitions, and we inline all node calls in our translation, this provides a ready means of abstracting and refining a program.

The basic idea is that stream variables in Δ' are classified as *refined*, *undefined*, or *true inputs*. Refined and undefined variables correspond to non-input streams in the original program while true inputs are inputs in the original. Refined variables are constrained by their equational definition from N, while undefined variables are not. True inputs have no such definition to begin with, and so remain unconstrained. Initially only the variables occurring in the property are provided definitions and so considered refined, and all others remain undefined. An exam-

ple of this can be seen in Figure 5.3. Here the concrete system consists of stream variables $\langle x_0 \ldots, x_p, y_0, \ldots, y_q, z_0, \ldots, z_r \rangle$, with $\overline{x} = x_0 \ldots, x_p$ being input variables, $\overline{z} = z_0, \ldots, z_r$ being non-input (local and output) variables occurring in the property, and $\overline{y} = y_0, \ldots, y_q$ being all remaining local and output variables. The abstraction initially considers only $\overline{z}$ to be non-input variables while all others are treated as (abstracted) inputs, greatly simplifying the definition of Δ' in (4.1') and (4.2'), when compared with the original Δ. As we attempt to prove a property invariant, it may be necessary to refine Δ'; we do this by choosing one or more variables in $\overline{y}$ and *refining* them, adding their definitions and effectively transforming them from undefined (abstracted inputs) to being refined (non-input) variables.

This strategy follows the general counterexample-guided refinement paradigm (CEGAR) [28], where refinement is based on analysis of spurious counterexamples. In the simplest version of CEGAR, the specific spurious counterexample $\pi = \overline{v}'_0, \ldots, \overline{v}'_{k+1}$ of N' is eliminated through the use of a focused constraint that just negates the formula representing that counterexample. Such a constraint can actually guarantee progress in finite transition systems, reducing the total search space; however, in infinite state systems, eliminating one of a possibly infinite set of spurious counterexamples is not sufficient. Instead of adding constraints to eliminate a specific counterexample, is possible to refine all variables that occur in the counterexample and so (possibly) constrain all these variables by their relation in Δ, effectively eliminating a (possibly infinite) set of spurious counterexamples that include this specific one.

It is possible, however, to do too much refinement and so get an abstraction that contains more detail than is necessary to prove the property, losing some of the simplification that is the benefit of abstraction. So instead we refine only a *few* variables derived from analysis of the counterexample as opposed to all that appear in it.

5.3.2 Basic Procedure with Abstraction/Refinement

Variables in $\overline{z}$ (see Figure 5.3) begin as defined, those in $\overline{y}$ are initially undefined. We then test (4.1') and (4.2') for a given k, similar to the basic procedure. If we find a counterexample for N in (4.1') or (4.2') holds, we may stop, as before. If (4.1') holds and there is a counterexample to N in (4.2'), we increase k and try again. As the procedure progresses, though, it is possible that we may discover a spurious counterexample in (4.1') or (4.2') by receiving an unsatisfiable result from a $sat_{checker}()$ query. This is an indication that our abstraction Δ' is too coarse, and needs to be refined. To do this we choose one or more undefined variables from $\overline{y}$ (the precise variables are determined by a refinement heuristic, see Section 5.3.4) and add their definitions to Δ'. This process continues until we prove the property invariant for the abstracted system or find a counterexample that is also a counterexample for N.

The pseudocode for this *procedure with abstraction/refinement* is shown in Figure 5.4. It differs from the basic procedure in Section 4.3 in three primary respects. First, it utilizes the current version of Δ' in both of the *base* and *step* instances of

(* initialize base *)
$assert_{base}(\Delta'(0) \wedge \Delta'(1))$;
$assert_{checker}(\Delta(0) \wedge \Delta(1))$;
(* verify initial base *)
while $\neg entailed_{base}(P(0) \wedge P(1))$ do
    if $sat_{checker}(cex_{base}(P(0) \wedge P(1)))$ then return $cex_{base}(P(0) \wedge P(1))$
    else $\Delta' := refine(\Delta', core(cex_{base}(P(0) \wedge P(1))))$;
done
(* initialize step *)
$assert_{step}(\Delta'(n) \wedge \Delta'(n+1) \wedge \Delta'(n+2) \wedge P(n) \wedge P(n+1))$;
$assert_{checker}(\Delta(n) \wedge \Delta(n+1) \wedge \Delta(n+2) \wedge P(n) \wedge P(n+1))$;
$k := 1$;
while true do
    (* INV: $\Delta'(0) \wedge \ldots \wedge \Delta'(k) \models_{\mathcal{IL}} P(0) \wedge \ldots \wedge P(k)$ ***)***
    (* INV: $\left\{ \begin{array}{c} \Delta'(n) \wedge \ldots \wedge \Delta'(n+k+1) \wedge \\ P(n) \wedge \ldots \wedge P(n+k) \end{array} \right. \not\models_{\mathcal{IL}} P(n+k+1) \Big\}$ ***)***
    $k := k + 1$;
    (* Base case / BMC *)
    $assert_{base}(\Delta'(k))$;
    $assert_{checker}(\Delta(k))$;
    while $\neg entailed_{base}(P(k))$ do
        if $sat_{checker}(cex_{base}(P(k)))$ then return $cex_{base}(P(k))$
        else $\Delta' := refine(\Delta', core(cex_{base}(P(k))))$;
    done
    (* Step case *)
    $assert_{step}(\Delta'(n+k+1) \wedge P(n+k))$;
    $assert_{checker}(\Delta(n+k+1) \wedge P(n+k))$;
    while true do
        if $entailed_{step}(P(n+k+1))$ then return Valid
        else if $sat_{checker}(cex_{step}(P(n+k+1)))$ then break;
        else $\Delta' := refine(\Delta', core(cex_{step}(P(n+k+1))))$;
    done
done

Figure 5.4: Base k-induction algorithm with abstraction. Here Δ' indicates the current abstraction of the fully defined Δ. When an $entailed(P(k))$ query fails, $cex_{base}(P(k))$ is the assignment returned by solver *base* modeling the invalidity (similarly for *checker* and *step*). If $sat_{checker}(cex_{base}(P(k)))$ fails, then $core(cex_{base}(P(k)))$ is a (smallish) subset of $cex_{base}(P(k))$ such that $entailed_{checker}(core(cex_{base}(P(k))) \rightarrow \neg P(k))$ holds. $refine(\Delta', core)$ is the abstraction refinement strategy, possibly based on the *core*, that returns Δ' with additional definitions. $refine(\Delta, core)$ always returns Δ. The specific strategies are detailed in the text.

the solver. Second, it assumes the use of a third instance of the SMT solver, *checker*, that uses un-abstracted Δ equations. This *checker* solver instance is what we use to determine if a counterexample π is spurious or not – $\Gamma_{checker}$ contains the concrete (un-abstracted) $\Delta(0) \wedge \ldots \wedge \Delta(k)$ and $\Delta(n) \wedge \ldots \wedge \Delta(n+k+1)$, defined through k. If π is satisfiable in the context of $\Gamma_{checker}$ then π is an actual counterexample for N, as opposed to a counterexample only for the abstraction Δ'. For each test (4.1') and (4.2'), the algorithm performs a refinement loop that lasts until the checker agrees with a proposed counterexample assignment.

In the pseudocode, $assert_{base}(\phi)$ and $entailed_{base}(\phi)$ are as in the basic algorithm (Section 4.3), similarly for *step*, though these as well as $cex_{base}()$ and $cex_{step}()$ are with respect to Δ'. $assert_{checker}(\phi)$ and the satisfiability test $sat_{checker}(cex_{base}())$ are with respect to the un-abstracted Δ. If $sat_{checker}(cex_{base}())$ holds, then the *base* (or *step*) counterexample for N' is also a counterexample for N. Otherwise the counterexample is spurious. In the latter case, we can retrieve a subset of the values of $cex_{base|step}$ from the checker, called $core_{checker}()$. $refine(\Delta', core())$ then refines the abstraction based on $core()$ and some strategy, and returns the updated Δ'. The refinement stops when $\Delta' = \Delta$, at which point the algorithm reduces to the basic k-induction procedure.

Progress in this algorithm is measured by a strengthening of the constraints, either through refinement of Δ' or an increase in k. It is generally necessary to do both. Increasing k with a too-abstract Δ' will often make the procedure diverge: Δ' often remains too weak for the property to ever be proven invariant, and can produce

an infinite sequence of spurious counterexamples. On the other hand, refining Δ' too soon can eliminate the benefits of abstraction. The algorithm therefore only performs refinements as necessary to ensure no counterexamples are spurious, here determined by the relatively fast entailment check of counterexamples against Δ.

5.3.3 Combining Abstractions and Path Compression

If is also possible to combine path compression and abstraction, however, these two enhancements are not completely orthogonal. If we simply utilize abstracted forms of equations (5.1) and (5.2) based on the current Δ': (5.1') and (5.2'), with $C_{n,k}$ containing only those state variables that are defined in N', it is possible to produce unsound results.

A simple example presents itself in a problem with just two state variables: x and y. Suppose N has a counterexample where

$$\pi = \left\{ \begin{array}{llll} x = & 1, & 2, & 3, \ldots \\ y = & 1, & 1, & 2, \ldots \\ \vdots & \vdots & \vdots & \vdots \end{array} \right\}$$

with the property holding for the first two tuples, but not the third. If we had a N' that had the state variable y defined, but not x, then the check (5.2') would hold at $k = 2$, causing the algorithm to halt prematurely with a "invariant" result, in this case unsound.

One solution to this problem (utilized in the experimental results in Section 6.3) is to initially utilize only the abstracted state variables in $C_{n,k}$, but to re-check any valid result by adding definitions for all state variables still abstracted (thereby

expanding $C_{n,k}$), and re-perform the appropriate test. The problem arises because $C_{n,k}$ may be too weak, so we strengthen it by including the additional terms. If the (induction step) test no longer holds, it will lead to a spurious counterexample, which can be resolved normally.

Another method is to simply utilize the fully-defined $C_{n,k}$, based on Δ, from the beginning.

5.3.4 Implementation: Refinement Strategies

As mentioned above, we base our refinements on an *unsatisfiable core* generated by the failed counterexample check. When we have a formula A denoting a spurious counterexample π, it is a counterexample for Δ' but not Δ. If $\Delta(0) \wedge \ldots \wedge \Delta(k) \wedge A$ is unsatisfiable, then the unsatisfiable core u is an unsatisfiable subset of $\Delta(0) \wedge \ldots \wedge \Delta(k) \wedge A$, ideally a minimal subset. Similarly an unsatisfiable core can be derived from an unsatisfiable $\Delta(n) \wedge \ldots \wedge \Delta(n+k+1) \wedge A$. Unless otherwise noted, we pick stream variables from those that appear in u on which to base our refinement. We do not wish to refine the entire unsatisfiable set of $\Delta(0) \wedge \ldots \wedge \Delta(k) \wedge A$, as this is too undirected; instead we utilize the (smaller) unsatisfiable core. Modern SMT engines can return such a set.

Also, unless otherwise noted, we look at a whole stream such as z as a candidate for refinement, rather than a particular point on the stream, such as $z(5)$.

While performing experiments with this abstraction paradigm, we tried a number of different methods of refinement. Some strategies only refine a single variable

at a time, while others refine multiple variables in a single *refinement step*.

The refinement methods can all be seen as search strategies. The graph we are searching is a *dependency graph* of program variable definitions, with nodes being labeled by the variables in the program, and edges indicating a direct dependency between two variables. A node n_c is the child of a parent n_p if n_c's variable occurs in the right hand side of the definition of n_p's variable. Each node is further marked as being refined, undefined, or fully refined. Refined and undefined correspond directly to the status of the node's variable. *Fully refined* indicates that either this node represents an input variable (and so cannot be refined) or it is refined and all of its descendents are fully refined.

While $\Delta' \neq \Delta$, each refinement step consists of one or more *refinement instances*. Each instance adds the definition of exactly one variable to Δ'. In general we maintain a set of nodes that are candidates for refinement in a priority queue, reinitialized for each refinement step. This queue is initialized with the nodes corresponding to variables found in the unsatisfiable core[2]. In each refinement instance, we remove the first (highest priority) node n_h in the queue. If n_h is undefined, we refine n_h's variable, adding its definition to Δ', mark n_h as refined, and stop this instance. If n_h is an input or n_h is refined and its children are fully refined, we mark n_h as fully refined and choose the next node. Otherwise we add its children to the queue and choose the next node. The overall refinement process stops once all nodes are fully

[2]In order to ensure each refinement instance adds one definition, we also include the graph source nodes (the property variables) with the lowest possible priority in the frontier initialization.

refined, as then $\Delta' = \Delta$.

The specific strategies can be differentiated by how they assign node priorities and the number of instances in each step.

5.3.4.1 Basic Refinement

This refinement is probably the most basic investigated, a simple depth-first search. Priority in the queue is the depth of a node (number of edges taken from the root), and each refinement step consists of a single instance.

Ideally this would result in a model that retained a large amount of abstraction, and so smaller state spaces. This strategy proved to not scale well with the number of variables in the problem, however, as these problems often require many refinement steps, and so expensive re-evaluations of (5.1') and (5.2').

5.3.4.2 Breadth-First

Basic refinement uses depth-first search. To use breadth-first search, we instead give the highest priority to the lowest depth nodes. This suffers the same basic problem of basic refinement in that not enough variables are refined in each step.

5.3.4.3 Core Number Refinement

Since basic refinement does not refine enough variables in each refinement step, we tried a number of variations. Core number refinement essentially calls basic refinement multiple times. Priority is still based on node depth, but we use the same number of instances in each step as there are variables in the unsatisfiable core. This

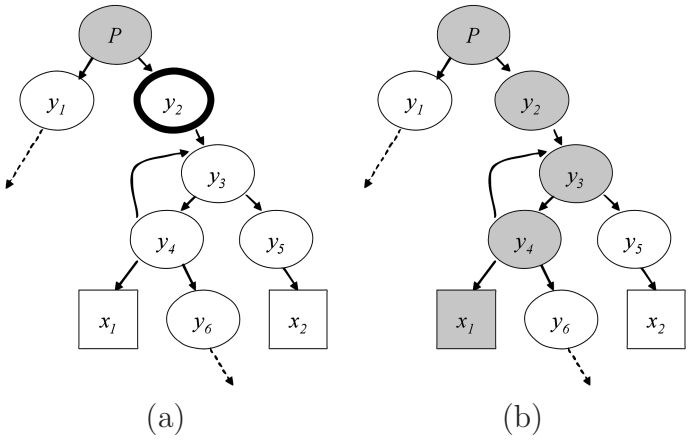

Figure 5.5: Path refinement example. For abstracted non-input variables y_i, input variables x_i, and (refined) property variable P, we choose some unrefined variable y_2 in (a). Path refinement adds definitions for abstracted variables in a depth-first traversal from y_2 through y_3 and y_4, until it reaches input variable x_1 in (b).

offers an improvement over basic refinement, especially in large problems.

5.3.4.4 Core Refinement

This version attempted to more closely tie the refinements to the core itself, refining the variables in the core or their closest undefined descendants. The number of instances equals the size of the core, but this time the core variables are re-assigned highest priorities after each instance. Again, this offers an improvement over basic refinement, although in the case of larger problems, it may still not refine a sufficient number of variables in each refinement step.

5.3.4.5 Path Refinment

One observation with the basic refinement strategy is that when a variable x in a spurious counterexample is refined, often the next iteration also results in a spurious

counterexample, but with a variable y in the unsatisfiable core that is a descendent of x in the dependency graph. This behavior is often repeated until the refinement reaches an input variable, meaning that the elimination of spurious counterexamples results in a path of refinements through the dependency tree from x to the input variable. See Figure 5.5.

Path refinement attempts to shortcut this process by taking an unsatisfiable core variable and defining it and its undefined children recursively in a depth-first fashion, until an output variable is reached. This proved to be one of the best overall refinement strategies explored. Path refinement prioritizes based on node depth, but includes as many instances as necessary to reach a node labeled by an input variable.

This strategy does prove to be inferior to some others in cases where there are large numbers of variables with large numbers of subterms (higher branching factor in the dependency graph), as it can still require a large number of iterations to completely eliminate spurious counterexamples.

5.3.4.6 Subtree Refinement

Since path refinement seems to work well, it seemed reasonable to try expanding *all* sub-terms of a chosen variable instead of just one. Priority is again based on node depth, but iterations continue until the we reach a sibling of the first node to be refined in this step.

While this helped in cases where there were a large number of variables and (nearly) all needed to be quickly refined, it tended to refine too quickly in other

problems, fully refining the abstraction in short order.

5.3.4.7 Heuristic Path Refinement

Initial attempts at a heuristic path refinement took a cue from SAT solver methods, specifically chaff's VSIDS[63] predicate ordering method, and attempted to prioritize nodes based on their presence in unsatisfiable cores earlier in the search, so instead of a simple depth-first search, a greedy search was performed. Variations giving priority scores based on the number of times a variable occurred in unsatisfiable cores (both high and low priority) were attempted, but initial tests did not provide significant overall improvement to the basic path refinement, and the added algorithmic complexity did not seem warranted.

5.3.4.8 Fine-grained Temporal Refinement

The above refinement strategies do not take the temporal situation into consideration. When a variable is defined, it is considered defined for all future k instants, as well as retroactively to previous k instants. Fine-grained refinement breaks with this approach and instead only defined variables for those *specific* time instances mentioned in the unsatisfiable core. It does not perform any particular search and does not retain a priority queue. The intent of this would be to retain maximal abstraction in Δ'. Unfortunately the overhead of this minimal refinement made this even less useful than basic refinement.

5.3.4.9 Hybrid Temporal Refinement

This combines basic and fine-grained strategies. In hybrid temporal refinement, once a variable is refined, it is refined from that point forward, but not retroactively. The reasoning is that if a conflict depends on v_i then it is likely in the next step a conflict will also depend on v_{i+1}, but since we did not have a conflict in the previous step, it may not be necessary to define v_{i-1}.

So once a variable is marked to be defined (as per fine-grained temporal refinement), all versions of later instants are defined as well. Previous instant versions are only refined as the unsatisfiable core demands. The additional bookkeeping for this added to the complexity, while performance did not significantly improve over the simpler basic strategy.

5.4 Summary

This chapter discussed two significant modifications to our basic k-induction algorithm, path compression and structural abstraction. These can offer significant improvements over the basic algorithm: we have shown abstraction to be effective in the case of invalid problems, and path compression reduces incompleteness for valid problems. In addition we have proved several properties of these algorithms.

CHAPTER 6
IMPLEMENTATION AND EXPERIMENTS

6.1 Introduction

In this chapter, we discuss the Lustre verifier we developed and compare it with several other state of the art systems.

6.2 The Kind System

We implemented and evaluated the ideas presented in the previous two chapters in a system called Kind, written in Objective Caml[53]. Kind parses Lustre programs and translates them into formulas, as described in Section 3.4, then sends these formulas to several instances of an SMT solver. It then performs a series of assertions and queries as described in Chapter 4.

Kind currently supports the Yices[35] and CVC3[5] SMT solvers. These both natively support the majority of functionalities that our algorithms assume, including being on-line, incremental, and backtrackable, and being able to return models. Yices is also able to natively compute unsatisfiable cores, while this must currently be approximated in CVC3.

The implementation does contain a few differences with the presented algorithms. The most notable difference is that Kind utilizes translations that are not reduced to a memory depth of at most one. Instead it allows for arbitrary memory depths in a node. The primary impact of this is that we generate a slightly more compact translation (with fewer variables) and the initialization process of the algo-

rithm may start with a k value greater than 1. This does mean Kind will have a minimum depth (possibly greater than 1) at which it can find a counterexample or prove a property invariant.

Another difference is that Kind internally utilizes indices with non-positive values, meaning the initial state will always have a $-k$ index and the last unrolled state in the base case will have index 0. Negative indices were implemented to test more efficiently the cone of influence variations mentioned in Section 4.4.2.2. This does not significantly alter the workings of the algorithms used.

As mentioned in Section 3.4, Kind tests an idealized version of Lustre; several standard features are not supported. Among these features not supported are forward references in node calls (calling a node before its block definition appears), user-defined types, the at-most-one operator (#), and arrays. Many programs containing these unsupported features can be rewritten into functionally equivalent programs that Kind does support. Support for records is currently minimal. Also, full support for when has not yet been implemented.

Kind can be divided into three main subsystems: the parser, the formula generator, and the induction loop control.

The parser translates an input program from Lustre into an intermediate format and stores it in an abstract syntax tree (AST). The abstract syntax tree is then sent to the formula generator, which translates the AST into a set of variable definition macros that will be used to define Δ. Dependencies among variables are then calculated, and these are used to perform the static slicing analysis (Section 4.4.2.1)

and inlining. Definitions that are sliced away are ignored, the rest are passed on to the induction loop control. Most formulas are computed statically and stored as macros, the main exception being the path compression constraint in the case of abstraction.

Kind then follows the algorithms in Figures 4.1, 5.2, and 5.4 quite closely, including a combination utilizing both path compression and abstraction. Definitions are instantiated from the macros as they are asserted.

Combinations of path compression and abstraction may be done soundly in two ways in Kind: by computing a path compression constraint $C(n, k)$ statically, including any currently-undefined state variables, or by computing $C(n, k)$ dynamically, using only the currently defined state variables. In the latter case, if the algorithm returns a Valid result while some state variables remain undefined, it is necessary to include an additional check to avoid unsoundness: if either the termination test (5.2) or the augmented step test (5.1) result in a positive result, we immediately refine all stream variables with memory (those that would appear in the unabstracted version of $C_{n,k}$) and repeat that test.

It is important to note that the algorithms presented in Figures 4.1, 5.2, and 5.4 (and implemented in Kind) are incremental. If k is incremented by larger values than 1, we may break the assumptions inherent in those algorithms, and so it is necessary to include in the process any definition assertions and account for any tests that might have been skipped as a result of a larger k increase. Failure to do so can result in unsoundness.

6.3 Experimental Results

To evaluate Kind against the state of the art in the automated verification of Lustre programs, we compared it with all the publicly-available Lustre verifiers we were aware of. Experiments were run with our Kind solver integrating Yices [35], the Lesar tool provided in the Lustre 4 distribution [49, 65], Rantanplan [40, 41], Luke [22], and SAL 3.0 [33, 72]. We tried also tried Nbac [54], but we were unable to run its latest version on our system. So we used instead the version provided with the Lustre distribution. Unfortunately, this version appears to be unsound as it claimed to have proved a number of properties for which at least two of the other systems were able to find a counterexample. For this reason, we will not include Nbac's results below.

As SAL does not natively accept Lustre programs for verification, we used a state of the art translation provided by Michael Whalen at Rockwell Collins (similar to [83]) in order to compare the systems.

We ran the experiments on a dedicated set of four 3.0 GHz Intel Pentium 4 machines, each with 1 Gb of physical memory, under RedHat Enterprise Linux 4.0.

To increase the number of test problems, we introduced a number of arbitrary modifications in existing problems, simulating likely user errors that would not be caught by a standard compiler: incrementing, decrementing, and negating arithmetic expressions, and switching conjunctions and disjunctions in Boolean expressions. Between one and three such errors were introduced, the script used picking a random character position (ignoring comments) and then applying the error to the next ap-

Table 6.1: Kind: abstraction vs. non-abstraction, invalid problems

Problem Class	**Size**	**Depth**	**Kind bmc**	**Kind bmc abs**
Large Sim.	35	23	1,300.98	1,254.00
Memory I	140	4	6.46	22.92
Memory II	112	5	2.84	7.96
Misc	49	11	0.70	1.89
Protocol	17	12	1.53	4.51
Simulation	94	43	1,708.71	476.66
Totals	**447**		**3,021.22**	**1,767.94**

Note: This table shows runtimes on invalid problems, checked by Kind in BMC mode, respectively without and with abstraction. Runtimes are in seconds. **Size** is the number of problems in each class. In the first table, for each class **Depth** is the maximum unrolling (k) needed to generate a (minimal) counterexample for a problem in this class.

propriate expression in the source.

We grouped together all variants of the same original problem generated this way and removed any *duplicates* from the group, considering two variants duplicates if they produced similar results (same valid / invalid answer and approximately the same runtimes) for all systems and configurations tried. This was done to prevent biasing the results against or in favor of any one system.

The final test set had 1047 problems, each with a single property, classifiable into 6 groups: two groups with problems about memory controllers, one involving protocols, one based on simulations (often involving vehicles), one based on larger simulations, with several hundred lines of Lustre code per problem, and one with mostly toy problems involving counters. Several problems in the large simulations group contain real number values, which many of the other tools do not support.

We partitioned the problems into: (i) a set of 447 *invalid problems*, problems

Table 6.2: Kind: abstraction vs. non-abstraction, valid problems

Problem Class	**Size**	**Depth**	**Kind ind**	**Kind ind abs**
Large Sim.	34	3	8.17	33.49
Memory I	112	3	9.77	24.94
Memory II	65	17	223.26	353.38
Misc	50	11/14	3.66	12.29
Protocol	21	3	1.37	5.07
Simulation	94	12	5.52	25.70
Totals	**376**		**251.75**	**454.87**

Note: This table shows runtimes on valid problems, checked with Kind in (native) induction mode, with iterative deepening, with and without abstraction. Runtimes are in seconds. **Size** is the number of problems in each class. In the first table, for each class **Depth** is the maximum unrolling (k) needed to prove the property for a problem in this class.

Table 6.3: Kind vs. other systems, invalid problems.

Problem Class	**Size**	**Depth**	**Kind bmc abs**	**SAL bmc**
Large Sim.	35	23	1,254.00	2,089.32
Memory I	140	4	22.92	47.07
Memory II	112	5	7.96	28.96
Misc	49	11	1.89	12.07
Protocol	17	12	4.51	7.89
Simulation	94	43	476.66	534.97
Totals	**447**		**1,767.94**	**2,720.28**

Note: This table shows runtimes on invalid problems, checked in BMC mode on Kind and SAL, with Kind using abstraction. Runtimes are in seconds. **Size** is the number of problems in each class. In the first table, for each class **Depth**, is the maximum unrolling (k) Kind needed to generate a minimal counterexample for a problem in this class.

Table 6.4: Kind vs. other systems, valid problems.

Problem Class	**Size**	**Dep.**	**Kind ind abs**	**SAL ind**	**Rantanplan**
Large Sim.	34	3	33.49	1,824.92 (2)	23,401.33 (26*)
Memory I	101	3	24.94	53.26	327.20
Memory II	76	17	353.38	6,327.44 (7)	12,637.82 (14)
Misc	50	14	12.29	920.71 (1)	1,848.84 (2**)
Protocol	21	3	5.07	10.46	3.22
Simulation	94	12	25.70	34.87	1,816.46 (2)
Totals	**376**		**454.87**	**9,171.66 (10)**	**40,034.87**

Note: This table shows runtimes data on valid problems, checked with Kind, SAL, and Rantanplan. Kind used induction mode with iterative deepening (restarts) and SAL used inductive mode with restarts. Runtimes are in seconds. **Size** is the number of problems in each class. In the first table, for each class **Depth**, is the maximum unrolling (k) Kind needed to prove the property for a problem in this class. Note 20 of these problems (marked with *) contain reals, which Rantanplan does not accept. Also Rantanplan returned invalid counterexamples with 2 of these problems (marked with **).

whose property was disproved with a (real) counterexample; (*ii*) a set of 376 *valid problems*, problems whose property was declared proved by at least one system; (*iii*) a set of 224 *unsolved problems*, which will not be considered below. Of the 823 solved problems, 20 valid and 18 invalid problems used real valued streams, the rest used only integer and/or Boolean streams.

The most advanced system in the comparison was sal-inf-bmc, part of the SAL 3.0 toolset [33]. Even if it has an induction mode, strictly speaking, and contrary to Kind, SAL is not a full-blown induction prover. In induction mode, SAL first performs a BMC-like test up to a user specified limit l on the number k of unrollings, and then follows that with a *single* induction test if the BMC test found no counterexamples. For a fairer comparison, we replicated this behavior in Kind as well. To simulate the

checking of problems with an unknown unrolling bound, we ran both SAL and Kind in an iterative deepening fashion on each problem, starting with $l = 0$ and repeatedly restarting the system with a greater value of l until a conclusive answer was returned. As mentioned in Section 4.4.3, induction steps are comparatively expensive and become more so as l grows, so it can be convenient to increase l by more than 1 each time. We eventually chose to increment l by 3 at each restart because it seemed to produce the best results for both systems, SAL and Kind.

The overall results can be seen in Tables 6.1, 6.2, 6.3, and 6.4. Tables 6.1 and 6.2 demonstrate the effects of abstraction on Kind, while Tables 6.3 and 6.4 compares Kind to its main competitors.

Of the two main enhancements to the basic k-induction procedure presented in this paper, we focus on the evaluation of structural abstraction. The other enhancement, path compression, gave increased precision for Kind in induction mode. In particular, it allowed Kind to determine the validity of 8 problems that were not solvable by any of the other systems. Hence, all the results presented in this section are with Kind running with path compression on, unless specified otherwise. The other input options provided by Kind, and not discussed here, were each given the same value across the board.

Note that checking for path compression can be a fairly expensive test, especially on larger problems, and can significantly increase run-times for invalid problems.

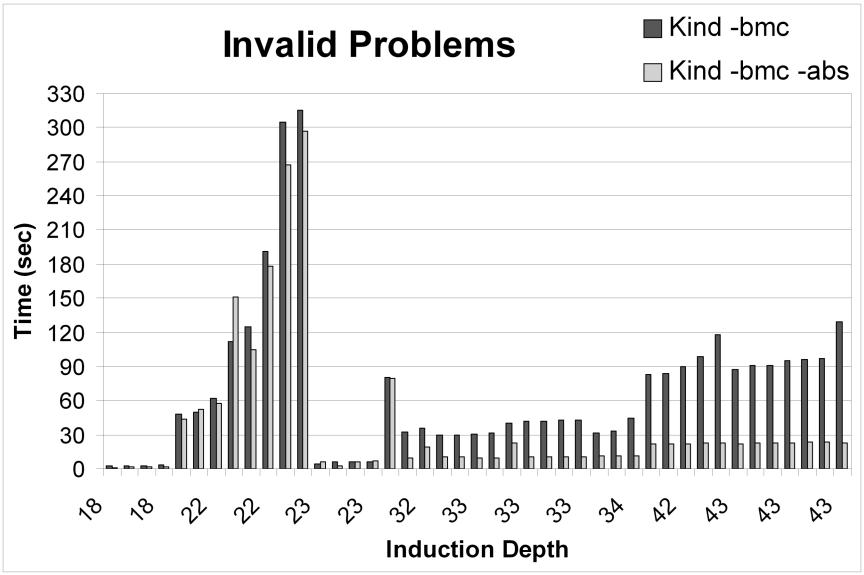

Figure 6.1: Runtimes for Kind with and without abstraction, on hard invalid problems.

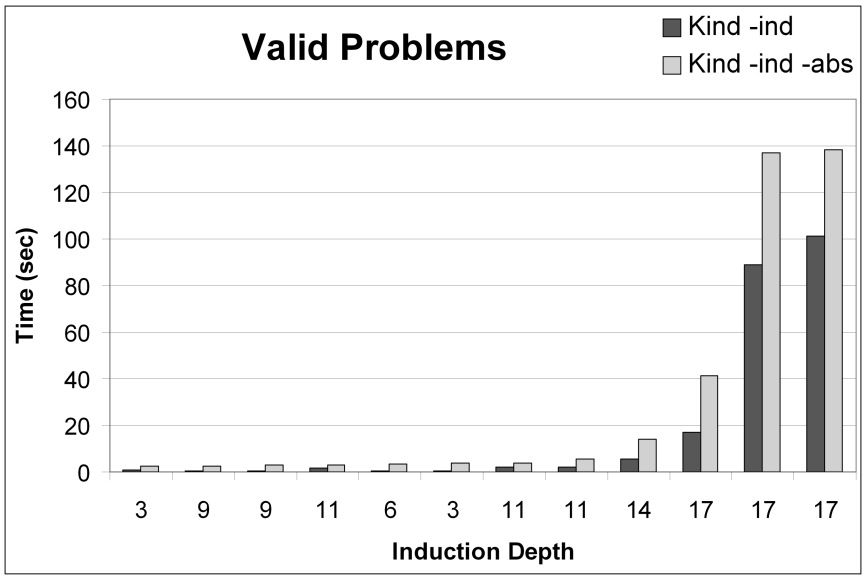

Figure 6.2: Runtimes for Kind with and without abstraction, on hard valid problems.

6.3.1 Abstraction vs. No Abstraction

When verifying safety properties, it is customary to attempt first a quick run of bounded model checking, to see if a counterexample can be found. If that fails, the more expensive full verification check is then performed. Following this practice, it makes sense to look at Kind's results when run in bmc mode on the invalid problems and in interative-deepening inductive more on the valid ones.

Our expectation was that abstraction would not be effective with easy problems, because of its significant overhead. Hence, the purpose of our evaluation was to verify whether abstraction is beneficial with complex problems without producing an unacceptable slowdown with simple ones. Our results partially confirm this thesis.

As a simple complexity measure, let us classify a problem as *easy* if Kind could solve it within 2 seconds *without* abstraction, and *hard* otherwise.

Of the 447 invalid problems, 404 were easy in this sense and were cumulatively solved in 32s. Each of them had a counterexample path of length at most 13. The remaining problems took from 2s to 316s each to solve, for a total of 2989s, with counterexample lengths ranging from 18 to 43. With abstraction, Kind solved each of the easy problems within 2s as well, but took 73s overall, with a slowdown factor of 2.3. However, it solved the hard problems in 1694s with an overall speed up factor of 1.8. Runtimes in seconds for the hard invalid problems are plotted in Figure 6.1. As can be clearly seen, only in one case does abstraction produce a significant slowdown. In most of the other cases it is instead quite effective, with speed-ups in excess of 300% for several problems.

In contrast, abstraction was not effective for the valid problems. To start, almost all of them, 370 over 376, were easy and were solved in a total of 35s without abstraction, with induction depths (the value of k) of 12 or less. The 6 hard problems took from 2s to 101s. The total time for them was 217s, with induction depths ranging from 11 to 17. With abstraction, Kind solved each of the easy problems within 2s except six (solved in less than 4s), with a total time of 115s and a slowdown factor of 3.3. It also solved all of the hard problems, but it was actually 1.6 times slower on them (340s) than Kind without abstraction. A comparison of the behavior with and without abstraction on the hard problems can be seen in Figure 6.2.

This slowdown seems to be primarily due to refinement at larger depths. Generally, detecting a spurious counterexample is comparatively simple, but test (4.2) is generally the most expensive part of the algorithm. Test (4.2') can be nearly as expensive, especially once most variables have been refined. As a rule, the valid problems from this set require nearly all of the variables to be refined in order to prove a property invariant. As a result, the slowdown is not terribly surprising. Recall from Section 5.3 that when we detect a spurious counterexample in test (4.2'), we immediately refine and re-perform test (4.2'). This can nearly double the run-time for each refinement made in the final step (and there may be several).

Given the restarting used in our comparisons, we also encounter this problem at lesser depths that are insufficient to prove a definitive result. In these cases, too, test (4.2') generally results in spurious counterexamples. So the most expensive test is generally run a number of additional times in the abstraction case.

Table 6.5: Performance of various Kind configurations on invalid problems.

Configuration	**Total time**	**Timeouts**
No abstraction	3,021.22	0
Core abstraction	4,027.39	0
Path abstraction	1,769.32	0
Path abstraction, ITE elim.	2,581.74	0
Path abstraction, term. check.	24,061.20	16

Note: This table shows runtimes on invalid problems, checked in BMC mode on Kind. The second line uses the "core refinement" (see Section 5.3.4), the last 3 use path refinement, by itself, with ITE elimination, and with the termination check, respectively.

If there are no restarts, refinement due to failed induction step tests typically emerge at lower k values, somewhat mitigating the effect of the high-k refinement process. Instead, however, we often mostly refine Δ' early in the process. This is a trade-off where we eliminate the benefits of the abstraction in the inductive base checks. This can be seen in the next section.

6.3.2 Other Configurations

It is important to note that we ran Kind with restarts in order to provide a more level playing field with SAL. In actuality, the algorithms covered in previous chapters are more sophisticated, and Kind performs significantly better in native mode.

We tested a number of other configurations with Kind. A small sample can be seen in Tables 6.5 and 6.6, including an alternate abstraction refinement strategy and ITE elimination from Section 4.4, as well as running Kind in native mode without restarts. These tables include a number of configuration on invalid problems (in BMC

Table 6.6: Performance of various Kind configurations on valid problems.

Configuration	**Total time**	**Timeouts**
No abstraction (native)	96.02	0
No abstraction (restart 3)	251.84	0
Core abstraction (native)	366.45	0
Path abstraction (native)	162.32	0
Path abstraction (restart 3)	468.11	0
Path abstraction (skip 3)	128.1	0
Path abstraction, ITE elim. (native)	176.33	0

Note: This table shows runtimes data on valid problems, run with path compression and termination check. *Native* indicates Kind was run in native mode (k increases by 1 each step, no restarts), while *restart 3* is in iterative deepening mode, step size 3, as in the rest of this section. *skip 3* is run without restarts, with k incremented by 3 each step. ITE elimination is used in the last line.

mode) and on valid problems (with path compression and termination checks).

The intuition concerning refinement behaviors seems to be justified when comparing path refinement and core refinement, with the more breadth-oriented core refinement performing more poorly (in fact it performs more poorly than no abstraction at all in both cases, probably due to the added overhead of the refinement combined with the modified search space it presents to the solver). On invalid problems, the termination check negates the benefits of running in BMC mode. The reason for ITE elimination's poor performance is less clear – in theory it should just reduce the search space, but it may be altering it in other ways that affect the performance of the underlying solver.

As mentioned above, abstraction does not seem to be beneficial on valid problems, and produces significant slowdown. This can be seen in Table 6.6. Not surprisingly, incrementing k by more than 1 (without restarting) offers further improvement

over normal native mode or restarting mode.

6.3.3 Kind vs. Other Lustre checkers

Lesar [49, 65] is a state-based verification tool included with the Lustre 4 distribution. It is primarily a BDD-based symbolic model checker, with some limited support for integers through abstraction and a polyhedral library. Due to this, it is incomplete for non-Boolean problems.

Rantanplan [40, 41] could be considered the closest precursor to Kind since it is based on k-induction and SMT techniques. As mentioned in related work, its induction procedure and its SMT support are however less sophisticated, in particular it does not perform abstraction and refinement. Finally, it does not support programs with rational streams.

Luke [22] is another k-induction verifier, inspiring much of the work in Rantanplan, but it is based on propositional logic and was developed mostly for educational purposes. It accepts programs with integers by treating them as bounded integers of a user-specified size.

Since none of these systems have something comparable to Kind's bmc mode, we report the results obtained by Kind when run in iterative-deepening induction mode on both valid and invalid problems. In that mode, Kind solved all of the valid problems but timed out on 8 of the invalid ones.

We ran Lesar from the distribution in its standard configuration (with the -poly argument). Lesar solved correctly fewer than 10% of the valid problems, often

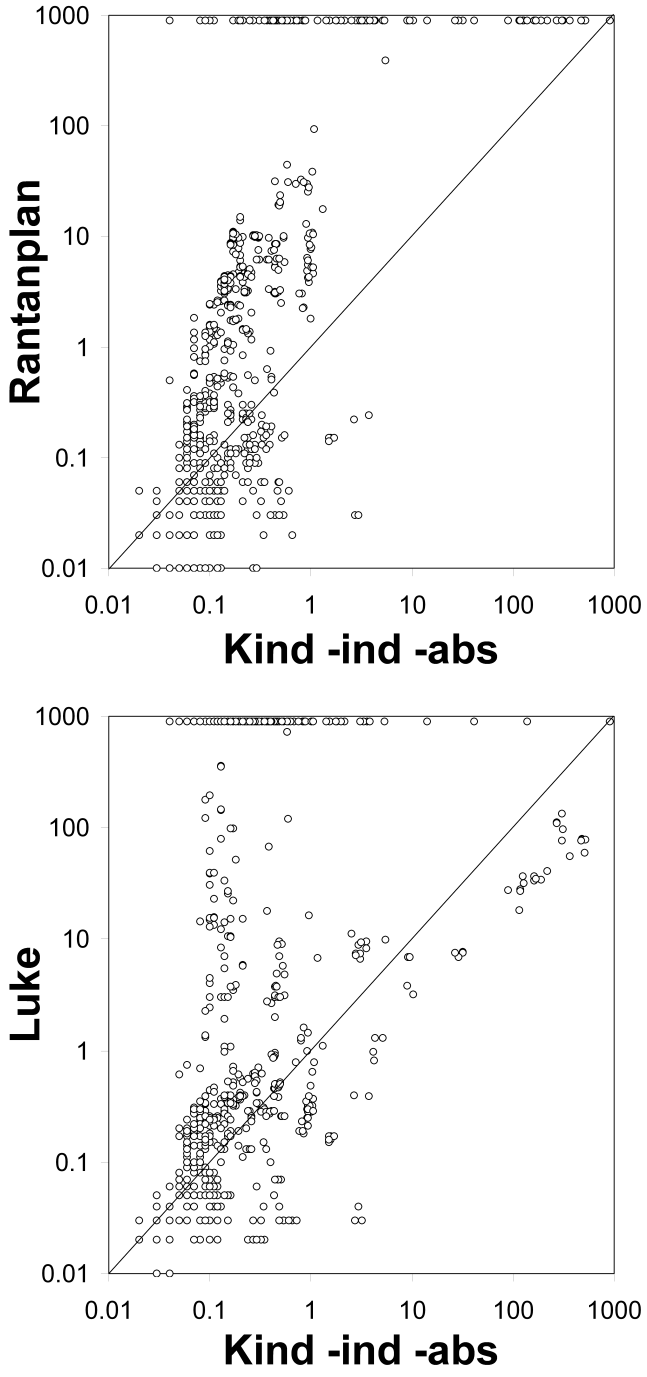

Figure 6.3: Kind vs. Rantanplan and Luke on valid and invalid problems.

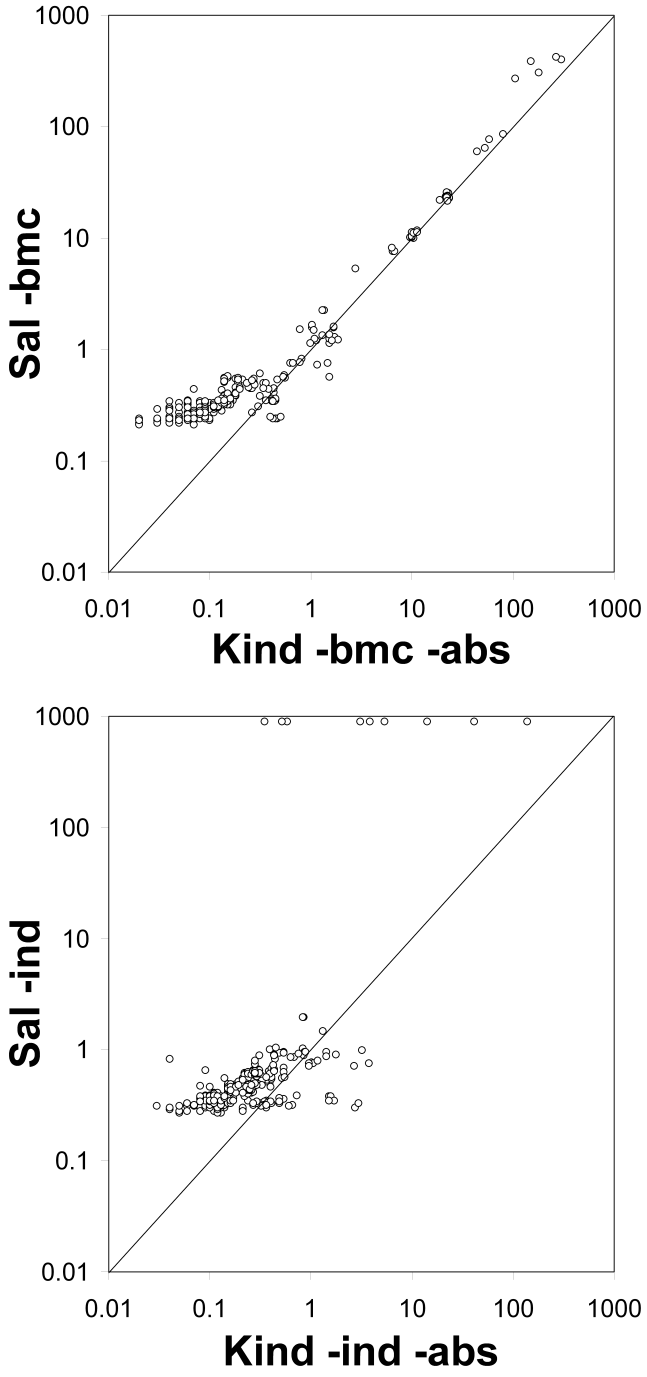

Figure 6.4: Kind vs. SAL, respectively on invalid and valid problems.

quickly producing an incorrect answer for the rest. Since Lesar is incomplete but does not return counterexamples, measuring its performance on the invalid problems is not meaningful.

Rantanplan and Luke were more comparable to Kind. Their performance against our system is summarized in the scatter plots of Figure 6.3, which show runtimes in seconds, on a log-log scale. Timeout points also include problems that a system solved incorrectly due to its incompleteness, or could not solve because they were out of its scope or caused a runtime error (such as stack overflow).

We tried several of the configurations for Rantanplan suggested in [40], with generally similar results, and chose one that seemed one of the best overall performers: deletion filtering with the Pooh checker used in online integration and GLPK as the offline infeasibility checker. Rantanplan solved 84% of the 823 solved problems, (against Kind's 99%) and was overall an order of magnitude slower than Kind on them. Of the 133 problems not solved by Rantanplan, 38 could not be solved due to presence of rational streams, 2 produced incorrect counterexamples, 26 caused run-time errors, and the rest timed out.

Consistently to what observed in [40], Luke performed better than Rantanplan on the invalid problems. However, it solved only 71% of the 823 solved problems (mostly finite-state problems), being overall 30% faster than Kind on those. Of the 242 problems not solved by Luke, 38 could not be solved due to presence of rational streams, 19 produced incorrect counterexamples due to the incompleteness of the bounded integer support, and the rest timed out.

6.3.4 Kind vs. SAL

Even if Kind was consistently and significantly better than the available Lustre-native verifiers, it could be argued that those systems are, for various reasons, not cutting-edge anymore. We looked then for a publicly available state-of-the-art tool that could verify safety properties of both finite- and infinite-state reactive systems. The eventual choice of the SAL toolset [33, 72] was motivated as follows.

The toolset contains a model checker (sal-inf-bmc, hereafter referred to as SAL) in many ways similar to Kind: it can be run in either bmc or induction mode, has a form of path compression, is SMT-based, and uses Yices. The main differences are that it does not use abstraction and is not Lustre-specific. Instead, it uses its own input language, based on the traditional two-state model. This allowed us to verify our hypothesis that adapting existing techniques to work directly on a logical model of Lustre leads to better performance.

To run SAL on Lustre problems we utilized a well-tuned Lustre-to-SAL translation developed at Rockwell Collins along the lines of the translations described in [83]. Rockwell Collins has been developing and continually improving translators from Lustre to various model checkers, including SAL, over several years for the verification of production-level Lustre models of its avionics software.

We ran SAL and Kind in their respective BMC mode on the invalid problems, and in iterative-deepening induction mode on the valid ones. Among SAL's different command-line configurations only its analogous to path compression (the -acyclic option) produced any significant differences in overall performance with respect to the

default configuration. Specifically, enabling path compression was better in induction mode and worse in bmc mode for SAL. The scatter plots of Figure 6.4 summarize SAL's results in those configurations against Kind's results with path compression enabled only in induction mode and abstraction enabled in both modes. As the plots show, the two systems are comparable but Kind's performance dominates.

Both systems solved all the invalid problems, with Kind being more than 50% faster overall. Moreover, Kind solved all valid problems while SAL timed out on 10 of them. On the valid problems solved by both systems, Kind was more than 50% faster. Interestingly, on the latter problems, Kind without abstraction, a configuration more closely comparable to SAL, was actually more than 500% faster.

Overall, these results seems to support our hypothesis that a Lustre-specific k-induction tool offers a performance premium over a translation-based approach. However, it should be noted that, from inspecting its source code, SAL does not appear to exploit the advanced features of Yices as much as Kind does (especially its incremental nature), which might be a significant factor in Kind's better performance.

6.4 Summary

This chapter discussed the Kind system, which implements the ideas presented in earlier chapters. We also compare its performance with several configurations and with that of several other systems, with Kind demonstrating improved performance over the others within our data set.

We have detailed our translation from the Lustre language into the suitable logic $\mathcal{IL}$, and then developed induction techniques including the use of path compression and abstraction / refinement to prove invariant properties. We feel this work offers an advancement in the state of the art of the formal verification of reactive systems, and through experimental results we have demonstrated these techniques offer an improvement over existing ones through our implementation of Kind.

CHAPTER 7
FUTURE WORK

There are a number of areas we would like to examine more thoroughly within this framework.

On a purely implementation level, it is possible to expand Kind to allow for more datatypes, such as bitvectors and bounded integers, as well as support a larger fraction of Lustre, such as all legal uses of temporal operators.

On the more algorithmic side, path compression could stand to be explored in more depth. There are several possible ways to determine configuration equivalence. The method we use, that of comparing state identities is rather naive. In the translation to $\mathcal{IL}$ we preemptively push down all pre terms to only apply to variables, and then compare all variables so marked. In some cases, at least, we could have fewer memory variables involved if we flattened terms and did not push down pres. Additionally there had been some work by Koen Claessen [21] involving a more in-depth analysis of program behavior in order to create smaller configuration comparisons.

There are also several variations on structural abstraction that could prove interesting. The structural abstraction we use can be fairly coarse-grained, depending on the programmer's style. We effectively refine Δ' only by entire definitions, and each definition may be a fairly complicated term involving numerous, possibly otherwise unrelated, control and data structures. Instead we could introduce intermediate variables for some or all terms we encounter, splitting them into their own definitions. Two possibilities of particular interest would be the cases of if-then-else

control structures (and possibly other Boolean expressions), as well as memory terms containing pre, as above. An example of more control-oriented abstraction would be rewriting `x = if b1 then` t_1 `else` t_2, with complex subterms t_1 and t_2, as

$$\texttt{x} = \texttt{if b1 then x1 else x2};$$

$$\texttt{x1} = t_1;$$

$$\texttt{x2} = t_2;$$

We could then use this more fine-grained division of the program to direct refinement more intelligently.

Modular verification is a larger-scale form of abstraction, where one proves a set of properties for a subsection of the overall system, and then uses those properties as a representation of that subsystem. In our case, we would prove properties about a given node, and instead of inlining the subnode as we currently do, we would first instead inline the subnode's properties, which may be sufficient to prove other, global properties of the overall program. Only if these are insufficient would we then add the definitions that make up the node. The expectation is that this will allow us to prove more complicated properties about larger systems.

Finally, another area of potential interest is to extend our techniques to (sometimes) handle *nonlinear arithmetic*, specifically allowing trigonometric functions and multiplication of variables. These types of operations are often used in real-world embedded systems, though they are not directly supported by most SMT solvers. We believe that it would be possible to adequately approximate such expressions using abstraction / refinement techniques. In addition to abstracting stream definitions as

we do now, or nodes with modular verification, we would also abstract these nonlinear operations, providing more and more accurate approximations as we refine the abstraction. The basic idea would be similar to using a Taylor series to approximate the sin function, with each refinement step adding additional terms to the calculation.

REFERENCES

[1] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. SAT-based induction for temporal safety properties. In *Proceedings of the Second International Workshop on Bounded Model Checking, Boston, Mass.*, July 2004.

[2] Mohammad Awedh and Fabio Somenzi. Proving more properties with bounded model checking. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'04), Boston, Mass.*, July 2004.

[3] Mohammad Awedh and Fabio Somenzi. Automatic invariant strengthening to prove properties in bounded model checking. In *Proceedings of the 43rd annual conference on Design automation (DAC '06)*, pages 1073–1076, New York, NY, USA, 2006. ACM.

[4] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 366–378. Springer, 2007.

[5] Clark Barrett. CVC3 webpage, 2006. http://www.cs.nyu.edu/acsys/cvc3/.

[6] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208. AAAI Press/The MIT Press, 1997.

[7] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[8] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In *Advances in Computers*, 58. Academic Press, 2003.

[10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the Design Automation Conference (DAC'99)*, 1999.

[11] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, 1999.

[12] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *In Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, 1999.

[13] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 372–389, 2000.

[14] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *In Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, July 2002.

[15] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Washington, D.C., 1990.

[16] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, 13:401–424, 1994.

[17] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '87)*, pages 178–188, New York, NY, USA, 1987. ACM.

[18] William Chan, Richard J. Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In Michal Young, editor, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 102–112, Clearwater Beach, Florida, USA, March 1998. ACM.

[19] Pankaj Chauhan, Edmund Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, pages 33–51, London, UK, November 2002. Springer-Verlag.

[20] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)*. Springer-Verlag, 2002.

[21] Koen Claessen. Improving temporal induction. Presentation at EqCheck05, Workshop on Equivalence Checking, August 2005.

[22] Koen Claessen. Luke webpage, 2006. http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html.

[23] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Mass, 1999.

[24] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model-checking. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, Venice, Italy, January 2004.

[25] Edmund Clarke, Daniel Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[26] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (Workshop)*, pages 428–437, 1988.

[27] Edmund M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 244–263, 1986.

[28] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, 2000.

[29] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. Sat-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23:1113–1123, July 2004.

[30] CNN. Unmanned European rocket explodes on first flight. *CNN Interactive*, July 1996. http://www.cnn.com/WORLD/9606/04/rocket.explode/.

[31] M. Davis, G. Logemann, and D. Loveland. A machine procedure for theorem-proving. *Communications of the ACM,*, 5:394–392, 1962.

[32] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM,*, 7:201–215, 1960.

[33] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. Tool presentation: SAL 2. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV 2004)*. Springer-Verlag, 2004.

[34] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, 2003.

[35] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Technical report, SRI International, 2006.

[36] N. Eén and N. Sorensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919/2004 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[37] N. Eén and N. Sorensson. Temporal induction by incremental SAT solving. In *Proceedings of the First International Workshop on Bounded Model Checking (BMC'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, 2003.

[38] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM,*, 33:151–178, January 1986.

[39] Martin Franzel and Christian Herde. Efficient proof engines for bounded model checking of hybrid systems. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 04), Linz, Austria*, 2004.

[40] Anders Franzén. Combining SAT solving and integer programming for inductive verification of Lustre programs. Master's thesis, Chalmers University of Technology, 2004.

[41] Anders Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. In *Third International Workshop on Bounded Model Checking (BMC 2005)*, volume 114 of *Electronic Notes in Theoretical Computer Science*, pages 19–33. Elsevier, Jan 2005.

[42] Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06)*, pages 794–801, New York, NY, USA, 2006. ACM.

[43] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *In Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, Boston, Mass, 2004.

[44] Aarti Gupta, Malay Ganai, and Pranav Ashar. Lazy constraints and sat heuristics for proof-based abstraction. In *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID '05)*, pages 183–188, Washington, DC, USA, 2005. IEEE Computer Society.

[45] Aarti Gupta, Malay Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design (ICCAD '03)*, page 416, Washington, DC, USA, November 2003. IEEE Computer Society.

[46] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[47] Nicholas Halbwachs. A synchronous language at work: the story of Lustre. In *Proceedings of the 3rd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2005)*, pages 3–11, July 2005.

[48] Nicholas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions in Software Engineering*, 18(9):785–793, 1992.

[49] Nicholas Halbwachs and Pascal Raymond. A tutorial of Lustre. http://www-verimag.imag.fr/ halbwach/lustre-tutorial.html, 2004.

[50] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[51] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 2002 Symposium on Principles of Programming Languages (POPL 2002)*, pages 58–70, 2002.

[52] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering,*, 23:279–295, May 1997.

[53] INRIA. Caml language webpage, 2001.

[54] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of synchronous programs. *Formal Methods in System Design*, 23:5–37, 2003.

[55] HoonSaang Jin and Fabio Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In *Proceedings of the Second International Workshop on Bounded Model Checking*, Boston, Massachusetts, July 2004.

[56] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameter. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, Jan 2003.

[57] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[58] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. ARIANE 5 flight 501 failure. Report by the inquiry board, European Space Agency, July 1996.

[59] Robin Lloyd. Metric mishap caused loss of NASA orbiter. *CNN Interactive*, July 1999. http://www.cnn.com/TECH/space/9909/30/mars.metric.02/.

[60] Joao P. Marques-Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *In Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[61] Kenneth L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.

[62] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[63] 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 (DAC'01)*, 2001.

[64] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[65] Gordon Pace, Nicolas Halbwachs, and Pascal Raymond. Counter-example generation in symbolic abstract model-checking. *International Journal on Software Tools and Technology Transfer (STTT)*, 5(2):158–164, 2004.

[66] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, New York, 2001.

[67] Pascal Raymond. The Lustre language, 2006. Synchronous programming: principles, languages, implementation lecture notes.

[68] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer-Verlag, Berlin, 2004.

[69] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.

[70] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD '00)*, pages 108–125, London, UK, 2000. Springer-Verlag.

[71] Hossein M. Sheini and Karem A. Sakallah. From propositional satisfiability to satisfiability modulo theories. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 1–9, 2006.

[72] SRI International. Symbolic Analysis Laboratory webpage. http://sal.csl.sri.com/.

[73] Arthur G. Stephenson, Daniel R. Mulville, Frank H. Bauer, Greg A. Dukeman, Peter Norvig, Edward J. Weiler, Lia S. LaPiana, Peter J. Rutledge, David Folta, Robert Sackheim, and Frederick D. Gregory. Mars climate orbiter mishap investigation board phase i report. Technical report, NASA Jet Propulsion Laboratory, November 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[74] Ofer Strichman. Tuning SAT checkers for bounded model-checking. In *Proceedings of the 12th International Conference of Computer Aided Verification (CAV'00)*, 2000.

[75] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Livingston, Scotland, September 2001.

[76] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence, Cosenza, Italy*, 2002.

[77] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symposium on Logic in Computer Science*, 1986.

[78] Eric Vecchié and Robert de Simone. Syntax-driven reachable state space construction of synchronous reactive programs. In *CAV*, pages 213–225, 2005.

[79] Reinhard von Hanxleden. Lecture 14: SCADE/compiling Lustre, 2006. Synchronous Languages class lecture notes.

[80] A. Wahba and D. Borrione. Automatic diagnosis may replace simulation for correcting simple design errors. In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 476–481, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[82] Todd R. Weiss. United axes troubled baggage system at Denver airport. *Computerworld*, June 2005.

[83] Michael Whalen, Darren Cofer, Steven Miller, Bruce Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In *12th International Workshop on Industrial Critical Systems (FMICS 2007)*, Berlin, Germany, July 2007.

[84] Mike Whalen. Autocoding tools interim report. Rockwell Collins internal report, 2004.

[85] Business Wire. Intel adopts upon-request replacement policy on pentium processors with floating point flaw; will take q4 charge against earnings, December 1994. http://findarticles.com/p/articles/mi_m0EIN/is_1994_Dec_20/ai_15939945.

[86] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, July 1997.