Temporal verification of programs

Eric Koskinen
University of Cambridge
Computer Laboratory
Jesus College

November 2012

This dissertation is submitted for
the degree of Doctor of Philosophy
Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60000 words, including tables and footnotes.
Modern software systems are ubiquitous and complex, often running on critical platforms ranging from human implant devices to nuclear power plant controls. How can we be sure that these programs will behave as intended? Programmers must first specify the correct behavior of their programs, and one widely accepted specification language is temporal logic. Despite decades of research in finite-state contexts and some adaptations of finite-state algorithms, we still lack scalable tools for proving temporal logic properties of software.

This dissertation presents scalable techniques for verifying temporal logic properties of transition systems that are described as programs. We exploit the structure of the property and program in such a way that modern abstraction techniques (interpolation, abstraction refinement, termination argument refinement, etc.) can be applied. We distill temporal property verification into the search for a solution to a safety problem, in tandem with the search for sufficient liveness (termination) arguments. This structure allows us to reduce temporal verification to a program analysis task: we generate an encoding such that when modern analysis tools are applied, they are performing the reasoning necessary for temporal verification.

The reduction pertains to verification of state-based temporal logics and, in particular, the universal fragment of Computation Tree Logic ($\forall$CTL). However, in industry, trace-based properties (e.g. LTL, ITL, PSL, SVA) are commonly used and tend to be a natural specification language. Nevertheless, we overcome the divide, showing how our state-based techniques can be used to prove trace-based (i.e. LTL) properties of programs. We observe that LTL can be approximated by CTL. We then introduce a method that examines the $\forall$CTL counterexamples and, if they are not valid LTL counterexamples, symbolically characterizes problematic nondeterminism with decision predicates. We then symbolically determinize the input program with respect to the decision predicates, introducing prophecy variables to predict the future outcome of these nondeterministic choices. Consequently, we are able to prove LTL properties with an $\forall$CTL verifier where we previously could not.

Together, these techniques enable us to prove useful temporal logic properties of imperative programs (e.g. written in the C programming language). We demonstrate—using code fragments from the PostgreSQL database server, Apache web server, and Windows OS kernel—that our method can yield enormous performance improvements in comparison to known tools, allowing us to automatically prove properties of programs where we could not prove them before.
For August.
I have been fortunate to have studied among a community of talented and devoted researchers. A proper enumeration of acknowledgment is difficult, so I shall mention only those from whom I have had particularly close support.

I am grateful to my official supervisor Michael Gordon who provided feedback on my dissertation and endless logistical support. While at the Computer Laboratory, I was fortunate to have worked closely with Matthew Parkinson and Peter Sewell, who provided ongoing support while I was developing mathematical fluency. My officemates Matko Botinčan, Mike Dodds, and John Wickerson were also often the source of valuable discussions. I would also like to thank Peter Sewell and Andrey Rybalchenko for their close examination of this dissertation. Several people outside the Computer Laboratory deserve special mention. First, I would like to thank Maurice Herlihy for drawing me into academia in the early stages of my career. I value my time spent with him and hope for lifelong collaboration. Second, I thank Tim Harris, Peter O’Hearn and Hongseok Yang for their encouragement as I came to Cambridge. On several occasions Josh Berdine and Nir Piterman helped ensure the mathematical integrity of my work. Finally, it was a pleasure to collaborate with Moshe Vardi.

Most significantly, I owe a debt of gratitude to my primary supervisor, Byron Cook. I began at Cambridge with only a minimal understanding of formal methods. He drew me deeper and deeper into theory, until I had developed mathematical fluency. Byron’s support went beyond what should be expected of a supervisor. His patience with me and belief that I could succeed have been particularly noteworthy. I am pleased to have him as both a close friend and collaborator.

I would like to thank the Gates Cambridge Scholarship program and the Computer Laboratory (esp. Lise Gough).

I would also like to acknowledge support from countless friends and family members, especially my parents and Charmaine Jelbert.
# Contents

<table>
<thead>
<tr>
<th>Nomenclature</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>11</td>
</tr>
<tr>
<td>1.1 Dissertation</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Results</td>
<td>14</td>
</tr>
<tr>
<td>1.3 Contributions and organization</td>
<td>17</td>
</tr>
<tr>
<td><strong>2 Foundations of temporal verification</strong></td>
<td>21</td>
</tr>
<tr>
<td>2.1 Transition systems</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Automatic software verification</td>
<td>25</td>
</tr>
<tr>
<td>2.3 Temporal logic</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Linear Temporal Logic</td>
<td>28</td>
</tr>
<tr>
<td>2.3.2 Existing LTL verification techniques</td>
<td>29</td>
</tr>
<tr>
<td>2.3.3 Computation Tree Logic</td>
<td>30</td>
</tr>
<tr>
<td>2.3.4 Over approximating LTL traces with ∀CTL trees</td>
<td>32</td>
</tr>
<tr>
<td><strong>3 Proof system for ∀CTL</strong></td>
<td>35</td>
</tr>
<tr>
<td>3.1 Example</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Proof system</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Frontiers</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Remaining proof rules</td>
<td>40</td>
</tr>
<tr>
<td>3.5 Example revisited</td>
<td>43</td>
</tr>
</tbody>
</table>
# Safety and liveness

## Preliminaries

## Encoding

## The transition predicate

## Example

## Correctness of the encoding

## Related work

# Symbolic determinization for LTL

## Example

## Characterizing nondeterministic choice

## Symbolic partial determinization

## Trace equivalence

## Proving LTL with ∀CTL and determinization

## Related work

# Linear arithmetic programs

## Simple Programming Language

## SPL verification as a program analysis task

### Specialization of Chapter 4

### Encoding for program analysis tools

### Finite representation of transition relation

## Example

## Rank function synthesis

# Determinization refinement

## Iterated algorithm

## Decision predicate refinement

### Correctness criteria for \textsc{Refine}

### Implementation

### Example

### Progress

## Decision predicate synthesis

## Examples
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>An atomic proposition</td>
<td>28</td>
</tr>
<tr>
<td>$\chi$</td>
<td>Counterexample</td>
<td>79</td>
</tr>
<tr>
<td>$\cong$</td>
<td>Simulation relation</td>
<td>64</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Subformula path constructor</td>
<td>32</td>
</tr>
<tr>
<td>$\eta$</td>
<td>Translation from LTL to CTL</td>
<td>32</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>Counterexample Flowgraph</td>
<td>85</td>
</tr>
<tr>
<td>$I^\Omega$</td>
<td>Initial state of determinized transition system</td>
<td>62</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>Subformula path expression</td>
<td>32</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Operational semantics of SPL</td>
<td>72</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>The natural numbers</td>
<td>22</td>
</tr>
<tr>
<td>$\text{Vars}$</td>
<td>The set of all variables (SPL)</td>
<td>10</td>
</tr>
<tr>
<td>$R$</td>
<td>Program counter constructor</td>
<td>120</td>
</tr>
<tr>
<td>$S$</td>
<td>Program counter constructor</td>
<td>120</td>
</tr>
<tr>
<td>$M^\Omega$</td>
<td>Symbolically determinized transition system</td>
<td>61</td>
</tr>
<tr>
<td>$\not\models$</td>
<td>CTL non-entailment</td>
<td>33</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>Set of all state streams</td>
<td>24</td>
</tr>
<tr>
<td>$\pi$</td>
<td>Infinite sequence (i.e. Stream) of states</td>
<td>23</td>
</tr>
<tr>
<td>$\pi^i$</td>
<td>$i$th tail of a trace</td>
<td>23</td>
</tr>
<tr>
<td>$\pi_0$</td>
<td>Head of a trace</td>
<td>23</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Vector of decision predicates</td>
<td>61</td>
</tr>
<tr>
<td>$S^\Omega$</td>
<td>States of determinized transition system</td>
<td>62</td>
</tr>
<tr>
<td>$L$</td>
<td>Subformula path constructor</td>
<td>32</td>
</tr>
<tr>
<td>$R$</td>
<td>Subformula path constructor</td>
<td>32</td>
</tr>
<tr>
<td>$\text{AF}$</td>
<td>CTL eventuality constructor</td>
<td>13</td>
</tr>
<tr>
<td>$\text{AG}$</td>
<td>CTL global constructor</td>
<td>13</td>
</tr>
<tr>
<td>$\text{elim}$</td>
<td>Counterexample elimination predicate</td>
<td>84</td>
</tr>
<tr>
<td>$\text{fault}$</td>
<td>Failure variable in a guarded transition system</td>
<td>47</td>
</tr>
<tr>
<td>$F$</td>
<td>LTL eventuality constructor</td>
<td>12</td>
</tr>
<tr>
<td>Symbol</td>
<td>Explanation</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>LTL global constructor, page 12</td>
<td></td>
</tr>
<tr>
<td>isTrace</td>
<td>Coinductive predicate for an execution, page 23</td>
<td></td>
</tr>
<tr>
<td>isub</td>
<td>Immediate subformula fixedpoint, page 32</td>
<td></td>
</tr>
<tr>
<td>nd</td>
<td>Control point in a guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>pc</td>
<td>Program counter, page 22</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>Subformula fixedpoint, page 32</td>
<td></td>
</tr>
<tr>
<td>walk</td>
<td>Transition relation upto a frontier, page 38</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>LTL next state constructor, page 28</td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>Assignment (SPL), page 36</td>
<td></td>
</tr>
<tr>
<td>*_b</td>
<td>Nondeterministic boolean value, page 71</td>
<td></td>
</tr>
<tr>
<td>::</td>
<td>Infinite sequence (i.e. Stream) cons, page 23</td>
<td></td>
</tr>
<tr>
<td>enc</td>
<td>Procedure in the encoding, page 73</td>
<td></td>
</tr>
<tr>
<td>hd</td>
<td>Infinite sequence (i.e. Stream) head, page 23</td>
<td></td>
</tr>
<tr>
<td>Θ</td>
<td>Transition predicate of a guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>RΩ</td>
<td>Transition relation of determinized transition system, page 62</td>
<td></td>
</tr>
<tr>
<td>ω</td>
<td>Finite sequence of states, page 80</td>
<td></td>
</tr>
<tr>
<td>Vars'</td>
<td>Primed dual of the set of all variables, page 25</td>
<td></td>
</tr>
<tr>
<td>⊢</td>
<td>Our proof system for CTL, page 36</td>
<td></td>
</tr>
<tr>
<td>⋵</td>
<td>CTL entailment, page 31</td>
<td></td>
</tr>
<tr>
<td>⋵️</td>
<td>LTL entailment, page 28</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>Decision predicate component (supposition), page 61</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Boolean expression (SPL), page 10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Decision predicate component (outcome), page 61</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Configurations of a guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Set of initial states of a transition system, page 9</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Transition system, page 21</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Control points of a guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Program text (SPL), page 45</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Transition relation of a transition system, page 10</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Set of states of a transition system, page 21</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>State, page 38</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>Variables of a guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Frontier, page 38</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Guarded transition system, page 45</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Domain of program counters, page 72</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Runs of a guarded transition system, page 115</td>
<td></td>
</tr>
<tr>
<td>nd</td>
<td>Node in a counterexample flow graph, page 85</td>
<td></td>
</tr>
</tbody>
</table>
Software is ubiquitous and complex. How can we be sure that these programs, running on critical platforms ranging from human implant devices to nuclear power plant controls, will behave as intended?

Researchers have sought answers to this question for decades. In particular, the field of formal verification aims to mathematically prove the correctness of computer programs. Unfortunately, most questions about correctness lead to problems which are undecidable: there are no general solutions which apply to all programs and all properties.

We must therefore focus and ask: what are the important programs that need to be verified? As well as: what are the important properties that must be proved of them? We can then search for sound (and sometimes complete or relatively complete) solutions for those domains. An important class of properties are those that characterize the behavior of a system over time. We call these temporal properties. Some examples include:

- Will my program terminate?
- Does my program always allocate memory before it writes to it?
- When my program acquires a lock does it eventually release it?

Proving these properties in general is undecidable in the context of software systems. Therefore, we must find suitable abstract representations that approximate the behaviors of a program in such a way that a proof that the abstract model satisfies a property implies that the program itself satisfies the same property.

### 1.1 Dissertation

There are several forms of temporal specifications that have arisen in the past few decades. Many such specifications are concerned with describing the behavior of the execution traces of a system. A trace property $\varphi$ is a predicate, built from constructors that specify
the current and future (and sometimes past) behaviors of an execution. Using a simple example of a traffic light controller, some trace properties include:

\[ G \neg (\text{green}(L_{NS}) \land \text{green}(L_{EW})) \] — Indicates that in every execution of the controller, there will never (Globally) be a moment when both the north/south light and the east/west light are both green.

\[ F \text{ green}(L_{NS}) \] — Indicates that in every execution of the controller, that eventually (Finally) the north/south light will become green.

Constructors such as \( G \) and \( F \) can be composed to build elaborate specifications such as \( GFp \) (“always eventually \( p \)”). A trace property \( \varphi \) is said to hold provided that it holds of every execution of the system. Because they are based on individual execution behavior, such trace-based logics are fairly intuitive. These logics are also common in industry. Examples include the logic above which is Linear Temporal Logic or LTL [Pnu77], as well as Interval Temporal Logic [Mos85], Property Specification Language [Acc03], and SVA [VR05].

In this dissertation, we explore techniques for verifying such trace-based properties. In particular, we focus on Linear Temporal Logic [Pnu77]. There are known algorithms for verifying LTL properties. These algorithms, described in Chapter 2, involve reasoning over traces in their entirety by performing a construction that crosses the program with an automaton representation of the property. This approach is conceptually simple. However, as we will discuss later, it leads to unnecessary complexity that impedes efficient implementation.

**Scalable temporal verification.** This dissertation is concerned with developing scalable temporal logic verification methodologies that are specifically geared toward systems described as programs. The main ideas in this dissertation are centered around exploiting the structure of the particular temporal logic property and finding better ways to incorporate modern abstraction techniques and tools. Our methods decompose the task of verification so that we can focus more directly on what is needed to find a proof or counterexample.

We will describe an encoding of the task of verification that allows us to automatically partition the input program’s state space (and thus also partitions on the transition relation) into the components that are relevant to a given subproperty. As we show, methodologies based on state space partitioning, rather than state space enumeration, are often tractable in programs which have infinite state spaces.

A benefit of our approach is that these partitions can be symbolically discovered and described using modern abstraction techniques such as abstraction refinement [BR01, CGJ+00], interpolation [BHJM07, McM06], etc. Specifically, we reduce temporal verification to a program analysis problem: we introduce an encoding which, with the use of
procedures and nondeterminism, enables existing interprocedural program analysis tools to naturally perform the reasoning necessary for proving temporal properties (e.g. backtracking, eventuality checking, counterexamples, etc.). Our encoding is designed so that we apply termination reasoning only when needed (as an eventuality subproperty becomes relevant) and only where needed (in the relevant portion of the program’s transition relation).

The strategies above represent a marked improvement in how to verify temporal properties of programs. Unfortunately, as described thus far, they do not directly apply to trace-based logics such as LTL. Instead the strategies apply to state-based logics, which are defined in terms of the possible successors of a given state rather than in terms of an entire trace of the system. An example state-based logic is Computation Tree Logic or CTL [CES86] which has constructors including:

- \( \text{AG} \neg \text{off}(L_{NS}) \) — Indicates that across all possible successors of a state (as well as their successors and so forth), that the north/south light is not turned off.
- \( \text{AF} \neg \text{green}(L_{EW}) \) — Indicates that across all possible successors of a state, that the east/west light is eventually green.

These state-based logics involve predicates that refer to the branching behavior of a system: from a given state, what are its possible successors? They are similar to their trace-based counterparts \( G \) and \( F \). However, they have a slightly different flavour. In a trace-based logic, a single trace has been selected a priori and all reasoning is fixed on that trace, with no branching possible. These two types of logics have some overlap, but neither contains the other. There are properties that can be expressed in one that cannot be expressed in the other and vice-versa. (We will discuss the two logics in more detail in Chapter 2.)

**Reasoning about trace behaviors.** We can exploit an overlap between these two forms of logics. State-based logics can approximate trace-based logics. For example, for a given LTL formula, one can construct a CTL formula (with the same number of temporal operators) such that if the CTL formula holds, then the LTL formula must also hold. This suggests a simple algorithm for verifying LTL properties of transition systems:

1. Take your LTL property and construct an approximation with a CTL formula.
2. Use a CTL proof procedure to show that the CTL formula holds.
3. If so, you are done.

This approach is promising because in most cases the approximation is sufficient to prove the property or discover a valid counterexample (we discuss what happens otherwise
CHAPTER 1. INTRODUCTION

below). Moreover, we can use the effective state-based verification techniques described above.

But what do we do when this approximation fails? That is, we may obtain a CTL counterexample that is not a valid counterexample to the original LTL formula. In these cases, there is a gap between our state-based reasoning methods and the trace-based properties we are trying to prove. The CTL machinery is able to reason about the current state and the subsequent temporal branching behaviors, but not the behavior of an entire trace.

To extend our state-based reasoning to apply to entire program traces (i.e. LTL properties) rather than single states or finite state traces, we need to be able to focus on the future behavior of individual or groups/partitions of traces. We must be able to partition traces based on the nondeterministic choices that the program will make in the future. We achieve this by incorporating an additional strategy: prediction. Prediction-based techniques date back to the prophecy variables of Abadi and Lamport \cite{AL91}. Our work involves ways to understand how choices made in the future impact whether a temporal property holds now.

Moreover, we automate the process of prediction. We automatically discover what kinds of choices will be made in the future, characterize them with decision predicates, and introduce prophecy variables \cite{AL91} that predict the outcomes of these future choices.

1.2 Results

The work of this dissertation has yielded an efficient tool for verifying trace-based temporal logic properties of software. Our work has been able to prove temporal logic (LTL) properties of fragments of C code, extracted from several prevailing software systems including the Apache web server \cite{apa}, the PostgreSQL database server \cite{pos}, and the Windows Operating System kernel. In each case we were frequently able to prove properties that previously could not be proved, and otherwise often able to reduce the proof time by an order of magnitude or more.

Example 1.1. A procedure from the back end of the PostgreSQL database server is given in Figure 1.1. Our technique applies to the original 259-line procedure. For presentation purposes, we have simplified it so that the reader can get the sense for the control-flow. The procedure, named StreamServerPort, iterates over a list of addresses (addr) and performs a listen on each relevant file descriptor. If a listen succeeds, then added is incremented.

Suppose we would like to mathematically prove that if added is incremented (suppose it is initially 0), that eventually the procedure will return STATUS_OK. We can express this
int StreamServerPort(int family, char * hostName, unsigned short portNumber, char * unixSocketDir,
        pgsocket ListenSocket[], int MaxListen){
        ret = getaddrinfo_all(hostName, service, &hint, &addrs);
        if(ret || not addrs)
            return(STATUS_ERROR);
        for (addr = addrs; addr < MAXADDR; addr++) {
            if (not IS_AF_UNIX(family) && IS_AF_UNIX(addr_ai_family)) continue;
            /* See if there is still room to add 1 more socket. */
            for (; listen_index < MaxListen; listen_index++){
                if (ListenSocket_OF/listen_index == -1) break;
            }
            if (listen_index >= MaxListen) break;
            if ((fd = socket(addr_ai_family, SOCK_STREAM, 0)) < 0)
                continue;
            if (not IS_AF_UNIX(addr_ai_family))
                if ((setsockopt(fd, ...) == -1) continue;
            err = bind(fd, addr->ai_addr, addr->ai_addrlen);
            if (err < 0) {
                closesocket(fd);
                continue;
            }
            maxconn = MaxBackends * 2;
            if (maxconn > PG_SOMAXCONN)
                maxconn = PG_SOMAXCONN;
            err = listen(fd, maxconn);
            if (err < 0) {
                closesocket(fd);
                continue;
            }
            ListenSocket_OF/listen_index = fd;
            added++;
        }
        if (not added)
            return(STATUS_ERROR);
        return(STATUS_OK);
    }

Figure 1.1: The body of PostgreSQL procedure StreamServerPort, which can be found in the source of pqcomm.c [pqc]. The method opens a listening port to accept connections. We are interested in proving that when a connection is added that eventually the method will return STATUS_OK.
property in Linear Temporal Logic as the property:

\[ G[\text{added} > 0 \Rightarrow F \text{ STATUS_OK}] \]

We hope that this LTL property holds along every execution path of the program. Proving this, however, is nontrivial. It requires us to reason about a complex reachability question, along with termination requirements.

Previously there were no known techniques for automatically proving that this property holds of this program: existing approaches, such as an adaptation of an algorithm from automata theory [CGP+07], timed out after four hours. With our tool we are able to prove this property in under ten seconds.

This is not a contrived example. In this dissertation we examine over 20 benchmark examples, taken from the systems listed above as well as from the literature. A visual comparison is given in Figure 1.2. Using a logarithmic scale, we compare the times required to prove the property of each benchmark with the previous technique [CGP+07] (on the x-axis) versus our technique (on the y-axis). Our tool is superior whenever a benchmark falls in the bottom-right half of the plot. Timeouts are plotted at 10,000s (seen in the circled area to the right) though they may have run much longer if we had not stopped them.
1.3 Contributions and organization

We now describe the technical contributions and outline for this dissertation. Note that much of this work was completed in collaboration with Dr. Byron Cook. I set out the goal of effective methods of temporal verification and Dr. Cook helped me achieve it. Below I have noted which components of the work I completed myself.

1. We begin by casting a new light on $\forall$CTL verification, introducing a novel proof system in Chapter 3. Unlike the standard formulation of $\forall$CTL, ours is more amenable to reasoning about infinite-state transition systems. Our proof rules are based on *partitioning* the state space rather than *enumerating* the state space. These rules allow us to later incorporate sophisticated abstraction techniques found in modern program analysis tools. We show that our proof system is sound and complete. *This work has appeared in FMSD 2012 [CKV12] and CAV 2011 [CKV11b]. The formulation of the proof system and development in Coq was completed by myself.*

2. In Chapter 4 we characterize the task of $\forall$CTL verification as a *reachability* problem in tandem with a search for sufficient *termination* arguments. Our formulation is such that modern abstraction techniques (abstraction refinement, termination argument refinement, and so forth) can be directly applied. Solving the reachability and termination problem yields a derivation in the proof system presented in Chapter 3. We further show that this technique is sound and relatively complete. *This work has appeared in FMSD 2012 [CKV12] and CAV 2011 [CKV11b]. Per above, this was a collaboration with Dr. Cook. However, the mathematical formulation was entirely done on my own and published initially as a technical report [CKV11a].*

3. In Chapter 5 we present a technique for partial symbolic determinization of a program such that state-based verification strategies can be used to prove trace-based properties expressed, for example, in Linear Temporal Logic (LTL). First, we describe *decision predicates*: predicate pairs that symbolically characterize nondeterministic choices in infinite-state transition systems. In particular, we use them to characterize problematic nondeterministic choices that occur in spurious $\forall$CTL counterexamples used to approximate LTL formulae. We then show how to introduce prophecy variables to predict the future outcomes of these choices. Consequently, state-based verification techniques (such as those described in the above chapters) can be used as a basis for an efficient LTL verification technique. We show that this symbolic determinization preserves trace equivalence. *This work has appeared in POPL 2011 [CK11]. This work was a collaboration with Dr. Byron Cook. Dr. Cook suggested the connection with prophecy variables, but I proved the trace equivalence theorem.*
CHAPTER 1. INTRODUCTION

The above results apply to general transition systems. In the next chapters we specialize our work to an important class: programs written in a simple while-like programming language with integer variables and linear arithmetic expressions.

4. In Chapter 6 we show that our characterization in Chapter 4 can then be specialized, taking an input while program and temporal logic property and generating an encoding of verification as a program analysis task. The encoding uses procedures and nondeterminism to enable off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (e.g. backtracking, reachability, eventuality checking, tree-counterexamples, abstraction refinement, etc.).

This work has appeared in CAV 2011 [CKV11]. This was a 50/50 collaboration with Dr. Byron Cook.

5. Chapter 5 illustrated how LTL verification can be accomplished with an ∀CTL verifier and symbolic determinization. What is missing is an automatic method for discovering sufficient decision predicates. In Chapter 7 we present an iterated determinization refinement technique that examines tree-shaped ∀CTL counterexamples and, when they are spurious to the LTL property, automatically synthesizes decision predicates that characterize the nondeterminism that led to branching in the counterexample.

This work has appeared in POPL 2011 [CK11]. This was a 50/50 collaboration with Dr. Byron Cook.

6. Finally, in Chapter 8 we describe our implementation of the techniques described above, leading us to an efficient tool for verifying LTL properties of programs. We demonstrate that our tool enables us to prove LTL properties of C procedures drawn from the Apache web server [apa], the PostgreSQL database server [pos], and the Windows Operating System kernel. In each case we were frequently able to prove properties that previously could not be proved, and otherwise often able to reduce the proof time by an order of magnitude or more.

This work has appeared in POPL 2011 [CK11] and in CAV 2011 [CKV11]. This implementation and evaluation was a collaboration with Dr. Byron Cook.

Summary. Figure 1.3 depicts a high-level diagram of how the components of this dissertation fit together. The outer component PROVE_LTL is the overall strategy for proving Linear Temporal Logic properties of programs. PROVE_LTL takes the program P and the LTL property ϕ and returns either Succceed or else a counterexample to the property. Internally it operates by approximating the LTL formula as an ∀CTL formula Φ and iteratively applying a state-based prover PROVE_∀CTL. If PROVE_∀CTL succeeds, then the LTL property holds. If not, then we analyze the ∀CTL counterexample with REFINE.
(Chapter 7) to determine if it is a valid LTL counterexample. If not, then there must be some nondeterministic choice that the input program is making which means that this ∀CTL counterexample is spurious with respect to $\varphi$. We then symbolically characterize this nondeterministic choice and introduce a corresponding prophecy variable to predict the outcome of this decision, via the partial determinization routine DETERMINIZE (Chapter 5).

The inner component $\text{PROVE}_{\forall CTL}$ is elaborated in the bottom half of Figure 1.3. This is
the subject of Chapters 3, 4, and 5. PROVE$\nu$CTL takes a program $P$ and $\forall$CTL property $\Phi$ and applies an encoding function $E$ to obtain a new program $P'$. If we can prove that $P'$ is safe (using existing program analysis tools such as SLAM [BR01] or BLAST [BHJM07]) then $\Phi$ must hold of $P$. If not, we examine the safety counterexample and if it consists of a loop we attempt to synthesize a ranking function $f$ that witnesses the termination of that loop. The encoding is then updated to incorporate $f$ and the cycle iterates.
We now set the stage for our work with some technical background. We shall define transition systems, temporal properties, and other mathematical constructs, as well as corresponding notations. We will also give an overview of existing work.

**Proof assistant.** To gain higher confidence in the detail of our work, we have proved many lemmas and theorems using the Coq proof assistant [BC04]. Chapter 2 and Chapter 3 have been developed in Coq. Proofs in Chapter 4 were done by hand (Appendix B) because it did not seem necessary to develop in Coq some of the elaborate but uninteresting details of the formulation. Chapter 5 has been mostly developed in Coq, except Theorem 5.1 which is a modified version of Abadi and Lamport’s result [AL91]. Reimplementing their entire foundational proof in Coq seemed unnecessary and irrelevant to the main ideas of this dissertation. Chapter 6 is a specialization of Chapter 4 and Chapter 7 is mostly implementation details.

Lemmas and Theorems that have been proved in Coq are designated as such with the ■ QED symbol, whereas □ indicates a hand proof. For those that have been proved in Coq, we provide an informal description of the proof, and the complete Coq proof script in Appendix D.

### 2.1 Transition systems

We assume nothing about the set of states $S$, except that state equality is decidable in finite time:

**Axiom 2.1** (Distinguishability). For all $s, s' \in S$, either $s = s'$ or $s \neq s'$ and this can be determined in finite time.

**Definition 2.1** (Transition system). A transition system $M = (S, R, I)$ is a set of states $S$, a transition relation $R \subseteq S \times S$, and a set of initial states $I \subseteq S$. 
Example 2.1. Let $M = (S, R, I)$ where

$$
S \equiv \mathbb{B} \times \{\ell_1, \ell_2, \ell_3, \ell_4\}, \quad \text{denoted } [x]_{pc}
$$

$$
R = \left\{ \left( [true_{\ell_1}], [true_{\ell_2}] \right), \left( [true_{\ell_2}], [true_{\ell_3}] \right), \left( [false_{\ell_3}], [true_{\ell_4}] \right), \left( [true_{\ell_4}], [true_{\ell_4}] \right) \right\},
$$

$$
I = \left\{ [true_{\ell_4}] \right\}.
$$

Visually, we can imagine Example 2.1 as follows:

There are four control points in this transition system: $l_1, l_2, l_3$ and $l_4$. A state of this transition system is a choice for which control point is currently active (we call this component of the state the program counter or “pc”) and whether $x$ is true or false. The first loop iterates some nondeterministic number of times or else infinitely many times. Outside of the first loop $x$ is set to false, then to true again. The final loop iterates infinitely many times.

Example 2.2. Let $M = (S, R, I)$ where

$$
S = \mathbb{B} \times \mathbb{N} \times \{\ell_0, \ell_1\} \quad \text{where } s \in S \text{ is denoted } [x]_{pc},
$$

$$
R = \left\{ \left( [false_{\ell_0}], [true_{\ell_1}] \right) \mid n > 0 \right\} \cup \left\{ \left( [true_{\ell_1}], [true_{\ell_1}] \right) \mid n > 0 \right\} \cup \left\{ \left( [true_{\ell_1}], [false_{\ell_0}] \right) \right\},
$$

$$
I = \left\{ [false_{\ell_0}] \right\}.
$$

In this example, there are infinitely many states. We have used a more symbolic representation of the transition relation. We can also think of this example as a program:

```
while (true) {
    x := true; n := *;  
    while(n>0) { n := n - 1 }
    x := false;
}
```

Here * represents nondeterministic choice, however in this case it represents a nondeterministically chosen natural number.
Streams, coinduction, greatest fixedpoint. In this dissertation we will work with infinite sequences of states (Streams). This simplifies the reasoning as we need not case split between infinite sequences and finite sequences, yet is without a loss of generality. Final states of finite traces can be encoded as states that loop back to themselves in the transition relation. To this end, we require that the transition relation be such that every state \( s \) has at least one successor state:

**Axiom 2.2** (Next state existence). For an input transition system \( M = (S, R, I) \),

\[
\forall s. s \in S \Rightarrow \exists t. t \in S \land (s, t) \in R.
\]

Streams are defined coinductively. **Inductive** definitions are built from the least fixedpoint operator: they begin with a base case and then add structure. By contrast, **coinductive** definitions are built with the greatest fixedpoint operator: they begin with the entire space of possibilities, and rules (may) eliminate members of the space. We will use coinduction throughout this dissertation, with coinductive fixedpoint definitions (greatest fixedpoint) and coinductive induction in our proofs. Our proofs have been verified in Coq (see Chapter 14). A stream can be defined with simply a `\texttt{cons}` operator `::` and no `\texttt{nil}` operator:

**Definition 2.2** (Stream \( X \)). A stream of elements of type \( X \) is the greatest fixedpoint solution to

\[
\begin{align*}
x &: X \\
xs &: \text{Stream } X \\
(x :: xs) &: \text{Stream } X
\end{align*}
\]

**Definition 2.3** (Stream head). For all \( x : X, xs : \text{Stream } X \), \( \text{hd } (x :: xs) \equiv x \).

We define a **trace** to be a stream of states that follows the transition relation:

**Definition 2.4** (Trace). For a transition relation \( R \), we say that \( \pi : \text{Stream } S \) is a trace provided that \( \text{isTrace } R \pi \) holds, defined coinductively:

\[
\begin{align*}
\text{isTrace } R \pi \\
(s, \text{hd } \pi) &\in R \\
\text{isTrace } R (s :: \pi)
\end{align*}
\]

We can conclude that the above predicate holds for the stream \( (s : : \pi) \) by deconstructing it into properties of the head element \( s \) and of the tail element \( \pi \), and the relationship between them.

Throughout this dissertation, when we use the notation \( \pi \), we assume that \( \text{isTrace } R \pi \) holds. When it is clear from the context, we may omit \( R \): \( \text{isTrace } \pi \) or, for a transition system \( M = (S, R, I) \), we lift: \( \text{isTrace } M \pi \equiv \text{isTrace } R \pi \). We use the notation \( \pi_0 \) to mean \( (\text{hd } \pi) \), and the notation \( \pi^i \) to mean the \( i \)th tail of \( \pi \):

\[
\pi^0 \equiv \pi \quad \text{and} \quad (s : : \pi)^{i+1} \equiv \pi^i
\]

Note that tail binds tighter than head, i.e. \( \pi^0 \equiv (\pi^i)_0 \). From the coinductive definition of predicate \( \text{isTrace} \) we can prove an *inductive* property of every trace:
Lemma 2.3. For all $\pi$, $\forall i \geq 0. (\pi_0, \pi_{i+1}) \in R$.

Proof. (Coq: Traces.tr_nth_trans_gen) By induction on $\pi$, unrolling the coinductive definition of $\pi$ at each step.

For Example 2.1 there are infinitely many infinite traces:

Here each oval is a state, where the control location and valuation of $x$ are given. Control begins at $l_1$ with $x=\text{true}$ (i.e. the initial state). It then takes infinitely many steps in the transition system, each time following an arc to a (possibly new) control point and updating $x$ in accordance with the label on the arc, thus reaching a (possibly new) state. The top trace occurs when the first loop does not iterate. After the third step of the trace, it loops forever in the second loop with $x=\text{true}$. There is then one trace corresponding to each natural number of iterations of the first loop—there are infinitely many such traces. Finally, there is a trace which iterates forever in the first loop.

With coinductive reasoning we can show that there exists a trace from every state:

Lemma 2.4 (Trace existence). For all $M = (S, R, I)$ and $\forall s \in S. \exists \pi. \text{isTrace } M \pi \land \pi_0 = s$.

Proof. (Coq: Traces.isRtrace_nsrx) First construct a cofixedpoint, using Axiom 2.2 to witness a next state, denoted $(\text{next } s)$\footnote{To construct this in Coq, we used the sigma operator proj1_sig, which is an existential operator that yields a witness.}, to obtain a trace $\pi$ as:

$$\text{traceFrom } s \equiv s : (\text{traceFrom } (\text{next } s)).$$

Then we can prove the lemma with coinduction by unfolding $\pi$ and showing that isTrace $\pi$ holds because $(s, (\text{next } s)) \in R$.

Definition 2.5 (Traces of $M$). For every $M = (S, R, I)$,

$$\Pi(M) \equiv \{ \pi \mid \text{isTrace } R \pi \land \pi_0 \in I \}.$$
**Predicates and symbolic relations.** For a state predicate \( p \), the meaning \([p]_S\) is defined as the set of concrete states for which \( p \) holds. We may describe a transition relation \( R \) symbolically as \([r]_R\) where \( r \) is a formula over the unprimed variables \( Vars \) and primed variables \( Vars' \equiv \{x' \mid x \in Vars\} \). The relational meaning of a formula \( r \) over primed and unprimed variables, is the set of pairs of states \( s, t \in S \times S \) such that when you take all of the variables of \( s \) together with a primed version of the variables of \( t \), \( (i.e. \) construct \( t' \) by replacing every variable with its primed counterpart) that their values imply \( r \). When it is clear from the context that we mean the real relation as opposed to the symbolic formula representation, we will drop the \([\,\,]_R\) brackets. First and second projections of a relation are defined as

\[
(R)_1 \equiv \{s \mid \exists t. (s, t) \in R\} \quad \text{and} \quad (R)_2 \equiv \{t \mid \exists s. (s, t) \in R\}.
\]

### 2.2 Automatic software verification

Correctness properties of transition systems can be categorized as either safety or liveness properties. We give a formal description of each in this section.

**Definition 2.6 (Safety).** A safety property is a state-predicate \( p \). We say that a transition system \( M \) is \( p \)-safe provided that

\[
\forall \pi \in \Pi(M). \forall i \in \mathbb{N}. \pi_0^i \in [p]_S
\]

Standard safety properties allow you to express many correctness properties of the flavor “nothing bad ever happens,” such as “the size of the buffer is never more than 10.” For Example 2.2 we may want to verify the property \( \neg(\text{pc} = \ell_1 \land x = \text{false}) \) which says that \( x \) can never be \text{false} when the program counter is at \( \ell_1 \). Indeed, this property holds because it holds for every reachable state.

**Temporal safety properties.** More elaborate safety properties involve the ordering of states in the behaviors of a system. For example, if we want to say “a lock is never released before it is acquired,” we need something more than Definition 2.6. We can accomplish this with a monitor automaton \([Lam77]\) such as the following:
Here there are three monitor states: $q_0, q_1,$ and $err$. The monitor begins at $q_0$ and when it observes a lock-acquire state transition (a state transition $s, t$ where the lock is unacquired in $s$ and acquired in $t$), makes a transition to $q_1$, and so forth. If the monitor sees a release state transition but has not seen an acquire state transition, then the automaton transition to the error state $err$ will be taken.

Proving temporal safety properties. There are a variety of recent tools that are exceptionally good at proving (temporal) safety properties [BR01, BHJM07, HJM+02] by answering the reachability question: does there exist an execution of program $P$ that reaches a state $s$ such that property $p$ does not hold. In these tools the monitor automata is conjoined with the input program as follows: the state space of the program is augmented with the current monitor automaton state, and the automaton state changes whenever a transition is taken that matches the arc of the automaton. Once this has been done it becomes a safety problem of showing that it is impossible for the conjoined program to reach a state in which the automaton is at $err$.

One common strategy to prove (or find counterexamples to) safety properties is called counterexample guided abstraction refinement [CGJ+00]. The strategy begins by abstracting away the values of all program variables, and considers only the control-flow graph in order to determine whether assertions hold. If an assertion violation is found then a counterexample will be given in the form of a program path (a path is a sequence of abstract states). This path, however, may be spurious: there may not be an execution of the concrete program corresponding to this abstract path. The tool then examines the spurious counterexample to determine what facts (e.g. predicates) about the state system must be tracked in the next round in order to eliminate the spurious counterexample [BR01].

Liveness. There are many properties that cannot be expressed as temporal safety properties. These properties have components that assert that something must eventually happen. A common example is: if a thread acquires a lock then it eventually releases it.

Definition 2.7 (Liveness). The general form of a liveness property (per [Lam77] who uses uses the notation $p \leadsto q$) is a pair of state-predicates $(p, q)$. We say that transition system $M$ is $(p, q)$-live provided

$$\forall \pi \in \Pi(M). \forall i \in \mathbb{N}. \pi_i^0 \in \llbracket p \rrbracket^S \Rightarrow \exists j \geq i. \pi_j^0 \in \llbracket q \rrbracket^S$$

That is to say that for every execution, if a state is reached in which $p$ holds, eventually a state will be reached such that $q$ holds.
Proving termination. The simplest example of a liveness property is termination, which can be thought of as: \((\text{true, halt})\)-liveness. If termination holds, then from every reachable state eventually a final (“halt”) state will be reached.

Definition 2.8 (Total-order). The structure \((X, \succeq)\) forms a total-order if and only if \(\forall a, b, c \in X\) that

1. \(a \succeq a\) (reflexive),
2. If \(a \preceq b\) and \(a \succeq b\) then \(a = b\) (antisymmetry),
3. If \(a \succeq b\) and \(b \succeq c\) then \(a \succeq c\) (transitivity),
4. \(a \succeq b\) or \(a \preceq b\) (totality).

Note that \(a < b \equiv a \preceq b \land a \neq b\).

Definition 2.9 (Well-ordered set). The structure \((X, \succeq)\) forms a well order if and only if it forms a total order and every nonempty subset of \(X\) has a least element.

Definition 2.10 (Ranking function). For state space \(S\), a ranking function \(f\) is a total map from \(S\) to a well-ordered set.

Definition 2.11 (Well-founded relation). A relation \(R \subseteq S \times S\) is well-founded if and only if there exists a ranking function \(f\) such that \(\forall (s, s') \in R. f(s') < f(s)\).

Theorem 2.5 (Rank function well-foundedness). For all \(R, I\), if \(R\) is well-founded, then from every state \(s \in I\) there are no infinite traces in \(R\).

Automation. While proving that a program terminates is famously undecidable (see reprint \cite{Turing36} of Alan Turing’s 1936 paper) there are known techniques for proving that some (in fact, many) programs terminate \cite{CPR06, PR04a}. These techniques require that we discover a ranking function that witnesses the fact that the program terminates.

In this dissertation we will often work with sets of ranking functions. We denote a finite set of ranking functions (or measures) as \(M\). Note that the existence of a finite set of ranking functions for a relation \(R\) is equivalent to containment of \(R^+\) (the non-reflexive transitive closer of \(R\)) within a finite union of well-founded relations \cite{PR04b}. That is to say that a set of ranking functions \(\{f_1, ..., f_n\}\) can denote the disjunctively well-founded relation \(\{(s, s') | f_1(s') < f_1(s) \lor ... \lor f_n(s') < f_n(s)\}\).

2.3 Temporal logic

While temporal safety and liveness can specify many properties, elaborate behaviors are more easily specified by combining safety with liveness behaviors. Indeed, temporal logic is designed for precisely this purpose.
One form of temporal logic is defined in terms of its traces. Such *trace-based* properties provide a fairly natural specification language and are commonly found in hardware and protocol verification milieux, and described in logics such as LTL [Pnu77], ITL [Mos85], PSL [Acc03] and SVA [VR05]. In particular, we will be interested in proving properties expressed in trace-based logics such as Linear Temporal Logic (LTL). However, in order to obtain an efficient verification strategy, we will show that it is easier to work with *state-based* verification techniques, and then bridge the gap between the two logics.

In this section we define both LTL and the universal fragment of the state-based logic Computation Tree Logic (∀CTL). We then describe an important relationship (approximation) between the two logics. In the following logics, we assume that an atomic proposition \( \alpha \) is from some abstract domain \( D \), and we assume that true, false \( \in D \) and that \( D \) is closed under negation (i.e. \( \forall \alpha \in D. \exists \beta \in D. \llbracket \beta \rrbracket_S = \llbracket \neg \alpha \rrbracket_S \)).

### 2.3.1 Linear Temporal Logic

Here we define Linear Temporal Logic [Pnu77], using the following syntax and semantics:

\[
\varphi ::= \alpha \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid F\varphi \mid \varphi W\varphi
\]

**Definition 2.12 (LTL semantics).**

\[
\pi \models_L \alpha \iff \pi_0 \in [\alpha]^S \\
\pi \models_L \varphi \lor \varphi' \iff \pi \models_L \varphi \lor \pi \models_L \varphi' \\
\pi \models_L \varphi \land \varphi' \iff \pi \models_L \varphi \land \pi \models_L \varphi' \\
\pi \models_L F\varphi \iff \exists i \geq 0. \pi_i \models_L \varphi \\
\pi \models_L \varphi W\varphi' \iff \forall i \geq 0. \pi_i \models_L \varphi \lor \exists j \geq 0. \pi_j \models_L \varphi' \land \forall i < j. \pi_i \models_L \varphi
\]

The LTL semantics is defined on traces: the relation \( \pi \models_L \varphi \) indicates that \( \varphi \) holds for a given trace \( \pi \). The subscript \( L \) will disambiguate \( \models_L \) from another entailment relation defined later. The constructor \( F\varphi \) specifies that along every trace, eventually a suffix will be reached where \( \varphi \) holds. The \( \varphi W\varphi' \) constructor specifies that \( \varphi \) holds forever or \( \varphi \) holds until \( \varphi' \) holds. Note that \( G\varphi \equiv \varphi W false \).

We use \( F \) and \( W \) as our base constructors (as opposed to the more standard \( U \) and \( R \)), as each corresponds to a distinct form of proof: \( F \) to termination and \( W \) to safety. \( U \) and \( R \) can be derived as follows:

\[
\varphi U \varphi' \equiv F\varphi' \land (\varphi W\varphi'), \quad \varphi R \varphi' \equiv \varphi' W(\varphi \land \varphi').
\]

We omit the next state constructor \( X \) as it is not very useful in the context of software and easily subsumed by \( F \).

**Definition 2.13 (LTL Machine Entailment).** For every \( M \),

\[
M \models_L \varphi \iff \forall \pi \in \Pi(M). \pi \models_L \varphi
\]
CHAPTER 2. FOUNDATIONS OF TEMPORAL VERIFICATION

Negation normal form. We will work with LTL properties that have been written in negation normal form where negation only occurs next to atomic propositions (recall that we assume the domain of atomic propositions is closed under negation). A formula that is not in negation normal form can be easily normalized using identities:

\[ \neg(\varphi \land \varphi') = \neg \varphi \lor \neg \varphi' \]
\[ \neg(\varphi \lor \varphi') = \neg \varphi \land \neg \varphi' \]
\[ \neg F \varphi = \neg \varphi \land \text{false} \]
\[ \neg(\varphi W \varphi') = (F(\neg \varphi \land \neg \varphi')) \land (\neg \varphi' \land W (\neg \varphi \land \neg \varphi')) \]

For Example 2.1 we may want to prove that the following LTL property holds:

\[ G (\neg x \Rightarrow F x) \]

This property means that for a given trace that if you take every tail of the trace (“G”) that whenever you are in a state where \( x = \text{false} \), then it must be that you will eventually (“F”) reach a state in which \( x = \text{true} \). These kinds of properties are quite common. Examples include “when you acquire a lock, you must eventually release it” and “when you allocate memory you must eventually free it.” The property indeed holds of this transition system, and is easy to check intuitively by examining the possible traces.

2.3.2 Existing LTL verification techniques

There are known automata-theoretic techniques for proving LTL properties [Var95]. We now describe these algorithms to give the reader a sense for the state-of-the-art.

The traditional automata-theoretic method begins by representing the LTL formula as a Büchi automaton [GO01, SB00]. A Büchi automaton operates over infinite inputs (in this case the traces of the program), and accepts the sequence if and only if there exists a run of the automaton which visits one or more of the final states infinitely often. Checking whether infinite traces visit a final state infinitely often can be done as follows: construct a Büchi automaton for the negated LTL formula. Then show that there is no trace of the transition system that is accepted by this (negated) automaton.

Example 2.3. Here is the negated Büchi automaton for \( \neg [G (\neg x \Rightarrow F x)] \), which can be rewritten to the equivalent property \( F (G \neg x \land \neg x) \) or just \( F (G \neg x) \):

[Diagram of Büchi automaton]
There are two Büchi states in this automaton: \textit{init} and \(q_1\), where \(q_1\) is the only final state. If \(q_1\) can be visited infinitely often by a trace of the transition system then the property does not hold. Let us now consider the traces of the transition system. For the last trace (where \(x\) is always \textit{true}), only the Büchi state \textit{init} is visited. Thus the automaton does not accept this run. In all of the other traces, a state where \(x=\text{false}\) is reached and the transition to \(q_1\) can be taken. However in the subsequent state of the transition system \(x=\text{true}\), so the arc returning to \(q_1\) cannot be taken so \(q_1\) cannot be visited infinitely often. Since there are no traces of the transition system accepted by the automaton for the negated property, the original property holds of the transition system.

This algorithm does not scale. We have described the algorithm in its simplest form. There are details involved in extending it to programs that have more elaborate transition systems \cite{CGP+07}. While this does lead to an algorithm for proving LTL properties of programs, the techniques do not scale in practice. For the Apache benchmark given in Chapter 1, the algorithm times out after four hours. The difficulty with this approach is that the mechanisms that allow us to ignore infinite executions not accepted by the fairness constraints are effectively the same as the expensive techniques used for proving termination. Thus, in practice it relies too heavily on termination proving machinery.

Previous work has also examined different methods of representing systems \cite{AJN+04,BLW05,VSV+05} in order to facilitate proving linear-time temporal properties or proving linear-time properties of abstractions (e.g. pushdown systems \cite{EKS03,SB05}). When model-checking is performed using explicit-state techniques \cite{HP00,Hol97,KNP02} then the converse of our assumption is true: linear-time traces are in fact more naturally explored than branching-time executions in this context.

### 2.3.3 Computation Tree Logic

Our strategy for proving LTL properties is built upon an efficient verification technique for properties in the universal fragment of Computation Tree Logic or \(\forall\text{CTL}\) \cite{CES86}. The syntax and semantics of \(\forall\text{CTL}\) is:

\[
\Phi ::= \alpha \mid \Phi \land \Phi \mid \Phi \lor \Phi \mid AF\Phi \mid A[\Phi W \Phi]
\]
Definition 2.14 (∀CTL semantics). The semantics of an ∀CTL formula is as follows:

\[ R, s \models \alpha \iff s \in [\alpha]^S \]
\[ R, s \models \Phi_1 \land \Phi_2 \iff R, s \models \Phi_1 \land R, s \models \Phi_2 \]
\[ R, s \models \Phi_1 \lor \Phi_2 \iff R, s \models \Phi_1 \lor R, s \models \Phi_2 \]
\[ R, s \models AF\Phi \iff \forall (s_0, s_1, \ldots) \in \Pi(S, R, \{s\}). \exists i \geq 0. R, s_i \models \Phi \]
\[ R, s \models A[\Phi_1 W \Phi_2] \iff \forall (s_0, s_1, \ldots) \in \Pi(S, \ldots, \{s\}) \]
\[ (\exists j \geq 0. R, s_j \models \Phi_2 \land \forall i \in [0, j). R, s_i \models \Phi_1) \]
\[ \lor (\forall i \geq 0. R, s_i \models \Phi_1) . \]

The temporal constructors in ∀CTL are state-based in structure. The constructor AF\Phi specifies that across all computation sequences from the current state, a state in which \( \Phi \) holds must be reached. The A[\Phi_1 W \Phi_2] constructor specifies that \( \Phi_1 \) holds in every state where \( \Phi_2 \) does not yet hold. Again notice that the constructor AG\Phi, which specifies that \( \Phi \) globally holds in all reachable future states, can be derived: A[\Phi W false].

For Example 2.1, the computation tree is as follows:

From the initial state on the far left, there is only one possible successor state (pc = \( l_2 \land x = true \)). From that state there are two successor states depending on which arc is taken, and so on. For a given transition system, its computation tree and complete set of traces are derivable from each other.

Definition 2.15 (∀CTL machine entailment). For every \( M = (S, R, I) \) and ∀CTL property \( \Phi \),

\[ M \models \Phi \iff \forall s \in I. R, s \models \Phi . \]
**Subformulae.** We will need to enumerate ∀CTL subformulae, taking care to uniquely identify each one. To this end, our definition of subformulae maintains a context path:

\[ \kappa \equiv \epsilon | L \kappa | R \kappa \]

that indicates the path from the root \( \epsilon \) (the outermost property \( \Phi \)), to the particular subproperty \( \Phi \) of interest, at each step taking either the left or right subformula (\( L\kappa \) or \( R\kappa \)). The set of subformulae is a set of \((\kappa, \Phi)\) pairs:

**Definition 2.16 (Subformulae).** For an ∀CTL property \( \Phi \),

\[
\begin{align*}
\text{sub}(\Phi) & \equiv \text{sub}(\epsilon, \Phi) \\
\text{sub}(\kappa, \alpha) & \equiv \{(\kappa, \alpha)\} \\
\text{sub}(\kappa, \Phi \lor \Phi') & \equiv \{(\kappa, \Phi \lor \Phi')\} \cup \text{sub}(L\kappa, \Phi) \cup \text{sub}(R\kappa, \Phi') \\
\text{sub}(\kappa, \Phi \land \Phi') & \equiv \{(\kappa, \Phi \land \Phi')\} \cup \text{sub}(L\kappa, \Phi) \cup \text{sub}(R\kappa, \Phi') \\
\text{sub}(\kappa, \text{AF}\Phi) & \equiv \{(\kappa, \text{AF}\Phi)\} \cup \text{sub}(L\kappa, \Phi) \\
\text{sub}(\kappa, \text{A}[\Phi W\Phi']) & \equiv \{(\kappa, \text{A}[\Phi W\Phi'])\} \cup \text{sub}(L\kappa, \Phi) \cup \text{sub}(R\kappa, \Phi')
\end{align*}
\]

For example, say we have the ∀CTL property \( \Phi = \text{AG}(p \Rightarrow \text{AF} q) \). Then,

\[
\text{sub}(\Phi) = \{(\epsilon, \text{AG}(\neg p \lor \text{AF} q)), (\text{L}\epsilon, \neg p \lor \text{AF} q), (\text{R}\epsilon, \neg p, \text{AF} q), (\text{LLR}\epsilon, q)\}.
\]

**Definition 2.17 (Immediate subformulae).** For an ∀CTL property \( \Phi \),

\[\text{isub}(\Phi) \equiv \{\psi \mid (L\epsilon, \psi) \in \text{sub}(\Phi) \lor (R\epsilon, \psi) \in \text{sub}(\Phi)\}\]

### 2.3.4 Over approximating LTL traces with ∀CTL trees

It is known that an LTL formula can be over-approximated with ∀CTL [Mai00]. Properties in ∀CTL describe the behavior across all branching subsequent paths (*i.e.* trees). This is similar to LTL which describes behavior across the set of all traces, each taken in their entirety. For every LTL property there is a sufficient over-approximation in ∀CTL: if an ∀CTL property (consisting of AG and AF subproperties) holds, then an analogous LTL property (where we replace AG with G and AF with F) also holds. We begin by defining a simple syntactic translation from a formula in LTL to its corresponding approximation in ∀CTL:

**Definition 2.18 (Over-approximation \( \eta \)).** An LTL formula \( \varphi \) can be translated into an over-approximation as an ∀CTL formula \( \eta(\varphi) \) as follows:

\[
\begin{align*}
\eta(\alpha) & \equiv \alpha \\
\eta(\varphi \land \varphi') & \equiv \eta(\varphi) \land \eta(\varphi') \\
\eta(\varphi \lor \varphi') & \equiv \eta(\varphi) \lor \eta(\varphi') \\
\eta(F\varphi) & \equiv \text{AF} \eta(\varphi) \\
\eta(\varphi W \varphi') & \equiv \text{A}[\eta(\varphi) W \eta(\varphi')]\]
\]
Lemma 2.6 (Approximating LTL with ∀CTL). For a machine \( M \) and LTL property \( \varphi \),

\[ M \vDash \eta(\varphi) \implies M \vDash \varphi \]

Proof. (Coq: \texttt{Ltl.m_ltl_imp_actl}) By corresponding structural induction on the formulae \( \eta(\varphi) \) and \( \varphi \). We first unlift the LHS (and then use the ∀CTL semantics) and the RHS (and then use the LTL semantics). Since \( \Pi \) is defined over \( I \), we are free to pick a state \( s \in I \) such that \( s \vDash \Phi \). We now pick a trace \( \pi \in \Pi(S,R,\{s\}) \) and the appropriate case from the CTL semantics. For example, in the \( s \vDash \exists x.\Phi \) case the universal quantification tells us that for \( \pi = (s_0,s_1,...) \) there exists an \( n. s_n \vDash \Phi' \). By the inductive hypothesis for all \( \tilde{\pi} \in \Pi(S,R,\{s_n\}) \), we have that \( \tilde{\pi} \vDash \varphi' \), and thus we have established the criteria for \( \pi \vDash \exists x.F.\varphi' \).

For example, consider the LTL property \( G(\neg x \Rightarrow Fx) \). We can replace \( F \) with \( AF \) and \( G \) with \( AG \) to obtain the following CTL property:

\[ AG(\neg x \Rightarrow AFx) \]

This property says that from every initial state, every reachable ("\( AG \)") state \( s \) is such that the implication holds. Specifically, if \( s \) is a state in which \( x = \text{false} \), then every traversal down the entire subtree from \( s \) will eventually ("\( AF \)") reach a state \( t \) in which \( x = \text{true} \). This CTL formula approximates the LTL formula. If we can prove that the CTL formula holds, then we know the LTL formula holds, i.e.:

\[ [AG(\neg x \Rightarrow AFx)] \implies [G(\neg x \Rightarrow Fx)] \]

**Sufficient, but not necessary.** Since we are working with an over-approximation, there are cases where the over-approximation in ∀CTL does not hold, but the original LTL property does hold. That is, \( M \vDash \varphi \) but \( M \not\vDash \eta(\varphi) \).

**Example 2.4.** We return to Example 2.1: Let \( M = (S,R,I) \) where

\[
S \equiv \mathbb{B} \times \{\ell_1, \ell_2, \ell_3, \ell_4\}, \text{ denoted } [s_{pc}]
\]

\[
R = \{([true], [true]), ([true], [true]), ([true], [true]), ([false], [true]), ([false], [true]), ([true], [true]), ([true], [true])\},
\]

\[
I \equiv \{[true]_{\ell_1}\}.
\]

This example, again, can be visualized as:

```
\[\begin{array}{c}
\ell_1 \\
\ell_2 \\
\ell_3 \\
\ell_4
\end{array}\]

\[
\{x = \text{false}\} \quad \xrightarrow{x := \text{false}} \quad \{x = \text{true}\}
\]
Let us say that we want to verify the LTL property $\varphi = F G x$, which, informally, can be read as “for every trace of the system, $x$ will eventually become true and stay true.” For this transition system, the LTL property $\varphi$ holds. This is because each individual trace eventually gets to a point where $x = \text{true}$ forever more. For example, if a trace never leaves the loop at $\ell_2$, then the property is valid because $x = \text{true}$ before entering the loop. For the traces that do reach $\ell_3$, $x = \text{true}$ will hold once $pc = \ell_4$, and then remain true.

Let us now consider what happens if we use $\forall$CTL to approximate LTL by letting $\Phi = \eta(\varphi)$:

$$M \models AF AG x \Rightarrow M \not\models F G x.$$ 

Unfortunately, this $\forall$CTL formula does not hold of the transition system and we obtain the following $\forall$CTL counterexample:

The above $\forall$CTL counterexample is an infinite tree of behaviors that arises from the choice as to when control exits the first loop. The $\forall$CTL property $AF AG x$ does not hold because there is no set of states $S$ that you are guaranteed to eventually reach from the initial state such that for every $t \in S$, all reachable states from $t$ are ones in which $x = \text{true}$. No matter how far you travel down the spine of $\ell_2$ states, there are always more behaviors that will lead to a state in which $x = \text{false}$.

In Chapter 3 and 4 we will present methods for reasoning about CTL properties of program. As we show in Chapter 5 when combined with the above approximation, these methods can be very effective. However, the question is as to how to handle this gap that exists between the logics LTL and $\forall$CTL. In Chapter 5 we will return to this issue and describe a symbolic partial determinization that preserves the relevance of using $\forall$CTL to verify general LTL properties.
In this chapter we present a novel proof system for ∀CTL. Unlike the standard formulation of ∀CTL ours is more amenable to reasoning about infinite-state transition systems. Our proof rules are based on partitioning the state space rather than enumerating the state space. The rules we obtain allow us to more easily incorporate sophisticated abstraction techniques found in modern program analysis tools, as discussed later in this dissertation.

In our treatment of $\text{AF}\Phi$, for example, we will partition the state space by identifying a frontier: a set of states where $\Phi$ will hold. Then, with the help of an inductive predicate parameterized by the frontier, we recast the ∀CTL semantics of AF in terms of the well-foundedness of a relation, rather than the existence of an $i$-th state in every trace. This formulation allows us to more efficiently prove AF properties because we can discover well-founded relations that are over-approximations of the inductive predicate rather than than searching for per-trace ranking functions.

Looking ahead, we will see that the proof system presented in this chapter serves as both the intuition and correctness argument for our verification technique in Chapters 4 and 5. Temporal reasoning in these chapters is restricted to the state-based logic ∀CTL, but we show that the technique we develop can be used in combination with an iterative symbolic determinization procedure described in Chapter 5 to obtain an efficient algorithm for verifying LTL.

### 3.1 Example

Let us say that we want to prove that the ∀CTL property $\Phi = \text{AG}(x \Rightarrow \text{AF} \neg x)$ holds of the transition system $M$ from Example 2.2.
while (true) {
    x := true; n := *;
    while(n>0) { n := n - 1 }
    x := false;
}

This is a common property shape, used in instances such as lock acquire/release: x := true is acting as acquire, x := false is acting as release. In order to prove that $R, I \models \forall C (AF \neg x \lor \neg x)$ we must show that for every $s_0, s_1, s_2, ... \in \Pi(M)$ that $R, s_i \models (AF \neg x \lor \neg x)$ for all $i \geq 0$. The non-deterministic assignment to n means that there is one trace $s_0, s_1, s_2, ...$ for every natural number.

The standard $\forall$CTL semantics would require that we construct an infinite derivation. Clearly, this is undecidable without some form of abstraction. Of course we know intuitively that the property holds because whenever control reaches $\ell_1$ it will eventually reach a state where $n = 0$ and transition to $\ell_0$, reassigning false to x. We now introduce a proof system for $\forall$CTL that will allow us to contruct a finite derivation for this example.

### 3.2 Proof system

We now develop a proof system for $\forall$CTL. Unlike the standard $\forall$CTL semantics, which is a relation between a single state and a formula, ours is a relation between a set of states and a formula. Consequently, we can develop proof rules that partition the state space rather than enumerating the state space. In the case of disjunction, for example, we can partition a set into two sets: one for which the left disjunct holds and another for which the right disjunct holds.

Partitioning is also used to characterize temporal operators. For example, in the case of formula $AF \Phi$, we will define a set of states (a partition) called a frontier. If the property holds, the frontier is the set of states you must reach where the subproperty $\Phi$ holds, when you follow along a path from an initial state.

Over the course of this section, we will discuss each proof rule in turn. The complete proof system is given at the end in Figure 3.6.

**Atomic proposition, conjunction and disjunction.** The proof system is of the form $(R, I) \vdash \Phi$ and relates a transition relation $R$ and a set of (initial) states $I$, with an $\forall$CTL formula $\Phi$. The rule $\text{RAP}$ for an atomic proposition $\alpha$ is given in Figure 3.4 and involves a simple check to see if $I$ is contained within the set of states for which $\alpha$ holds. This is not surprising: it can be thought of as simply lifting the standard $\forall$CTL semantics to sets.

We will also prove soundness and completeness results for each rule in turn. The overall soundness and completeness result is obtained by induction on the structure of $\Phi$: $\text{RAP}$
is the base case and in the inductive step we assume that \( \psi \) holds for all \((\psi, \kappa) \in \text{isub}(\Phi)\).

The proofs have been verified with the Coq proof assistant and are designated as such with the \(\blacklozenge\) QED symbol. The first case is for \(\text{RAP}:\)

**Lemma 3.1** (Soundness and completeness of \(\text{RAP}\)). For all \(\Phi, R, I\),

\[
\langle R, I \rangle \vdash \alpha \iff M \vDash \alpha.
\]

*Proof.* (Coq: \texttt{Cktl.soundness}, \texttt{Cktl.completeness}) Unlift \(\vDash\), then the claim simply amounts to \(I \subseteq [\alpha]^S \iff \forall s \in I. s \in [\alpha]^S\). \(\blacklozenge\)

The conjunction rule (Figure 3.2) requires that both \(\Phi_1\) and \(\Phi_2\) hold of all states in \(I\).

![Figure 3.2: Conjunction](image)

The disjunction rule (Figure 3.3) partitions the states into two sets, one for which \(\Phi_1\) holds and one for which \(\Phi_2\) holds.

![Figure 3.3: Disjunction](image)

These rules can be thought of as simply lifting the standard ∀CTL semantics to sets. The benefit, however, is that we characterize ∀CTL verification as a partitioning, rather than enumeration, of states.

**Lemma 3.2** (Soundness and completeness of \(\text{Rand}\)). For all \(\Phi, R, I\),

\[
\langle R, I \rangle \vdash \Phi_1 \land \Phi_2 \iff M \vDash \Phi_1 \land \Phi_2.
\]

*Proof.* (Coq: \texttt{Cktl.soundness}, \texttt{Cktl.completeness}) Trivial. \(\blacklozenge\)
Lemma 3.3 (Soundness and completeness of ROR). For all $\Phi, R, I,$

\[
\langle R, I \rangle \vdash \Phi_1 \lor \Phi_2 \iff M \models \Phi_1 \lor \Phi_2.
\]

Proof. (Coq: Cktl.soundness, Cktl.completeness) Trivial. □

In the proofs of Lemmas 3.1, 3.2, and 3.3 we needed two important side lemmas. The following lemmas allow us to decompose $I$ into individual states $s$, and compose individual states $s$ into a set $I$.

Lemma 3.4 (Decomposability of $\vdash$). For all $\Phi, M, s \in I$, $\langle R, I \rangle \vdash \Phi \implies \langle R, \{s\} \rangle \vdash \Phi$.

Proof. (See Appendix A.) □

Lemma 3.5 (Composability of $\vdash$). For all $\Phi, I$, ($\forall s \in I. \langle R, \{s\} \rangle \vdash \Phi$) $\implies \langle R, I \rangle \vdash \Phi$.

Proof. (See Appendix A.) □

3.3 Frontiers

The remaining proof rules depend on the existence of a set of states $\mathcal{F} \subseteq S$ that we will call a frontier. The frontier is used as part of the following inductive predicate walk.

Definition 3.1 (Frontier walk). For all $R, I$, and frontier $\mathcal{F}$,

\[
\begin{align*}
R(s, s') &\quad s \not\in \mathcal{F} \quad s \in I \\
\text{walk}^\mathcal{F}_I (s, s') &\quad R(s', s'') \quad s' \not\in \mathcal{F} \\
\text{walk}^\mathcal{F}_I (s, s') &\quad \text{walk}^\mathcal{F}_I (s', s'')
\end{align*}
\]

The above relation $\text{walk}^\mathcal{F}_I$ allows us to characterize the region that includes every possible transition along every trace from $I$ up to, but not including, $\mathcal{F}$. Note that $\text{walk}^\mathcal{F}_I \subseteq R$, and that when $\mathcal{F} = \emptyset$, $\text{walk}^\emptyset_I$ is equivalent to the portion of the transition relation accessible from $I$. Here is an example:
Above we have a set of initial states \( I = \{c, g, k\} \) and a frontier \( \mathcal{F} = \{a, f, i, m, n\} \) denoted as double-lined states. For this example,

\[
\text{walk}^\mathcal{F}_I = \{(c, d), (d, a), (d, h), (h, e), (e, f), (h, m)(k, g), (g, h), (k, l), (l, m), (l, n)\},
\]

which is depicted as the bolded arrows in the following diagram:

![Diagram](image)

We will now consider several lemmas that are important to proving soundness and completeness, before we return to the remaining proof rules. The following lemma says that if a trace reaches a state in a frontier \( \mathcal{F} \), then there will be a transition along that trace that is not in \( \text{walk}^\mathcal{F}_I \).

**Lemma 3.6** (Traces escape \( \text{walk} \)). For all \( \pi, I, \mathcal{F}, n, \pi^n_0 \in \mathcal{F} \implies (\pi^n_0, \pi^{n+1}_0) \notin \text{walk}^\mathcal{F}_I \).

*Proof.* (Coq: escape_front) By induction on \( n \) and definition of \( \text{walk}^\mathcal{F}_I \). \( \square \)

The following lemma states that a path \( \pi \) will either never reach the frontier or else there will be some state in \( \pi \) that is in the frontier.

**Lemma 3.7** (Reach \( \mathcal{F} \) choice). For all \( R, I, \mathcal{F}, \pi \in \Pi(S, R, I) \), either

\[
\forall n. \pi^n_0 \notin \mathcal{F} \quad \text{or else} \quad \exists n. \pi^n_0 \in \mathcal{F} \land \forall m < n. \pi^m_0 \notin \mathcal{F}.
\]

*Proof.* (Coq: Cktl.escape_or_not) Either the first case holds or it does not (axiom of choice). If it does not hold, then we can show that the second case must hold by coinductive reasoning as follows. Assume that \( \exists n'. \pi^n_0 \in \mathcal{F} \land \forall m < n'. \pi^m_0 \notin \mathcal{F} \) holds for some \( \pi \) where \( \text{isTrace} \ R \ \pi \). Now consider some \( s \) such that \( \text{isTrace} \ R \ (s :: \pi) \). For this new trace \( \tilde{\pi} \equiv s :: \pi \) there are two possibilities: (i) \( s \in \mathcal{F} \) and so we can let \( n = 0 \) to satisfy the lemma for \( \tilde{\pi} \) (ii) \( s \in \mathcal{F} \) and so we can let \( n = n' + 1 \) to satisfy the lemma for \( \tilde{\pi} \). \( \square \)

The next lemma states that every path through \( \text{walk} \) is also a path through the original transition relation \( R \).

**Lemma 3.8.** For every \( I, \mathcal{F} \quad \Pi(S, \text{walk}^\mathcal{F}_I, I) \subseteq \Pi(S, R, I) \).

*Proof.* (Coq: walk_is_RF) Follows from the fact that \( \text{walk}^\mathcal{F}_I \subseteq R \). \( \square \)
3.4 Remaining proof rules

We use frontiers in our formulation of AF\(\Phi\) to characterize the places where \(\Phi\) holds, requiring that all paths from \(I\) eventually reach a frontier.

\[
\text{walk}_I^F \text{ is well-founded} \quad \langle R, F \rangle \vdash \Phi \\
\langle R, I \rangle \vdash AF\Phi
\]

Figure 3.4: Eventuality

This proof rule is given in Figure 3.4 and requires that walk\(^F_I\) be well-founded. In this way, we recast the \(\forall\)CTL semantics of AF in terms of the well-foundedness of a relation, rather than the existence of an \(i\)-th state in every trace. This formulation allows us to more efficiently prove AF properties because we can discover well-founded relations that are over-approximations of walk\(^F_I\) rather than searching for per-trace ranking functions.

The soundness and completeness of this rule depends on the following lemma. Lemma 3.9 says that if all traces from a state \(s\) reach a frontier \(F\), then walk\(^F_{\{s\}}\) is well-founded.

Lemma 3.9 (Traces reach frontier). For a frontier \(F\) and state \(s\),

\[
\forall \pi \in \Pi(S, R, \{s\}). \exists i. \pi_i^0 \in F \iff \text{walk}_{\{s\}}^F \text{ is well-founded}
\]

Proof. (Coq: wf_trace_has_n) Using Theorem 2.5 letting \(I = \{s\}\). Then using the law of excluded middle and Lemmas 3.6 and 3.8.

Lemma 3.10 (Soundness and completeness of RAF). For all \(\Phi, R, I\),

\[
\langle R, I \rangle \vdash AF\Phi' \iff M \models AF\Phi'.
\]

Proof. (Coq: Cktl.soundness,Cktl.completeness) (\(\Rightarrow\)) Requires that Theorem 2.5 be applied to the relation walk\(^F_I\), and then proceeds by contradiction.

(\(\Leftarrow\)) We define a frontier \(F\) to be the set of all states such that the subformula \(\Phi'\) holds. We then have two obligations:

1. Show walk\(^F_{\{s\}}\) is well-founded. This holds by using Lemma 3.9 since every \(\pi\) in walk\(^F_{\{s\}}\) starting from \(s\) has an index \(n\) where \(\pi_0^n \in F\), walk\(^F_{\{s\}}\) is well-founded.
2. Show that \(\langle R, F \rangle \vdash \Phi'\). We use Lemma 3.5 to show that

\[
\forall s \in F. \langle R, \{s\} \rangle \vdash \Phi' \Rightarrow \langle R, F \rangle \vdash \Phi'.
\]
Sequencing RAW. The final proof rule is for the \( \text{AW} \) operator, given in Figure 3.5. RAW also uses a frontier and the relation \( \text{walk}_F \) representing the arcs along the way to the frontier \( F \).

The condition \( (R, (\text{walk}_F)|_1) \vdash \Phi_1 \) specifies that \( \Phi_1 \) must hold of all states reachable from \( I \) up to \( F \). To prove \( A[\Phi_1 W \Phi_2] \), all states along the path to the frontier must satisfy \( \Phi_1 \) and states at the frontier—should one ever get there—all must satisfy \( \Phi_2 \).

**Lemma 3.11** (Soundness and completeness of RAW). For all \( \Phi, R, I \),

\[
(R, I) \vdash A[\Phi_1 W \Phi_2] \iff M \models C \Phi_1 W \Phi_2.
\]

**Proof.** (Coq: Cktl.soundness, Cktl.completeness) (⇒) For every trace \( \pi \), Lemma 3.7 says that there are two cases. The possibilities then align with those in the semantics of \( \models_c \).

(⇐) We first define a frontier by the axiom of choice:

\[
F \equiv \{ t \mid \exists \pi. \exists n. \pi_n^0 = t \land (R, \{ t \}) \vdash \Phi_2 \}
\]

Now, the semantics of \( \text{AW} \) gives us two cases:

- \( \forall s_1, s_2. \text{walk}_F(s_1, s_2) \Rightarrow (R, \{ s_1 \}) \vdash \Phi_1 \)
  
  We use an equivalent definition of \( \text{walk}_F \) which is parameterized by \( m \), the number of steps taken from \( s \). We can then show inductively (over \( m \)) that there exists a trace \( \pi \) such that \( \pi_0^m = s_1 \) and \( (R, \{ s_1 \}) \notin \Phi_2 \) (because if \( \Phi_2 \) held, \( s_1 \) would be in frontier \( F \) and \( \text{walk}_F(s) \) would not hold).

Now, for this \( \pi \), it must be the case (because \( s \models_c A[\Phi_1 W \Phi_2] \)) that either:

1. \( \forall i. \pi^i_0 \models_c \Phi_1 \).

   With the inductive hypothesis, we can easily conclude that \( (R, \{ s_1 \}) \vdash \Phi_1 \).

2. \( \exists j. \pi^j_0 \models_c \Phi_2 \land \forall i < j. \pi^i_0 \models_c \Phi_1 \).

   It must be the case that \( j > m \) because recall that \( \Phi_2 \) does not hold up to \( m \).

- \( (R, F) \vdash \Phi_2 \)

  Follows from Lemma 3.5, the definition of \( F \) and the inductive hypothesis. 

**Theorem 3.12** (Soundness and completeness of \( \vdash \)). For all \( \Phi, M = (S, R, I) \),

\[
(R, I) \vdash \Phi \iff M \models_c \Phi.
\]
Proof. (Coq: Cktl.soundness,Cktl.completeness) Consequence of above Lemmas. ■

We also introduce the following notation:

**Definition 3.2** (Lifting ⊢). For all $M = (S, R, I)$ and ∀CTL property $\Phi$,

$$M \vdash \Phi \equiv (R, I) \vdash \Phi.$$  

**Related work.** Most similar to our work is the proof system of Pnueli and Kesten [PK02]. In their work they characterize CTL* temporal operators as elaborate predicates on states, permitting them to develop a deductive proof system. For example, their proof rule for $p \Rightarrow \text{AF} q$ (written $p \Rightarrow A_f \Diamond q$ in Fig. 9 of [PK02]) is given by a finite set of justice requirements $J_i$, a finite set of intermediate assertions $\varphi_i$ and a finite set of rank functions $\delta_i$. The rule is given as an elaborate transition predicate, involving (finite) quantification over these sets. A benefit of their approach is that they can decompose the proof of eventually $q$ into a finite sequence of $m$ steps, each potentially hidden by a justice requirement or accounted for by rank function.

While a proof of $\text{AF} q$ in our case involves reasoning along each path until $q$, we do not include the intermediate predicates in the proof system. Instead, we identify the appropriate fragment of the transition relation (i.e. $\text{walk}_I^\mathcal{F}$) and a general property that must hold of this transition relation (well-foundedness). In this way, we leave the details of how to prove the eventuality to underlying abstraction techniques such as interpolation or abstraction refinement in the encoding (described in Chapter [1]). This simplifies the proof system and leaves it amenable to applying many different forms of reasoning over $\text{walk}_I^\mathcal{F}$.  

---

**Figure 3.6:** Full proof system for ∀CTL.
3.5 Example revisited

For Example 2.2, the ∀CTL property holds but involves an infinite derivation. Instead, we can construct the following finite derivation in the proof system from this chapter:

\[
X \cup Y = (\text{walk}_{F_1}^{X})_{|I_1} \quad \frac{X \subseteq \llbracket \neg x \rrbracket_S}{(R, X) \vdash \neg x} \quad \frac{\text{walk}_{F_2}^{Y} \text{ is well-founded}}{(R, Y) \vdash \text{AF} \neg x} \quad \frac{\mathcal{F}_2 \subseteq \llbracket \neg x \rrbracket_S}{(R, \mathcal{F}_2) \vdash \neg x} \\
\frac{\text{walk}_{F_1}^{X} \text{ is well-founded}}{(R, (\text{walk}_{F_1}^{X})_{|I_1}) \vdash \text{AF} \neg x \lor \neg x} \quad \frac{\mathcal{F}_1 \subseteq \llbracket \text{false} \rrbracket_S}{(R, \mathcal{F}_1) \vdash \text{false}} \\
\frac{(R, I) \vdash \text{A}[\text{AF} \neg x \lor \neg x] \text{ W false}}{
}
\]

where \( \mathcal{F}_1 = \emptyset \) \quad \( X = \mathbb{B} \times \mathbb{N} \times \{\ell_0\} \) \quad \( \mathcal{F}_2 = \{\text{false}\} \times \mathbb{N} \times \{\ell_0\} \) \quad \( Y = \mathbb{B} \times \mathbb{N} \times \{\ell_1\} \)

This leaves us with five obligations, proved below. Notice, however, that this derivation has finitely many inference rules. Moreover, individual elements of sets of states \( I, \mathcal{F}_1, \mathcal{F}_2, X, Y \) and relations \( \text{walk}_{F_1}^{X}, \text{walk}_{F_2}^{Y} \) are not mentioned. Consequently, algorithms and tools can discover finite over-approximations of them (e.g. \( \llbracket \neg x \rrbracket_S \) or \( \text{walk}_{F_2}^{Y} \subseteq \llbracket n' < n \land n > 0 \rrbracket_R \)) as we will see later.

The proof obligations are as follows:

1. \( X \cup Y = (\text{walk}_{\mathcal{F}_1}^{X})_{|I_1} \). Since \( \mathcal{F}_1 = \emptyset \), the right-hand side is the set of all (reachable) states. The LHS is the set of all states. In this example the two are equivalent.
2. \( X \subseteq \llbracket \neg x \rrbracket_S \). Initially when \( pc = \ell_0 \) then \( \neg x \). Moreover, \( x \) only becomes true when control changes to \( \ell_1 \), and then \( x \) becomes false again whenever control changes back to \( \ell_0 \).
3. \( \text{walk}_{F_2}^{Y} \) is well-founded. Substituting, we must show that \( \text{walk}_{F_2}^{Y_{pc=\ell_0}} \subseteq \llbracket \neg x \rrbracket_S \) is well-founded. If we unroll the definition of \( \text{walk} \) we see that \( \text{walk}_{F_2}^{Y_{pc=\ell_0}} \subseteq \llbracket \neg x \rrbracket_S \) is the set of all state transitions from \( \ell_1 \), returning to \( \ell_1 \). This relation is well-founded because there is a ranking function:

\[
f = \lambda \left[ \frac{n}{pc} \right]. n
\]

where the well-order is simply the natural numbers.
4. \( \mathcal{F}_2 \subseteq \llbracket \neg x \rrbracket_S \). Same as 2 above.
5. \( \mathcal{F}_1 \subseteq \llbracket \text{false} \rrbracket_S \). Trivial.
Safety and liveness

There is a long history of temporal verification of finite state systems. In recent decades researchers have also focused on infinite state systems—in particular, programs—as software has increasingly made its way to critical platforms. When interested in proving trace-based properties such as LTL, classic automata theory has been adapted [CGP⁺07]. For state-based properties such as CTL, standard so-called “bottom-up” (or “tableaux”) methods for finite state have been adapted. These adapted CTL algorithms typically involve first abstracting the state space into a finite model, and then applying finite reasoning strategies. As we discuss in Chapter 8, unfortunately none of these tacks have lead to viable tools. For example, when applying abstraction a priori one does not know what is the right kind of abstraction needed to be able to answer termination/reachability questions.

In this dissertation, we describe new methods that allow us to tackle infinite-state space verification in a more tractable way. As we showed in the previous chapter this is accomplished, in part, by splitting the state space into partitions that satisfy various infinite-state criteria (e.g. well-foundedness). We continue in this chapter by showing how to encode the search for such a proof derivation. In particular, we characterize verification as a safety problem in tandem with a search for sufficient termination arguments to solve a liveness problem. This construction poses the problems so that modern abstraction techniques can work together to discover proofs.

Intuitively, we will describe a symbolic encoding \( \text{enc}(P, M, \Phi) \) that is parameterized by the program \( P \) and property \( \Phi \) as well as a set of termination arguments \( M \). We can reason about the behavior of the encoding \( \text{enc} \) using standard abstraction techniques such as abstraction refinement or interpolation. Meanwhile, we can use termination argument refinement to automatically construct \( M \). In this way, the two types of machinery are focused closely on their own pieces that are needed to prove that the property holds (or find a counterexample). An output solution to the tandem problems gives us (typically, a symbolic representation of) the states, frontiers, and termination arguments that comprise a derivation in the proof system from Chapter 3.
Our encoding is described as a guarded transition system in this chapter and is designed to exploit the structure of the temporal property. In particular, our encoding is designed such that only the reachable states that are needed to construct a proof derivation are considered. Abstraction techniques may over-approximate this set of states, but will never need to find approximations for states that don’t appear in a derivation. Moreover, only relevant portions of the state/subformula space are considered.

As we see in Chapter 8, our technique is more amenable to infinite-state transition systems (e.g. programs) because often solutions can be found where bottom-up techniques would diverge.

Note that the algorithm presented here is state-based in nature, allowing us to prove ∀CTL properties. However, in Chapter 5 we describe how it can be used in combination with an iterative symbolic determinization procedure to obtain an efficient algorithm for verifying trace-based properties in LTL. Moreover, in Chapter 6 we will show that an instance of this encoding can be given as a finite set of procedures in a simple while language. Consequently, we can take an input while program $P$, perform a source-to-source translation, and obtain a new program whose correctness proof (obtained from standard program analysis tools for safety and termination) implies that the ∀CTL property holds of $P$.

### 4.1 Preliminaries

As discussed above, this chapter describes an encoding of temporal verification as a safety problem, in tandem with a liveness problem. The encoding is presented in this chapter as a guarded transition system, constructed inductively over the ∀CTL property. We begin with some definitions.

**Definition 4.1.** A guarded transition system $G = (N, W, C^0, \Theta)$ is a finite set of control points $N$, a finite set of typed variables $W$, an initial configuration predicate $C^0$ and a transition predicate $\Theta$ over unhatted ($\hat{W}$) and hatted ($\hat{\hat{W}} \equiv \{ \hat{x} \mid x \in W \}$) variables. The set of variables includes one special variable $nd : N$ denoting the current control point. A configuration $c \in C$ is a valuation for the variables.

Note that we use “hat” rather than “prime.” Primes in this dissertation are typically used to denote multiple instances (e.g. $\Phi$ versus $\Phi'$) where as hatting is used to indicate a state or configuration where variables have been renamed. A hatted configuration $\hat{c}$ is a mapping from (typed) hatted variables $\hat{x}$ to values. For configurations $c_1, c_2$ we say $c_1 \leadsto c_2$ if and only if $\Theta_\Phi(c_1)(\hat{c}_2)$. In order to decompose a transition predicate, we lift

---

1We use $nd$ because $pc$ is reserved for the program counter in input transition systems used in the encoding in Section 4.2.
disjunction over transition predicates:

\[ \Theta_1 \lor \Theta_2 \equiv \lambda c_1 \, \hat{c}_2. \Theta_1(c_1)(\hat{c}_2) \lor \Theta_2(c_1)(\hat{c}_2). \]

We will describe \( \Theta \) symbolically, using guards and actions as follows. A guard \( g \) is a predicate on a configuration, and an action \( a : (W \times \varepsilon) \text{ list} \) is a list of tuples consisting of a hatted variable and a boolean or arithmetic expression \( \varepsilon \) that may involve unhatted variables. For example, we may have

\[ a = \{(\hat{x}, 5), (\hat{y}, x)\} \quad \text{or, notationally:} \quad x := 5 \ ; \ y := x \quad \text{or, alternatively:} \quad \hat{x} = 5 \land \hat{y} = x. \]

We use the following notation:

\[
\frac{\{g\}}{\lambda c_1 \, \lambda c_2. \ c_1(\text{nd}) = n_1 \land c_2(\text{nd}) = n_2 \land g(c_1) \land}
\]

\[
\begin{align*}
\forall x \in W \left( c_2(\hat{x}) = c_1(x) \quad \text{if } \exists (x, \varepsilon) \in a \\
\bigwedge x \in W \left( c_2(\hat{x}) = \varepsilon \quad \text{if } \exists (x, \varepsilon) \in a \right)
\end{align*}
\]

Guarded transition systems are commonly used as the internal representation of programs in program analysis tools. Similar formulations can be found for representing interprocedural control flow [RHS95]. Such representations allow us to symbolically represent systems (e.g. programs) whose transition relation has a finite representation but whose state spaces are infinite.

**Example.** Returning to Example 2.2, we can give a guarded transition system \( \mathcal{G} = (N, W, C^0, \Theta) \) instead of a simple transition system, where:

\[
\begin{align*}
N & \equiv \{\text{nd}_0, \text{nd}_1\}, \\
W & \equiv \{x : \mathbb{B}, \ n : \mathbb{N}, \ \text{nd} : N\}, \\
C^0 & \equiv \text{nd}_0 \land \neg x, \\
\Theta & \equiv \text{nd}_0\{\text{true}\} \xrightarrow{x := \text{true}; \ n := \ast + 1} \text{nd}_1 \\
& \lor \text{nd}_1\{n > 0\} \xrightarrow{n := n - 1} \text{nd}_1 \\
& \lor \text{nd}_1\{n = 0\} \xrightarrow{x := \text{false}} \text{nd}_0.
\end{align*}
\]

Visually, we can think of this as the following:
4.2 Encoding

We encode the task of \( \forall \text{CTL} \) verification as a safety problem in tandem with a liveness problem. We will describe the encoding as a guarded transition system \( G_{M,\Phi}^M \) that is parameterized by the program \( P \) and property \( \Phi \) as well as a set of termination arguments \( M \). This encoding is built inductively over the property \( \Phi \), and is designed such that abstraction can be applied in concert with reachability and termination reasoning. When these tools are applied on the encoding, they are effectively coerced into performing the reasoning necessary for proving that \( \Phi \) holds of \( P \).

Specifically, the encoding characterizes a search for a derivation (in the proof system from Chapter 3) as the task of proving that a guarded transition system cannot fault. In Chapter 6 we can specialize this to program analysis tools that search for assertion (reachability) violations.

**Definition 4.2 (Encoding).** For all \( M = (S, R, I) \), \( \forall \text{CTL} \) property \( \Phi \), and finite set of measures \( M \), the guarded transition system encoding \( G_{M,\Phi}^M = (N_\Phi, W_\Phi, C_\Phi^0, \Theta_\Phi^M(M)) \) where:

\[
N_\Phi \equiv \{en, ex\} \times sub(\Phi),
\]

\[
W_\Phi \equiv \{nd : N_\Phi\} \cup \bigcup_{(\kappa, \psi) \in sub(\Phi)} \{\text{fault}_\psi^\kappa : \mathbb{B}, \ \sigma_\psi^\kappa : S, \ \text{dup}_\psi^\kappa : \mathbb{B}, \ \\text{‘}\sigma_\psi^\kappa : S\},
\]

\[
C_\Phi^0 \equiv \lambda c. c(nd) = (en, e, \Phi) \wedge c(\sigma_\Phi^e) \in I \wedge \bigwedge_{(\kappa, \psi) \in sub(\Phi)} c(\text{dup}_\psi^\kappa) = \text{false},
\]

\[
\Theta_\Phi^M(M) \equiv (\text{see Section 4.3}).
\]

We often will refer to the guarded transition system in which \( M \) has not yet been specified, using the notation \( G_{(\cdot),\Phi}^M \equiv \lambda x. G_{x,\Phi}^M \). The above definition is inductive over the \( \forall \text{CTL} \) property \( \Phi \). Consequently there are finitely many nodes, two for each \( (\psi, \kappa) \)-subformula of \( \Phi \): an entry node denoted \( (en, \kappa, \psi) \) and an exit node denoted \( (ex, \kappa, \psi) \). The root nodes are denoted \( (en, e, \Phi) \) and \( (ex, e, \Phi) \). For Example 2.2 we have:

\[
N_\Phi = \{ (en, e, \text{AG}[\neg x \vee \text{AF} \neg x]), (ex, e, \text{AG}[\neg x \vee \text{AF} \neg x]), (en, Le, [\neg x \vee \text{AF} \neg x]), (ex, Le, [\neg x \vee \text{AF} \neg x]), (en, LLe, \neg x), (ex, LLe, \neg x), (en, RRL, AF \neg x), (ex, RRL, AF \neg x), (en, LRL, \neg x), (ex, LRL, \neg x) \}
\]

**Definition 4.3 (Run, complete run, fault).** For all \( G_{M,\Phi}^M \), and \( (\kappa, \psi) \in \text{sub}(\Phi) \),

1. A \( (en, \kappa, \psi) \)-run is a sequence \( c_0, c_1, c_2, \ldots \) such that \( c_0(nd) = (en, \kappa, \psi) \) and \( \forall i \in \mathbb{N}, c_i \sim c_{i+1} \).
2. A \( (en, \kappa, \psi) \)-run, \( c_0, \ldots, c_n \) is a complete run if and only if \( c_n(nd) = (ex, \kappa, \psi) \).
3. A complete \( (en, \kappa, \psi) \)-run \( c_0, \ldots, c_n \) does not fault if and only if \( c_n(\text{fault}_\psi^\kappa) = \text{false} \).
4. \( G_{M,\Phi}^M \) cannot fault if and only if every complete \( (en, e, \Phi) \)-run does not fault.
As noted above, if this inductively defined guarded transition system $G_{M,\Phi}^M$ can be proved to never fault, it must be the case that the property $\Phi$ holds of every initial state $I$. This is captured by Theorem 4.3 from Section 4.3 and proved in Appendix B:

$$\forall M, \Phi. \exists M, G_{M,\Phi}^M \text{ cannot fault } \Rightarrow M \models \Phi.$$  

### 4.3 The transition predicate

Runs of $G_{M,\Phi}^M$ explore the $S \times \text{sub}(\Phi)$ state space, tracking the current state with a per-subformula variable $\sigma_{\psi_1}^\kappa$, starting with a configuration $c_0$ at the root node: $c_0(\text{nd}) = (\text{en}, \epsilon, \Phi)$ and at an initial state: $c_0(\sigma_{\psi_1}^\kappa) = s \in I$. At each successive transition from a node $(\text{en}, \kappa, \psi_1)$ into a subnode $(\text{en}, \lambda\kappa, \psi_2)$ where $(\lambda\kappa, \psi_2) \in \text{sub}(\psi_1)$, $G_{M,\Phi}^M$ is attempting to determine whether subformula $\psi_1$ holds of a particular state $s$ recorded in the variable $\sigma_{\psi_2}^{\lambda\kappa}$. Rather than explicitly tracking this information, however, a $(\text{en}, \lambda\kappa, \psi_2)$-run simply *faults* whenever $\psi_2$ does not hold of $s$. Failure is recorded by setting $\text{fault}_{\psi_2}^{\lambda\kappa}$ to true and enforcing a transition to $(\text{ex}, \lambda\kappa, \psi_2)$.

What remains is to understand how the failure behavior of a complete $(\text{en}, \kappa, \psi)$-run (which, as per Definition 4.3, begins at $(\text{en}, \kappa, \psi)$, ends at $(\text{ex}, \kappa, \psi)$) determines whether a subformula $(\psi, \kappa)$ holds of a state $s$. To this end, we now explore the transition predicate $\Theta_{\Phi}^\kappa$. Note that $\Theta_{\Phi}^\kappa$ is parameterized by $M$ (but we will write $\Theta_{\Phi}^\kappa$ instead of $\Theta_{\Phi}(M)$ for notational convenience), and given over an alphabet of unhatted variables $W_{\Phi}$ and hatted variables $\hat{W}_{\Phi}$.

The transition predicate $\Theta_{\Phi}^\kappa$ is defined inductively over the structure of $\Phi$, with one case for each $(\kappa, \psi) \forall\text{CTL subformula}:

\[
\begin{align*}
\Theta_{\alpha}^\kappa & \equiv \ldots \quad (\text{see below}) \\
\Theta_{\psi_1 \land \psi_2}^\kappa & \equiv \Theta_{\psi_1}^L \lor \Theta_{\psi_2}^R \lor \ldots \quad (\text{see below}) \\
\Theta_{\psi_1 \lor \psi_2}^\kappa & \equiv \Theta_{\psi_1}^L \lor \Theta_{\psi_2}^R \lor \ldots \quad (\text{see below}) \\
\Theta_{A \psi_1}^\kappa & \equiv \Theta_{\psi_1}^L \lor \ldots \quad (\text{see below}) \\
\Theta_{\forall \psi_1 W \psi_2}^\kappa & \equiv \Theta_{\psi_1}^L \lor \Theta_{\psi_2}^R \lor \ldots \quad (\text{see below})
\end{align*}
\]

**Atomic proposition**

\[
\begin{align*}
\Theta_{\alpha}^\kappa & \equiv \quad \begin{cases} 
(\text{en}, \kappa, \alpha) \{\sigma_{\alpha}^\kappa \in \llbracket \alpha \rrbracket^S \} \xrightarrow{\text{fault}_{\alpha}^\kappa=\text{false}} (\text{ex}, \kappa, \alpha) \\
(\text{en}, \kappa, \alpha) \{\sigma_{\alpha}^\kappa \notin \llbracket \alpha \rrbracket^S \} \xrightarrow{\text{fault}_{\alpha}^\kappa=\text{true}} (\text{ex}, \kappa, \alpha)
\end{cases} \\
\text{or, visually:}
\end{align*}
\]

\[
\begin{array}{c}
\text{en, } \kappa, \alpha \\
\{\sigma_{\alpha}^\kappa \in \llbracket \alpha \rrbracket \} \text{fault}_{\alpha}^\kappa := \text{false}
\end{array} \xrightarrow{\text{false}}
\begin{array}{c}
\text{ex, } \kappa, \alpha \\
\{\sigma_{\alpha}^\kappa \notin \llbracket \alpha \rrbracket \} \text{fault}_{\alpha}^\kappa := \text{true}
\end{array}
\]
The base case is for an atomic proposition $\alpha$ and is the simplest. There are two possible transitions from $(en, \kappa, \alpha)$. In the top transition, the state $\sigma^\kappa_\psi$ is an $\alpha$-state. Every run that takes this transition to $(ex, \kappa, \alpha)$ does not fault. Consequently, if we can show that $G^M_{M, \alpha}$ cannot fault from every $C^0$ configuration (i.e. for all $s \in I$), it is easy to see that this implies that $M \models \psi$. If there exists a state $s \in I$ for which $s \notin [\alpha]^S$, then there will be a run that takes the bottom transition and faults, preventing us from proving that $G^M_{M, \alpha}$ cannot fault.

The remaining $\forall$CTL subformula constructors are inductive. By storing the state $s$ in the per-subformula variable $\sigma^\kappa_\psi$ we can consider multiple $\forall$CTL branching scenarios. We will see this, for example, in the next constructor.

\[ \Theta^{\kappa}_\psi_1 \land \psi_2 \equiv \Theta^{\kappa}_{\psi_1} \lor \Theta^{\kappa}_{\psi_2} \]

\[ \lor (en, \kappa, \psi_1 \land \psi_2)\{true\} \xrightarrow{\sigma^{L\kappa}_{\psi_1} = \sigma^{L\kappa}_{\psi_1 \land \psi_2}} (en, \kappa, \psi_1) \]

\[ \lor (en, \kappa, \psi_1 \land \psi_2)\{true\} \xrightarrow{\sigma^{R\kappa}_{\psi_1} = \sigma^{R\kappa}_{\psi_1 \land \psi_2}} (en, \kappa, \psi_2) \]

\[ \lor (ex, \kappa, \psi_1)\{true\} \xrightarrow{\mathrm{fault}^{R\kappa}_{\psi_1 \land \psi_2} = \mathrm{fault}^{L\kappa}_{\psi_1}} (ex, \kappa, \psi_1 \land \psi_2) \]

\[ \lor (ex, \kappa, \psi_1)\{true\} \xrightarrow{\mathrm{fault}^{R\kappa}_{\psi_1 \land \psi_2} = \mathrm{fault}^{R\kappa}_{\psi_1}} (ex, \kappa, \psi_1 \land \psi_2) \]

Conjunction ensures that all possibilities are considered by establishing feasible from $(en, \kappa, \psi_1 \land \psi_2)$-runs to each subformulae. From every configuration at node $(en, \kappa, \psi_1 \land \psi_2)$ where $\sigma^{\kappa}_{\psi_1 \land \psi_2} = s$, there is a complete set of $(en, \kappa, \psi_1)$-runs where $\sigma^{L\kappa}_{\psi_1} = s$, and similar for $\psi_2$. Consequently, if it is possible for a $(en, \kappa, \psi_1 \land \psi_2)$-run to fault, then there is a $(en, \kappa, \psi_1 \land \psi_2)$-run that faults, and similar for $\psi_2$. Thus, if we can prove that no $(en, \kappa, \psi_1 \land \psi_2)$-run faults from every $C^0_{\psi_1 \land \psi_2}$ configuration (i.e. for all $s \in I$), then it must be the case that there is no $(en, \kappa, \psi_1 \land \psi_2)$-run or $(en, \kappa, \psi_2)$-run that faults and hence, via an inductive argument, $M \models \psi_1 \land \psi_2$.

**Existential choice.** When a particular $\psi$ is a $\lor$ or AF subformula constructor, $G^M_{M, \Phi}$ enables $(en, \kappa, \psi)$-runs at a state $s$ to consider all of the possible cases that might cause $\psi$ to hold.
of \( s \). If one is found the run does not fault. Otherwise, the run faults if none are found. This is the intuition behind the first invariant maintained by \( G_{M, \Phi}^M \):

**Lemma 4.1** (INV\(_1\)). For all \( s, \psi, \kappa, \) and \( c_0 \) such that \( c_0(\text{nd}) = (\text{en}, \kappa, \psi) \) and \( c_0(\sigma_\psi^c) = s \),

\[
R, s \notin \psi \implies \exists \text{ a complete run } c_0, ..., c_n \text{ that faults}
\]

**Proof.** By induction on the transition predicate \( \Theta_\Phi \).

\[
\Theta_{\psi_1 \vee \psi_2}^L \equiv \Theta_{\psi_1}^L \vee \Theta_{\psi_2}^R \vee \{\text{true}\} \sigma_{\psi_1}^L \equiv \sigma_{\psi_1 \vee \psi_2}^L \xrightarrow{\text{fault}_{\psi_1 \vee \psi_2}^L = \text{false}} (\text{en}, \kappa, \psi_1) \\
\vee (\text{ex}, \kappa, \psi_1) \{\text{false}_{\psi_1}^L \} \xrightarrow{\text{fault}_{\psi_1 \vee \psi_2}^L = \text{false}} (\text{ex}, \kappa, \psi_1 \vee \psi_2) \\
\vee (\text{ex}, \kappa, \psi_1) \{\text{true}_{\psi_1}^L \} \xrightarrow{\text{fault}_{\psi_1 \vee \psi_2}^L = \text{false}} (\text{ex}, \kappa, \psi_1 \vee \psi_2)
\]

or, visually:

\[
\xrightarrow{\text{true}_{\psi_1}^L} \xrightarrow{\text{false}_{\psi_1}^L} \xrightarrow{\text{true}_{\psi_1}^L}
\]

In the case of disjunction, we want to know that one of the subformulae \( \psi_1 \) or \( \psi_2 \) holds. \( G_{M, \Phi}^M \) ensures that if, for a given state \( s \), the disjunction holds, then there will be no \((\text{en}, \kappa, \psi_1 \vee \psi_2)\)-run that faults. To this end, a transition is made first to \((\text{en}, \kappa, \psi_1)\). From there, if the run can fault then \( \psi_1 \) does not hold of \( s \), so a transition is made to \((\text{en}, \kappa, \psi_1)\). If \( \psi_2 \) does not hold of \( s \) the \((\text{en}, \kappa, \psi_2)\)-run will fault and so the run faults and makes a transition to \((\text{ex}, \kappa, \psi_1 \vee \psi_2)\). If, however, either \( \psi_1 \) holds or \( \psi_2 \) holds, then it will be impossible for a \((\text{en}, \kappa, \psi_1 \vee \psi_2)\)-run to fault. Either the \((\text{en}, \kappa, \psi_1)\)-run will not fault or else the \((\text{en}, \kappa, \psi_2)\)-run will not fault, and so a transition will be taken to \((\text{ex}, \kappa, \psi_1 \vee \psi_2)\) without faulting.
For a state \( s \), the \( \forall \text{CTL} \) formula \( A[\psi_1 W \psi_2] \) holds if, on every trace from \( s \), \( \psi_1 \) holds at each step until the first instance that \( \psi_2 \) holds (which may never occur). The search for a proof of an AW subformula consists of a loop, checking first to see if we have reached the point where \( \psi_2 \) holds (via \( \Theta_{\psi_2}^R \)) and if not, ensuring that \( \psi_1 \) still holds (via \( \Theta_{\psi_1}^L \)) and stepping forward in the transition relation. When we find that \( \psi_2 \) holds (\( \text{fault}^R_{\psi_2} = \text{false} \)) then a non-faulting transition is made to \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \), completing the run. If \( \psi_2 \) does not hold, and \( \psi_1 \) also does not hold, then a faulting transition is made to \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \), completing the run. Additionally, for every reachable state, there is a transition that can be taken from \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \) to \( (\text{ex}, \kappa, A[\psi_1 W \psi_2]) \), ending the run.

Let us consider the case where for a given trace \( s, s_0, s_1, ... \), there is some state \( s_i \) such that \( \psi_2 \) holds and for all \( 0 \leq j < i \), \( \psi_1 \) holds of \( s_j \). Runs will loop, checking for each \( k \geq 0 \) first that \( \psi_2 \) holds. If it holds, the \( (\text{en}, \kappa, \psi_2) \)-run will not fault, and a non-faulting transition will be made to \( (\text{ex}, \kappa, A[\psi_1 W \psi_2]) \), completing the run. If \( \psi_2 \) does not hold yet, a transition will be made to initiate a \( (\text{en}, \kappa, \psi_1) \)-run from \( s_k \). Since \( \psi_1 \) holds of \( s_k \), a transition to \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \) will be made, advancing the state, and the \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \)-run will not fault. Consequently, no \( (\text{en}, \kappa, A[\psi_1 W \psi_2]) \)-run will fault.
Now let us consider the other case where, for a given trace \( s, s_0, s_1, \ldots, \psi_1 \) holds for all \( s_i \) where \( i \geq 0 \) and \( \psi_2 \) never holds. Every inner \((\text{en}, \kappa, \psi_2)\)-run will therefore fault, and a transition will be made to initiate a \((\text{en}, \kappa, \psi_1)\)-run. Each such \((\text{en}, \kappa, \psi_1)\)-run will not fault, and a transition will be made to \((\text{en}, \kappa, A[\psi_1 W \psi_2])\) from which point, the run can complete and not fault. Consequently, the only possible \((\text{en}, \kappa, A[\psi_1 W \psi_2])\)-runs are runs that loop and, at some point, complete without faulting.

In the diagram above, we use the notation \( \{\neg \text{fault}_{\psi_1}^{\lambda_{\psi_1}}\} \text{next}(\sigma_{\psi_1 W \psi_2}^{\kappa}) \) to indicate a set of possible transitions. That is, for every \( s' \) such that \( R(\sigma_{\psi_1 W \psi_2}^{\kappa}, s') \), there is an arc where the guard \( \neg \text{fault}_{\psi_1}^{\lambda_{\psi_1}} \) holds and the assignment \( \sigma_{\psi_1 W \psi_2}^{\kappa} := s' \) is made.

**Backtracking.** Notice that in the above definition of \( \Theta_{A[\psi_1 W \psi_2]}^{\kappa} \) that from every reachable value for \( \sigma_{\psi_1 W \psi_2}^{\kappa} \) there is a run that can complete. This construction allows our treatment of \( AWF \) (and of \( AF \)) to consider every state that is reachable after finitely many steps along a trace from \( s \). If it were possible for runs through a subformula \( \kappa, \psi \) to diverge (thus never reaching \( (\text{ex}, \kappa, \psi) \)) then we could not determine the failure behavior of the outer nodes. To this end, \( \Theta_{\Phi}^{\kappa} \) maintains a second invariant:

**Lemma 4.2 (INV)\(_{2} \).** For all \( s, \psi, \kappa \) and \( c \) such that \( c(\text{nd}) = (\text{en}, \kappa, \psi) \) and \( c(\sigma_{\psi}^{\kappa}) = s \), there is a complete run from \( c \).

**Proof.** By induction on \( \Theta_{\Phi}^{\kappa} \). \( \square \)

One can think of this as a form of backtracking. A transition to \( (\text{ex}, \kappa, \psi) \) does not mean that the property holds (there must never be a run that faults to do that). Instead, a transition to \( (\text{ex}, \kappa, \psi) \) means that a safety analysis can freely backtrack and switch to other possible scenarios during its search for a proof.
Eventuality

\[ \Theta^\kappa_{\psi_1} = \Theta^L_{\psi_1} \]
\[ \vee (en, \kappa, AF_{\psi_1}) \{true \} \xrightarrow{\text{fault}_{AF_{\psi_1}} = \text{false}} (ex, \kappa, AF_{\psi_1}) \]
\[ \vee (en, \kappa, AF_{\psi_1}) \{true \} \xrightarrow{\delta^L_{\psi_1} = \sigma^L_{AF_{\psi_1}} = \sigma^L_{\psi_1}} (en, \kappa, \psi_1) \]
\[ \vee (ex, \kappa, \psi_1) \{-\text{fault}_{\psi_1} \} \xrightarrow{\text{fault}_{AF_{\psi_1}} = \text{false}} (ex, \kappa, AF_{\psi_1}) \]
\[ \forall s, s' \in R \ (ex, \kappa, \psi_1) \{\text{fault}^L_{\psi_1} \land \sigma^L_{\psi_1} = s \} \xrightarrow{\delta^L_{\psi_1} = s'} (en, \kappa, \psi_1) \]
\[ \forall s, s' \in R \ (ex, \kappa, \psi_1) \{\text{fault}^L_{\psi_1} \land -\text{dup}_{AF_{\psi_1}} \land \sigma^L_{\psi_1} = s \} \]
\[ \xrightarrow{\text{dup}_{AF_{\psi_1}} = \text{true} \land \sigma^L_{AF_{\psi_1}} = \sigma^L_{\psi_1} \land \delta^L_{\psi_1} = s'} (en, \kappa, \psi_1) \]
\[ \vee (ex, \kappa, \psi_1) \{\text{fault}^L_{\psi_1} \land \text{dup}_{AF_{\psi_1}} \land \exists f \in M. f(\sigma^L_{\psi_1}) < f(\sigma^L_{AF_{\psi_1}}) \} \]
\[ \xrightarrow{\text{true}} (ex, \kappa, AF_{\psi_1}) \]

or, visually:

For eventuality, \( G^M_{M, \psi} \) cannot fault provided that all sequences of transition system states from an initial state eventually reach a state where the subformula holds. Runs from \((en, \kappa, AF_{\psi})\) first transition to \((en, \kappa, \psi)\), assigning the current state to \(\sigma^L_{\psi}\). If a \((en, \kappa, \psi)\)-run for this valuation of \(\sigma^L_{\psi}\) cannot fault, then this \((en, \kappa, AF_{\psi})\)-run will not fault. Otherwise, a step is taken in the transition relation \((next())\), and the a new \((en, \kappa, \psi)\)-run begins.

Rank functions. While exploring these reachable states, at every point a nondeterministic choice may be made to the current state. This choice is indicated by setting the variable \(\text{dup}_{AF_{\psi}}\) to \text{true}, and duplicating the current state: \(\sigma^L_{AF_{\psi}} := \sigma^L_{\psi}\). When this happens, a check is performed on every subsequent step to determine if there is some rank function \(f \in M\) that witnesses the well-foundedness of this particular subset of the transitive closure of the transition system.

---

\(^2\)This is an adaptation of a known technique [CPR06]. However, rather than using \text{assert} to check that one of the ranking functions in \(M\) holds, \(G^M_{M, \psi}\) simply faults, allowing other possibilities to be considered (if any exist) in outer disjunctive or AF formulæ.
Ultimately we must find a finite set of ranking functions $\mathcal{M}$ such that $\mathcal{G}_{\mathcal{M}, \Phi}$ cannot fault. In this Chapter, we assume that a suitable $\mathcal{M}$ is given to us. However, in Chapter 5 we show how to adapt a known method [CPR06] in order to iteratively find a sufficient $\mathcal{M}$.

### 4.4 Example

**Example.** For Example 6.2, we replace implication with disjunction in the formula to obtain $\Phi \equiv \text{AG}[\text{AF}(x = 0) \vee x = 0]$. The encoding we obtain is as follows:

\[
N_{\Phi} = \{(en, \epsilon, \text{AG}[\text{AF}(x = 0) \vee x = 0]), (ex, \epsilon, \text{AG}[\text{AF}(x = 0) \vee x = 0])\},
\]

\[
(\text{en, le}, \text{AF}(x = 0) \vee x = 0), (\text{ex, le}, \text{AF}(x = 0) \vee x = 0),
\]

\[
(\text{en, lle}, \text{AF}(x = 0)), (\text{ex, lle}, \text{AF}(x = 0)),
\]

\[
(\text{en, llle}, x = 0), (\text{ex, llle}, x = 0),
\]

\[
(\text{en, rle}, x = 0), (\text{ex, rle}, x = 0)
\]\n
\[
C_{\Phi} = \lambda c. c(\text{nd}) = (\text{en, } \epsilon, \text{AG}) \land c(\sigma_{AG}^+) \in I \land c(\text{dup}) = \text{false}
\]

\[
\Theta_{\Phi} = (\text{see Figure 4.1})
\]

Note that in Figure 4.1 we use the following abbreviations of subformulae:

\[
\text{AG} \equiv \text{AG}[\text{AF}(x = 0) \vee x = 0]
\]

\[
\vee \equiv \text{AF}(x = 0) \vee x = 0
\]

\[
\text{AF} \equiv \text{AF}(x = 0)
\]
Since there is only one $\text{AF}$ case in the encoding, we assume only one $\text{dup}$ variable. The transition predicate $\Theta_\Phi$ is given in Figure 4.1. Recall that $\text{AG}\Phi = A[\Phi \text{ W false}]$. Consequently, some arcs given by $\Theta_{\text{AW}}$ are unneeded, and omitted in Figure 4.1. In two cases we use the notation “$\sigma^*_\text{AG} := R(\sigma^*_\text{AG})$,” which is to say that a successor in $R$ is chosen nondeterministically.

4.5 Correctness of the encoding

**Theorem 4.3** (Soundness of $\mathcal{G}^M_{M,\Phi}$).

$$\forall M, \Phi, \exists M. \mathcal{G}^M_{M,\Phi} \text{ cannot fault } \Rightarrow M \not\models \Phi.$$ 

*Proof.* (See Appendix B.) $\square$

**Theorem 4.4** (Relative completeness of $\mathcal{G}^M_{M,\Phi}$).

$$\forall M, \Phi, \exists M. \mathcal{G}^M_{M,\Phi} \text{ cannot fault } \Leftarrow M \not\models \Phi.$$ 

Provided that each ranking function $f \in M$ is enumerable (e.g. represented as a possibly infinite list of state/rank pairs).

*Proof.* (See Appendix B.) $\square$

4.6 Related work

We will now describe related work on $\forall$CTL verification. There are many tools and techniques for proving temporal properties of transition systems of finite-state systems (e.g. BC*92, CES86, KVW00). A common approach is the so-called bottom-up method [CES86]. This algorithm considers all state/subformula pairs. It begins with the pairs consisting of atomic propositions $(s, \alpha)$ (subformulæ of length 1) and works upwards to the full formulæ (subformulæ of length $n$). At each stage, the algorithm labels every state $s$ with label $\psi$ provided that subformula $\psi$ holds of $s$. Note that the algorithm assumes that the set of states is available, and therefore does not directly apply to the infinite-state context.

**Infinite-state systems.** Some work has been done on verification techniques for certain classes of infinite-state systems with specific structure (e.g. pushdown systems [Wal96, Wal00] or parameterized systems [EN96]). These, again, do not apply to general-purpose programs that have have infinite-state spaces with no imposed structure.
Other work has attempted to apply finite-state, bottom-up methods for state-based logics (e.g. [CGP99, DP99, FPPS10]) to infinite-state transition systems. These works must somehow address the fact that the state space cannot be enumerated, nor can reachability. Chaki et al. [CCG*05] attempt to address the problem by first computing a finite abstraction of the system \textit{a priori} that is never refined again. Bottom-up techniques are then applied. In our approach we reverse the order: rather than applying abstraction first, we let the underlying program analysis tools perform abstraction after we have encoded the search for a proof as a new program. This strategy facilitates abstraction refinement: after our encoding has been generated, the underlying program analysis tool can iteratively perform abstraction and refinement. Schmidt and Steffen [SS98] take a similar tack to Chaki et al.

The tool YASM [GWC06] takes an alternative approach: it implements a refinement mechanism that examines paths which represent abstractions of tree counterexamples (using multi-valued logic). This abstraction loses information that limits the properties that YASM can prove (e.g. the tool will usually fail to prove $\text{AFAG}_p$). With our encoding the underlying tools are performing abstraction-refinement over tree counterexamples. Moreover, YASM is primarily designed to work for unnested existential properties [Gur10] (e.g. $\text{EF}_p$ or $\text{EG}_p$), whereas our focus is on precise support for arbitrary (possibly nested) universal properties.

\textbf{Comparison to our work.} We use a symbolic encoding (discussed earlier in this chapter) in order to check whether a subformulae holds. This stands in contrast to the standard method of directly tracking the valuations of subformulae in the property with additional variables. In Chapter 6 we will show that our method allows us to apply advanced abstraction techniques found in modern program analysis tools. As an interprocedural analysis computes procedure summaries it is in effect symbolically tracking the valuations of these subformulae depending on the context of the encoded system’s state. Thus, in contrast to bottom-up techniques, ours only considers reachable states (via the underlying program analysis). A safety analysis for infinite-state systems will of course over-approximate this set of states, but it will never need to find approximations for unreachable states. By contrast, bottom-up-based algorithms require that concrete unreachable states be considered. Furthermore, in our technique, only relevant state/subformula pairs are considered. Our encoding will only consider a pair $s, \psi$ where $R, s \Vdash \psi$ is needed to either prove the outermost property, or is part of a valid counterexample. For example, let us say the state space is \{s_0, s_1, s_2\} and the transition relation is \{(s_0, s_1), (s_1, s_2), (s_2, s_2)\} and we want to know whether the property $p \land q$ holds, where $p,q$ are atomic propositions. Our encoding explores the cases $s_0 \Vdash p \land q$, $s_0 \Vdash p$, and $s_0 \Vdash q$, but not the cases $s_1 \Vdash p \land q$, $s_2 \Vdash p \land q$, $s_1 \Vdash p$, $s_1 \Vdash q$, $s_2 \Vdash p$, or $s_2 \Vdash q$. A bottom-up algorithm will explore these superfluous cases.
Games. As noted in Chapter 2, ∀CTL verification has previously been given in the form of finding winning strategies in finite-state games or game-like structures such as alternating automata [BVW94, KVW00, Shi96]. The encoding we presented in this Chapter is, effectively, a generalization of prior work to games over infinite state spaces.

The problem of proving that the guarded transition system $G^M_{M, \Phi}$ cannot fault can be expressed as the problem of finding a strategy for Player 0 in a two-player game. Player 0 attempts to discover nondeterministic choices for existential paths (in $\lor$ and $AF$ cases) such that Player 0 wins (by ensuring that all atomic propositions are satisfied). Meanwhile, Player 1 attempts to foil Player 0 by requiring that all possible universal paths (in $\land$ and $AW$ cases) are considered.

We have chosen the guarded transition system representation specifically because it allows us to encode the check that eventualities ($AF$) are satisfied. Guards and updates are used to encode the check that, for a given $walk^F_X$, that the non-reflexive transitive closure of $walk^F_X$ is well-founded.
As we show in Chapter 8, the methods we have developed for temporal verification surpass previous techniques. Part of their success is due to the fact that they are grounded in a (universal-only) state-based symbolic exploration that is amenable to modern abstraction techniques. (This should also not be surprising given that, in finite domains, CTL is polynomial and LTL is exponential.)

However, we are ultimately concerned with proving trace-based properties. Logics such as LTL allow one to specify the behaviors over traces, which is often more intuitive than state-based logics. This can be seen, for example, in that trace-based logics are common in industrial specification languages such as Linear Temporal Logic or LTL [Pnu77], Interval Temporal Logic [Mos85], Property Specification Language [Acc03], and SVA [VR05].

In Chapter 2 we reminded the reader that one can often over-approximate an LTL property in $\forall$CTL. If we replace $F$s and $G$s with $AF$s and $AG$s and then prove that the $\forall$CTL property holds, then the LTL property must also hold. The fact that you can approximate LTL with $\forall$CTL is promising because in most cases the approximation is sufficient to prove the property or discover a valid counterexample. Thus, we can immediately apply the methods described in Chapter 3 and 4.

However, this is indeed only an approximation. When an $\forall$CTL counterexample is returned we do not know whether it means the LTL property holds: the $\forall$CTL counterexample may be spurious. The problem in these cases is that some nondeterministic choices in the transition system appear in $\forall$CTL counterexamples, and foil our ability to prove the trace-based LTL property.

In this chapter, we show how you can trade the nondeterminism in the transition relation for nondeterminism in the state space. We describe a partial symbolic determinization that enables us to eliminate problematic nondeterministic choices that appear in $\forall$CTL counterexamples that we have characterized as decision predicates. For each pair of decision predicates, our method introduces a prophecy variable [AL91] into the state space that is not externally visible but predicts the outcome of these nondeterministic choices.
Thus the behavior of the program is preordained. This determinization allows us to eliminate spurious $\forall$CTL counterexamples on demand and prove more LTL properties than we could simply via approximation. Finally, we show that our determinization preserves trace equivalence with the original machine.

In this chapter we develop the technical groundwork for symbolic determinization, but assume that decision predicates are given to us by an oracle. In Chapter 7 we will describe an iterative symbolic determinization procedure that examines $\forall$CTL counterexamples and, when they are spurious, synthesizes decision predicates.

### 5.1 Example

**Example 5.1.** We return to Example 2.1: Let $M = (S, R, I)$ where

$S \equiv \mathbb{B} \times \{\ell_1, \ell_2, \ell_3, \ell_4\}$, denoted $[x_{pc}]$

$R = \{(\left[\text{true} \ell_1\right], \left[\text{true} \ell_2\right]), (\left[\text{true} \ell_2\right], \left[\text{true} \ell_3\right]), (\left[\text{true} \ell_3\right], \left[\text{true} \ell_4\right]), (\left[\text{true} \ell_4\right], \left[\text{true} \ell_4\right])\}$,

$I \equiv \{[\text{true} \ell_1]\}$.

This example, again, can be visualized as:

```
   \ell_1
    \downarrow
  \{ x = true \}  \ell_2 \quad x := false \quad \ell_3 \quad x := true \quad \ell_4
```

Let us say that we want to verify the LTL property $\varphi = F G x$, which, informally, can be read as “for every trace of the system, $x$ will eventually become true and stay true.” For this transition system, the LTL property $\varphi$ holds. This is because each *individual* trace eventually gets to a point where $x = \text{true}$ forever more. For example, if a trace never leaves the loop at $\ell_2$, then the property is valid because $x = \text{true}$ before entering the loop. For the traces that do reach $\ell_3$, $x = \text{true}$ will hold once $pc = \ell_4$, and then remain true.

As we discussed in Chapter 2 $\forall$CTL can be used to approximate LTL by letting $\Phi = \eta(\varphi)$:

$M \models \forall \varphi$ $\Rightarrow$ $M \models F G x$.

Unfortunately, this $\forall$CTL formula does not hold of the transition system and we obtain the following $\forall$CTL counterexample:
The above $\forall$CTL counterexample is an infinite tree of behaviors that arises from the choice as to when control exits the first loop. The $\forall$CTL property $AF\ AG\ x$ does not hold because there is no set of states $S$ that you are guaranteed to eventually reach from the initial state such that for every $t \in S$, all reachable states from $t$ are ones in which $x = true$. No matter how far you travel down the spine of $\ell_2$ states, there are always more behaviors that will lead to a state in which $x = false$.

Yet the LTL property $F\ G\ x$ holds because if we consider each individual execution (where we already know all nondeterministic choices made) we see that execution eventually reaches a point where $x = true$ forever more. We have a dilemma. $\forall$CTL methods provide an efficient way to prove LTL properties via over approximation. However, in cases such as the above, nondeterminism in the transition system (here, the choice as to when control exits from the first loop) foils our ability to prove the LTL property.

### 5.2 Characterizing nondeterministic choice

In Example 5.1, when we are at a state $[true]_{\ell_2}$ there is a nondeterministic choice to either go to state $[true]_{\ell_2}$ or to state $[false]_{\ell_3}$. This nondeterministic choice is the reason that our method of proving LTL via an $\forall$CTL verification algorithm fails. When we are in the loop at $\ell_2$ we cannot know if we will eventually leave the loop or not. We struggle when trying to decide if a state at $\ell_2$ is the point at which $x$ will be globally true, as it is only after considering a full trace that we would know (i.e. in this case we need to be looking at sets of traces, not sets of states). This is the fundamental difference between the LTL property $\varphi$ and the $\forall$CTL property $\Phi$. The LTL view of the world is in terms of traces and when we consider the transition system on a per-trace basis, the choice has already been made. In $\forall$CTL however, we have reached an individual state and this choice remains undecided.

We begin by characterizing such nondeterministic choices with decision predicates:
**Definition 5.1** (Decision predicates). For a transition relation $R$, suppose $(s, t) \in R$ and $(s, t') \in R$. A decision predicate is a pair of formulae $(a, b)$ such that

$$s \in [a]^S \quad \text{and} \quad t \in [b]^S \quad \text{and} \quad t' \in [-b]^S.$$  

Informally, for a decision predicate pair $(a, b)$, the formula $a$ defines the decision predicate's presupposition (i.e. when the decision is made), and $b$ characterizes the binary choice made when this presupposition holds. We can represent this visually as:

![Diagram](image)

A decision predicate allows us to partition the transition relation, distinguishing certain types of transitions. For a given $R$ and decision predicate pair $(a, b)$, let

$$r^b_a = \{(s, t) \in R \mid s \in [a]^S \land t \in [b]^S\},$$

$$r^{-b}_a = \{(s, t) \in R \mid s \in [a]^S \land t \in [-b]^S\}.$$  

Notice that $r^b_a \cap r^{-b}_a = \emptyset$ and that $r^b_a \cup r^{-b}_a = \{(s, t) \in R \mid s \in [a]^S\}$. For the above example, we can use the following decision predicate pair to characterize the nondeterministic choice:

$$a \equiv \text{“pc} = \ell_2'' \quad \text{and} \quad b \equiv \text{“pc} = \ell_3''$$

This pair distinguishes a transition from pc = $\ell_2$ to pc = $\ell_3$ (where b holds in the second state) from a transition from pc = $\ell_2$ to pc = $\ell_2$ (where $-b$ holds in the second state).

### 5.3 Symbolic partial determinization

When we obtain a counterexample to an $\forall$CTL approximation $\eta(\varphi)$ of an LTL formula $\varphi$, the counterexample does not necessarily mean that the LTL property does not hold. There may be branching in the $\forall$CTL counterexample that arises from certain nondeterminism in the transition system.

Let us say that we have a vector of decision predicate pairs $\Omega$ that characterize this nondeterminism in the $\forall$CTL counterexample. In this section, we will describe a symbolic partial determinization on our transition system $M$. We will obtain a new transition system $M^\Omega$ that is trace equivalent to $M$ but has been determined with respect to $\Omega$.

In this new transition system $M^\Omega$, each time execution reaches a $a_i$-state for some decision predicate pair $(a_i, b_i) \in \Omega$ a special internal prophecy variable $\rho_i$ (that has been previously nondeterministically intialized) will instruct the transition system as to what decision it should make. Intuitively, if $\rho_i$ is true a transition will be made to a $b_i$-state. If $\rho_i$ is false a transition will be made to a $-b_i$-state. (Note that in the formalization below, $\rho$ values are taken from the natural numbers rather than booleans.)
Symbolic determinization. Formally, we begin by describing \textsc{Determinize}, which takes a transition system \( M \) and a vector of decision predicates \( \Omega \) and returns a partially determinized transition system \( M^\Omega \). We will later show that \( M^\Omega \) is trace equivalent to the original \( M \).

We will be constructing new systems by adding variables to a transition system and equating the new system to the original version. Thus, it is convenient to build in a notion of internal and external state elements. We assume that \( S = S^{ex} \times S^{in} \) (i.e. states consist of an external (visible) component and an internal component). We refer to an individual state as \( \langle s, \rho \rangle \in S \) and when a transition system has no internal components, we omit the \( \langle \rangle \) brackets (e.g. our treatment of states in the previous chapters). We also augment the definition of a transition system \( M \) to include a projection function:

\[
\text{proj} \equiv \lambda \langle s, \rho \rangle. s
\]

In Definition 5.2 below, we describe \textsc{Determinize} which consumes a transition system \( M \) and returns a new transition system \( M^\Omega \) that has been partially determinized with respect to a vector of decision predicates \( \Omega \). That is,

\[
M^\Omega = \textsc{Determinize}(M, \Omega)
\]

The new transition system \( M^\Omega \) includes prophecy variables denoted \( \rho_i \). These correspond to the predicate pairs \( (a_i, b_i) \) in the vector \( \Omega \).

Definition 5.2 (Symbolic determinization). For all \( M = (S, R, I) \) and a finite vector of decision predicates \( \Omega \), let \( \textsc{Determinize}((S, R, I), \Omega) = (S^\Omega, R^\Omega, I^\Omega) \) where

\[
\begin{align*}
S^\Omega &= S \times \overline{\mathbb{N}_1} \\
I^\Omega &= I \times \overline{\mathbb{N}_1} \\
R^\Omega &= \{((s, \rho), (s', \rho')) | (s, s') \in R \land \forall (a_i, b_i) \in \Omega. \\
&\quad \left[ (a_i(s) \land \rho_i = 1) \Rightarrow b_i(s') \land \rho_i' = 1 \right] \quad (5.2) \\
&\quad \land \left[ (a_i(s) \land \rho_i > 0) \Rightarrow b_i(s') \land \rho_i' = \rho_i - 1 \right] \quad (5.3) \\
&\quad \land \left[ (a_i(s) \land \rho_i = 0) \Rightarrow \neg b_i(s') \land \rho_i' \in \mathbb{N}_1 \right] \quad (5.4) \\
&\quad \land \left[ \neg a_i(s) \Rightarrow \rho_i' = \rho_i \right] \quad (5.5)
\end{align*}
\]

where \( \mathbb{N}_1 \equiv \mathbb{N} \cup \{1\} \) and \( \overline{\mathbb{N}_1} \equiv (\mathbb{N}_1)^\perp \).

This new transition system \( M^\Omega \) is similar to \( M \) but has been augmented with some internal state and a vector of prophecy variables \( \rho_i \). In accordance with \( I^\Omega \) these prophecy variables are free to be a natural number or \( \perp \) in the initial state. The choice of initial values (and the choice in Eqn. 5.4) is the driving force behind determinization.

Transitions in \( R^\Omega \) are made in accordance with \( R \), but constrained by the values of \( \rho \) when states are reached that match a decision predicate \( (a_i, b_i) \) in \( \Omega \). Specifically, when a state is reached where \( a_i \) holds and the prophecy variable \( \rho_i = 1 \), then \( b_i \) must hold in the next
state and \( \rho_i \) is unchanged (Eqn. 5.2). This rule corresponds to behaviors where a \( s \in [a_i]S \) state is visited infinitely often. Alternatively, if \( \rho_i > 0 \) (Eqn. 5.3) then \( b_i \) must also hold in the next state, except that \( \rho_i \) is decremented. When \( \rho_i \) reaches zero, then \( \neg b_i \) must hold in the next state and \( \rho_i \) is free to take a new value from \( \mathbb{N}_1 \), starting the process again (Eqn. 5.4). Finally, when \( a_i \) doesn’t hold of a particular state, \( \rho_i \) is unchanged (Eqn. 5.5).

The prophecy variables introduced here trade nondeterminism in the transition relation \( R \) for a larger, nondeterministic state space. The state space nondeterminism is either determined by the initial choice of values for \( \rho \) given by \( P^\Omega \), or else later in a trace (Eqn. 5.4) by choosing new nondeterministic values for \( \rho \). This lazy selection of nondeterministic values means that \( M^\Omega \) need not consist of infinitely many prophecy variables for each predicate pair. This formulation restricts us to treat transition systems with only countable nondeterminism. One could conceive of more powerful forms of nondeterminism, but we intend to use this technique in the context of programs for which countable nondeterminism is sufficient.

**Example revisited.** Returning to Example 5.1, let \( \Omega = \{ (pc = \ell_2, pc = \ell_2) \} \). DETERMINEZ\( (M, \Omega) \) then returns \( M^\Omega = (S^\Omega, R^\Omega, I^\Omega) \) where

\[
S^\Omega = S \times \mathbb{N}_1 \quad \text{denoted } \{ [\vec{x} , \rho] \},
\]

\[
I^\Omega = \{ [\vec{x} , \rho] \mid \rho \in \mathbb{N}_1 \},
\]

\[
R^\Omega = \{ [\vec{x} , \rho] , [\vec{x} , \rho] \mid \text{true} \}
\]

\[
\cup \{ [\vec{x} , \rho] , [\vec{x} , \rho - 1] \mid \rho \neq 0 \}
\]

\[
\cup \{ [\vec{x} , \rho] , [\vec{x} , \rho] \mid \rho = 0 \wedge \rho' \in \mathbb{N}_1 \}
\]

\[
\cup \{ [\vec{x} , \rho] , [\vec{x} , \rho] \mid \text{true} \}
\]

\[
\cup \{ [\vec{x} , \rho] , [\vec{x} , \rho] \mid \text{true} \}.
\]

Note that subtraction over the bottom element is the identity operation: \( -1 - 1 = 1 \). \( M^\Omega \) can be thought of as a guarded transition system:

\[
\{ \rho \neq 0 \wedge x = \text{true} \}
\]

\[
\rho := \rho - 1
\]

In this example the new prophecy variable \( \rho \) predicts the outcomes of the decision predicate \( (pc = 2, pc = 2) \). We initialize \( \rho \) to be an element of \( \mathbb{N}_1 \). For every given trace
of the system, the concrete number chosen at the command \( \rho := * \) (indicating that \( \rho' \in \mathbb{N} \)) predicts the number of instances of the \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_2}) \) transition before we see a \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transition. The choice of \( \rho = 0 \) represents the case where the execution will never see a \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transition (i.e. non-termination). Whenever the program makes a \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_2}) \) transition it knows that \( \rho \neq 0 \), because the prophecy made previously does not allow it. The program also decrements \( \rho \) whenever we see a \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transition, for we know that (if we are going to see it at all) we are one step closer to seeing \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \). If and when a \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transition finally occurs, we know that \( \rho = 0 \). The program then predicts how many \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transitions will be visited the next time around until seeing another \( ([\text{true}]_{\ell_2}, [\text{true}]_{\ell_3}) \) transition (which will never occur in this example). Because the old prediction is not needed again, we can re-use the same variable \( \rho \) for the new prophecy.

With the prophecy variable \( \rho \) in place, the computation tree is as follows:

Notice that the transition relation has been determined (in this example it is fully determined, though in most cases \textsc{Determinize} will only partially determinize, only affecting some subset of \( R \)). There is now a set of states \( X \) where \( \text{AG} \; x \) holds:

\[
X \equiv \{ ([\text{true}]_{pc}, \rho) \mid (\rho = 0 \land pc = \ell_1) \lor pc = \ell_4 \}
\]

Furthermore we can prove that \( X \) is eventually reached. So we can now use \( \forall \text{CTL} \) where it previously failed. Applying an \( \forall \text{CTL} \) verifier to \( M^\Omega \) will produce no counterexample to \( \text{AF AG} \; x \). In the next sections we will show that this means that \( \varphi \) holds. Specifically, we will show that:

\[
M^\Omega \models_e \eta(\varphi) \Rightarrow M \models_e \varphi
\]

To this end, we will first show (Section 5.4) that the determinized \( M^\Omega \) is trace equivalent to \( M \): \( M^\Omega \equiv M \), following the criteria of prophecy variables from Abadi and Lamport [AL91]. We will then show that the above lemma holds (Section 5.5).
5.4 Trace equivalence

In this section we will prove that our determinization preserves trace equivalence with the original machine. We will continue to work with the coinductive definition of a trace $\pi$. Note the standard coinductive definition of equivalence over traces:

$$\text{EqSt}\ (\pi_1, \pi_2) \equiv (\text{EqSt}\ (\text{hd}\ \pi_1, \text{hd}\ \pi_2)) \land (\text{EqSt}\ (\text{tl}\ \pi_1, \text{tl}\ \pi_2))$$

**Definition 5.3 (Trace projection).** We lift projection to traces:

$$\pi |_{\text{ext}} \equiv (\text{hd}\ \pi |_{\text{ext}} : (\text{tl}\ \pi |_{\text{ext}})$$

**Definition 5.4 (External transition system trace).** For a transition system $M$, we say that a stream of external states $\pi = (s_0, s_1, \ldots)$ is a trace of $M$ as follows:

$$\text{isExtTrace}\ M\ \pi \equiv \exists \pi'. \text{isTrace}\ M\ \pi' \land \text{EqSt}\ (\pi' |_{\text{ext}})\ \pi.$$ 

Informally this means that $\pi$ is a trace of (external states of) the transition system provided that there is some corresponding trace of (internal and external states of) the transition system.

**Definition 5.5 (Trace equivalence).** For every $M_1 = (S_1, R_1, I_1)$ and $M_2 = (S_2, R_2, I_2)$,

$$M_1 \cong M_2 \equiv (\forall \pi. \text{isTrace}\ M_1\ \pi \Rightarrow \text{isExtTrace}\ M_2\ \pi |_{\text{ext}}) \land (\forall \pi. \text{isTrace}\ M_2\ \pi \Rightarrow \text{isExtTrace}\ M_1\ \pi |_{\text{ext}})$$

The following theorem states that the determinization procedure DETERMINIZE produces a new transition system that is trace equivalent to the original transition system.

**Theorem 5.1.** For all $\Omega$, $M^\Omega \cong M$.

**Proof.** The theorem holds if each of the conditions $P_1$, $P_2$, $P_3$ and $P_4$ and $P_B$ described below are met. These conditions are a variation of Proposition 5 from Abadi and Lamport [AL91]. Conditions $P_1$, $P_2$, $P_3$ and $P_4$ directly match Abadi and Lamport’s conditions. We omit Condition $P_5$ as it involves liveness restrictions on the behavior of transition systems and we assume that our transition systems have no liveness restrictions. We loosen the restriction of Abadi and Lamport’s $P_6$ with $P_B$ (detailed below), as our prophecy variables do not respect the condition of finite nondeterminism. The new condition $P_B$ is in fact a consequence of $P_6$: in the second part of the proof, Abadi and Lamport show that all the behaviors of $M$ are contained within $M^\Omega$ (note that regardless of superscript, $P = M$ because $L = \text{true}$). Part 2.1 defines a directed graph, and then introduces Claim 2.1, which is not true in our setting. However, Claim 2.1 is only used
in conjunction with Claim 2.2 and König’s Lemma in order to prove Claim 2.3. In our setting we have simply included Claim 2.3 as condition PB.

We now describe why each condition holds:

(P1) \( S^\Omega \subseteq S \times S^P \) for some \( S^P \). √

(P2) \( I^\Omega \equiv \Pi_p^{-1}(I) \) where \( \Pi_p^{-1} \) maps \( S \times S^\Omega \) onto \( S \). √

(P3) If \( ((s, p), (s', p')) \in R^\Omega \) then \( (s, s') \in R \) or \( s = s' \). This holds by construction of \( R^\Omega \) from \( R \). √

(P4) If \( (s, s') \in R \) and \( (s', p') \in S^\Omega \) then there exists \( p \in S^\Omega \) such that \( ((s, p), (s', p')) \in R^\Omega \). Again, this holds by construction of \( R^\Omega \) from \( R \), case splitting on the value of \( p' \) and quantifying over \( i \in \Omega \). √

(PB) For every \( (s_0, s_1, ...) \in \Pi(M) \) there exists \( (p_0, p_1, ...) \) such that \( ((s_0, p_0), (s_1, p_1), ...) \in \Pi(M^\Omega) \). Quantifying over each \( i \leq |\Omega| \), consider all of the (possibly infinitely many) transitions \( (s_j, s_{j+1}) \) such that \( a_i(s_j) \) holds. Now for each transition \( b_i(s_{j+1}) \) may or may not hold. This can be modeled by:

\[
(\exists m. b_i^m \cdot b_i)_{\infty}^* (b_i^\infty)
\]

i.e. a head \( (\exists m. b_i^m \cdot b_i)_{\infty}^* \) consisting of repeated instances of finitely many \( b_i \)-states and a single \( \neg b_i \)-state, and a tail consisting of infinitely many \( b_i \)-states. So we can choose \( \rho_i \) accordingly, setting \( \rho_i = m \) in each (potentially zero or infinitely many) instances of the head, and setting \( \rho_i = \bot \) in the tail. √

\[\square\]

Example 5.6. (Nondeterministic Choice) Consider the following transition system:

\[
S = \begin{bmatrix} N \end{bmatrix} \text{ denoted } [\bar{x}]
\]

\[
I = \begin{bmatrix} 0 \end{bmatrix}
\]

\[
R = \{ ([0], [1]), ([0], [0]), ([1], [1]), ([0], [0]) \}
\]

In this transition relation there is nondeterminism in the first transition. We can determine this with the predicates \( a = (x = 0 \land y = 0) \) and \( b = (x = 1) \). With these predicates we can construct the corresponding \( M^\Omega \).

\[
S^\Omega = \begin{bmatrix} N \end{bmatrix} \times N_1 \text{ denoted } ([\bar{x}], \rho)
\]

\[
I^\Omega = ([0], N_1)
\]

\[
R^\Omega = \{ ([0], 1), ([0], 0), ([0], 0), ([1], 0), ([0], N_1), ([1], N_1), ([0], N_1), ([1], N_1) \}
\]
The first two transitions have now been determinized: from the initial state, depending on the initial choice of \( \rho \), either \((x = 1)\) or \(\neg(x = 1)\) will hold in the next state. In this example, since the nondeterministic transition only happens once, the (external) behaviors when \( \rho > 1 \) or \( \rho = \bot \) in the initial state are all equivalent to \( \rho = 1 \) in the initial state so, for presentation purposes, we have omitted them. The additional behaviors will be used in the next example.

Example 5.7. (Termination) Consider the following infinite-state system which we represent symbolically

\[
S = N \text{ denoted } x \\
I = N \\
R = \llbracket (x > 0 \land x' = x + 1) \lor (x > 0 \land x' = 0) \lor (x = 0 \land x' = 0) \rrbracket^R
\]

In this transition relation, when \( x > 0 \) initially, there is nondeterminism in how many times the first transition is chosen before the second transition is chosen. We can determinize this with the predicates \( a = (x > 0) \) and \( b = (x > 0) \), constructing the corresponding \( M^\Omega \) as follows:

\[
S^\Omega = N \times N \times \bot \\
I^\Omega = N \times N \times \bot \\
R^\Omega = \llbracket (x > 0 \land \rho = 1 \land x' = x + 1 \land \rho' = 1) \lor \\
(x > 0 \land \rho > 0 \land x' = x + 1 \land \rho' = \rho - 1) \lor \\
(x > 0 \land \rho = 0 \land x' = 0 \land \rho' \in N) \lor \\
(x = 0 \land \rho \in N \land x' = 0 \land \rho' \in N) \rrbracket^R
\]

In \( M^\Omega \) the first choice of how many times a transition from \( [x > 0 \land x' = x + 1]^R \) is taken is given by the choice of an initial value for \( \rho \). Any finite number of iterations corresponds to an arbitrarily chosen numeric value of \( \rho \). The case where the transition is taken infinitely many times corresponds to the initial choice of \( \bot \) for \( \rho \).

5.5 Proving LTL with \( \forall \text{CTL} \) and determinization

In this section we will show that, given an LTL formula \( \varphi \) and a decision predicate vector \( \Omega \), a proof that \( \eta(\varphi) \) holds of a transition system \( M^\Omega \) implies that \( M \models L \varphi \).

Lemma 2.6 shows that one can prove LTL properties with an \( \forall \text{CTL} \) verifier and an unmodified transition relation. We now extend this to show that one can prove (perhaps even more) LTL properties with \( \forall \text{CTL} \) and a predicate-determinized transition system. We begin with several lemmas before proving our main result Theorem 5.7.

Lemma 5.2 (Tail equality). For all \( \pi_1, \pi_2 \) such that \( \text{EqSt } \pi_1 \pi_2 \), \( \text{EqSt } (tl \pi_1)(tl \pi_2) \)

Lemma 5.3 (Rewrite tail). For all \( \pi \), \( \text{EqSt}(\text{tl}\,(\pi)|_{\text{ext}}
(\text{tl}\,(\pi)|_{\text{ext}})\))

Proof. (Coq: Dpred.tl_outside) Coinduction.

Lemma 5.4 (nth tail equality). For all \( \pi_1, \pi_2 \)
\[
\text{EqSt}\,\pi_1|_{\text{ext}} \pi_2|_{\text{ext}} \Rightarrow \forall n. \text{EqSt}(\pi_1)^n|_{\text{ext}} \pi_2)^n|_{\text{ext}}
\]

Proof. (Coq: Dpred.tl_eq_n) Induction and Lemma 5.2 and fact that the tails of two
\text{EqSt} paths are also \text{EqSt}.

Lemma 5.5. For all \( M_1, M_2, \pi_1, \pi_2 \) and LTL property \( \varphi \),
\[
\text{isExtTrace}\,M_1\,\pi_1 \land \text{isExtTrace}\,M_2\,\pi_2 \land \text{EqSt}\,(\pi_1)|_{\text{ext}}(\pi_2)|_{\text{ext}}
\Rightarrow
\]
\[
(\pi_1 \models \varphi \Rightarrow \pi_2 \models \varphi)
\]

Proof. (Coq: Dpred.equiv_ltl_tr) By induction on \( \varphi \), using Lemma 5.4 in each case.

Lemma 5.6 (LTL-equivalent transition systems). For all \( M_1, M_2 \) such that \( M_1 \cong M_2 \),
\[
\forall \varphi. M_1 \models \varphi \Rightarrow M_2 \models \varphi
\]

Proof. (Coq: Dpred.equiv_ltl) Choose a trace \( \pi \) of \( M_2 \) (which has internal and external
components). Show that \text{isExtTrace} \((\pi)|_{\text{ext}}\) \( M_1 \) (Trivial). Now get the corresponding
internal+external trace \( \pi' \) of \( M_1 \). Finally, apply Lemma 5.5.

Theorem 5.7 (\forall CTL Approximation with Determinization). For a transition system \( M \),
LTL property \( \varphi \) and predicates \( \Omega \),
\[
M^\Omega \models \eta(\varphi) \Rightarrow M \models \varphi
\]

Proof. (Coq: Dpred.four_two) Lemma 2.6 says that \( M \models \eta(\varphi) \Rightarrow M \models \varphi \). Moreover, by
Theorem 5.1 we know that \( M^\Omega \cong M \). Thus we can use Lemma 5.6 to obtain that the
theorem holds.
5.6 Related work

Others have considered this trade-off between linear-time specifications and efficient branching-time verification procedures. For example, Cadence SMV [cad] reduces LTL to CTL using additional fairness constraints [BC+92, CGH97]. This technique still relies heavily on reasoning about fairness. This is a sensible engineering choice for finite-state systems for the reasons discussed above, but not for infinite-state systems. Schneider describes a method of translating an LTL formula into a semantically equivalent CTL formula [Sch98]. However, this leads to an exponential blowup in the size of the CTL formula, and requires a modification to the model checking algorithm. Maidl identifies the subset of ∀CTL (called ∀CTL\text{det}) which is expressible in LTL. Consequentially, for such formulae, an ∀CTL prover can be used [Mai00].

It is well known that determinization addresses the subtle semantic distinctions between linear-time and branching-time logics [Saf88]. However, for infinite-state systems, open questions still remain if we hope to develop a practical determinization-based strategy: a) what to determinize, since complete determinization does not lead to a viable automatic tool for infinite-state systems, and b) how to determinize in a way that facilitates the application of current formal verification tools. We address these two questions in this dissertation.

Our work uses prophecy variables [AL91]. We are of course not the first to use them in applications related to the one addressed here. Prophecy variables have been used for many years to resolve nondeterminism in proofs, including some recent work [HAN08, STQ10].
We now specialize our algorithms in previous chapters to target *programs* (rather than arbitrary transition systems), and do so in such a way that existing program analysis tools can be used to search for the proof that a temporal logic property holds of the program. To align our work more closely with programs, we will define a simple *while*-like language. Our implementation operates on a subset of C programs, similar to this *while*-like language.

Our specialization, when given a program $P$ and a temporal logic property $\varphi$, performs a source-to-source translation, generating a new procedural linear arithmetic program $P'$ that encodes the search for the proof that $\varphi$ holds of $P$. In this way, we have produced a *program analysis task* and modern interprocedural program analysis tools (*e.g.* [BR01, BHJM07, BCC*03, CDOY09, RHS95]) together with techniques for finding termination arguments (*e.g.* [BCC*07, BMS05, CPR06]) can then be immediately used to reason about the validity of the property. When these tools are applied to the procedural encoding, they are effectively performing temporal reasoning on the original program.

Our restriction to programs means that we can take advantage of the fact that the transition relation has a finite representation, and perform a variety of optimizations that lead to a tractable verification tool.

As a second specialization, we will work with programs that are defined over integer variables and assignments consisting of linear arithmetic expressions. While this is certainly a strong restriction on the domain, the domain is nonetheless useful. Moreover, in addition to modeling boolean and numeric variables, there are techniques [MBCC07] for reducing heap manipulating programs to numeric programs, while preserving the termination and other liveness behavior of the original program [MTL10]. Finally, rank function synthesis for this domain is decidable [PR04a].

Finally, we adapt a known algorithm [CPR06] to obtain an iterative method of discovering a sufficient set of termination arguments by examining spurious $\forall$CTL counterexamples and synthesizing linear arithmetic rank functions. In our implementation, we obtain these
counterexamples from existing interprocedural program analysis tools \cite{BR01, CDOY09, RHS95} that report stack-based traces for assertion failures.

**Limitation.** A source of incompleteness of our implementation comes from our reliance on lassos (representing a counterexample as a finite stem-path and an [infinitely repeated] finite loop-path). Some non-terminating programs have only well-founded lassos, meaning that in these cases our refinement algorithm will fail to find useful refinements. The same problem occurs elsewhere \cite{CPR06}, but in industrial examples these programs are rare.

### 6.1 Simple Programming Language

We begin by formally defining an imperative C-style language. The operational semantics of this language can be given in the form of a transition system and then our results can be applied.

**Definition 6.1** (Simple programming language, SPL \cite{MP95}). For a set of basic commands $c$ and boolean expressions $b$,

$$
P ::= P ; P \quad \text{Sequential composition} \\
| P + P \quad \text{Nondeterministic choice} \\
| P^* \quad \text{Looping} \\
| \text{assume}(b) \quad \text{Assume} \\
| c \quad \text{Basic command} \\
| \text{skip} \quad \text{Skip}
$$

SPL is parameterized by the set of commands and expressions (and the respective semantics $\llbracket c \rrbracket$ and $\llbracket b \rrbracket$). In general, programs could involve a memory and commands could be operations on that memory. This would allow us to, for example, model heap manipulating programs. For this dissertation, however, we will assume we are working with states that map (stack) variables $\text{Vars}$ to integers (and expressions from the domain of linear arithmetic). Hence the following definitions:

$c ::= x := e$

e ::= z \in \mathbb{Z} \mid v \in \text{Vars} \mid (e + e) \mid (e - e) \mid (e \times e) \mid (e \div e) \mid *_e$

$b ::= \text{true} \mid \neg b \mid *_b \mid (b \&\& b) \mid (e == e) \mid (e > e) \mid (e \geq e)$

The notations $*_b$ and $*_e$ indicate nondeterministic choice. We typically we drop the $e$ and $b$ subscripts on $*$, as the type is given by the context. Standard programming language idioms can be derived as follows:

\[
\begin{align*}
\text{while } (b) \{ P \} & \equiv (\text{assume}(b) ; P)^* ; \text{assume}(\neg b) \\
\text{if } (b) \{ P_1 \} \text{ else } \{ P_2 \} & \equiv (\text{assume}(b) ; P_1) + (\text{assume}(\neg b) ; P_2) \\
\text{if } (b) \{ P_1 \} & \equiv (\text{assume}(b) ; P_1) + (\text{assume}(\neg b) ; \text{skip})
\end{align*}
\]
Figure 6.1: Operational semantics of SPL, which is parameterized by commands \( c \) and boolean expressions \( b \), as well as the corresponding semantics.

Standard expressions (e.g. \( \text{false}, <, \leq, \neq, || \)) can also be derived.

The small-step operational semantics for SPL are given in Figure 6.1. Configurations consist of the program text \( P \) and the current state \( s \) from the set of states \( S_P \) (in a given domain \( D \)) and the initial states \( I_P \). We will, for the most part, assume the state \( s \) is a mapping from (typed) variables \( \text{Vars} \) to values.

A program text is finite. We assume a special variable denoted \( \text{pc} \) which maps each subcommand \( P \) to a unique element in a finite domain \( L \). In Appendix C we give a modified version of the operational semantics that explicitly updates the value of \( \text{pc} \) accordingly in each step.

The transition system for program text \( P \) and set of variables \( \text{Vars} \), \( M_P = (S_P, R_P, I_P) \) (usually written simply \( M \)) requires that we define the transition relation:

\[
R_P \equiv \{ (s, t) \mid s \in S \land \exists P' \in \text{sub}(P). \exists P''. P', s \rightarrow P'', t \}
\]

\( \text{sub}(P) \) is defined recursively over the grammar of \( P \) and returns all subterms of \( P \). The initial states \( I_P \) are those in which \( \text{pc} = \ell_0 \) and the initial conditions hold on program variables.

**Example 6.1** (Acquire/release). The following SPL program is a typical lock acquire/release-style program, where we are interested in proving the \( \forall \)CTL property \( \Phi = AG[(x = 1) \Rightarrow AF(x = 0)] \). Assume that initially \( x = 0 \).
6.2 SPL verification as a program analysis task

We now describe how our specialized encoding allows modern program analysis tools to perform what is necessary to validate temporal logic properties of programs. For an SPL program $P$ and $\forall$CTL property $\Phi$, our encoding generates a finite number of SPL procedures, one for each subformula:

$$\mathcal{E}(P, M, \Phi) \equiv \bigcup_{(\kappa, \psi) \in \text{sub}(\Phi)} \{ \text{enc}^\kappa_\psi : s \rightarrow B \}$$

For Example 6.2, the encoding consists of the following set of procedures:

$$\{ \text{enc}^{AG}_{(x=1) \lor AF(x=0)}, \text{enc}^{Lr}_{(x=1) \lor AF(x=0)}, \text{enc}^{Ll}_{x=1}, \text{enc}^{RLr}_{AF(x=0)}, \text{enc}^{RLl}_{x=0} \}$$

The per-subformula encoding is given in Figure 6.2 and is a specialization of the encoding given in Chapter 4. Rather than formulating $\forall$CTL verification as a behavioral property of a guarded transition system, here we formulate it as a safety property of an SPL program in tandem with termination arguments. Notice that this encoding is parameterized by a finite set of rank functions $M$. These procedures encode the search for the proof that $\Phi$ holds of $P$: if a sufficient $M$ is found such that $\text{assert} (\text{enc}^\kappa_\psi(s))$ can be proved safe for all $s \in I$, then $\Phi$ holds of $P$ (i.e. $P \models_\square \Phi$).

6.2.1 Specialization of Chapter 4

The encoding in Figure 6.2 is largely equivalent to our formulation in Chapter 4. Both explore the $S \times \text{sub}(\Phi)$ state space in a depth-first manner. Our procedural encoding here is analogous to the guarded transition system encoding $G^M_{M, \Phi}$ in Chapter 4 as follows:

- A procedure $\text{enc}^\kappa_\psi$ is used, rather than per-subformula nodes $(\text{en}, \kappa, \psi)$ and $(\text{en}, \kappa, \psi)$.
  
  The root procedure $\text{enc}^\kappa_\psi$ is analogous to root node $(\text{en}, \epsilon, \psi)$. 

\[ \mathcal{E}(P, \mathcal{M}, \Phi) \equiv \bigcup_{(\kappa, \psi) \in \text{sub}(\Phi)} \{ \text{enc}_{\psi}^\kappa : s \rightarrow B \} \]

where

\[
\begin{align*}
\text{bool enc}_{\psi \wedge \psi'}^\kappa (\text{state } s) & \{
\begin{array}{l}
\text{if (*) return enc}_{\psi}^{L\kappa} (s);
\text{else return enc}_{\psi'}^{R\kappa} (s);
\end{array}
\} \\

\text{bool enc}_{\psi \vee \psi'}^\kappa (\text{state } s) & \{
\begin{array}{l}
\text{if (enc}_{\psi}^{L\kappa} (s)) \text{ return true;}
\text{else return enc}_{\psi'}^{R\kappa} (s);
\end{array}
\}
\end{align*}
\]

\[
\begin{align*}
\text{bool enc}_{\psi}^\alpha (\text{state } s) & \{ 
\begin{array}{l}
\text{return } \alpha (s); 
\end{array}
\}
\end{align*}
\]

\[
\begin{align*}
\text{bool enc}_{\psi}^{A[\psi \wedge \psi']}^\kappa (\text{state } s) & \{
\begin{array}{l}
P[c / \{ 
\begin{array}{l}
\text{if (*) return true; }
\text{if (enc}_{\psi}^{L\kappa} (s)) \text{ return enc}_{\psi'}^{R\kappa} (s); 
\end{array}
\} ]; c]
\end{array}
\}
\end{align*}
\]

\[
\begin{align*}
\text{bool enc}_{\psi}^{A\Phi}^\kappa (\text{state } s) & \{
\begin{array}{l}
\text{bool dup = false; state 's; }
\text{if (enc}_{\psi}^{L\kappa} (s)) \text{ return true; }
\text{if (dup && (\exists f \in \mathcal{M}. f(s) < f('s'))) return false; }
\text{if (\neg dup && *) \{ dup := true; 's := s; \}}
\end{array}
\}
\end{align*}
\]

Figure 6.2: Encoding \( \forall \text{CTL} \) verification as a finite set of SPL procedures.

- The current state is passed on the stack rather than stored in a per-subformula variable \( \sigma_{\psi}^\kappa \).
- A procedure \( \text{enc}_{\psi}^\kappa \) has numerous executions, rather than a node \( (\text{en}, \kappa, \psi) \) that has numerous runs.
- A procedure \( \text{enc}_{\psi}^\kappa (s) \) returns false whenever \( \psi \) does not hold of \( s \), whereas a \( (\text{en}, \kappa, \psi) \)-run will fault.
- If \( \text{enc}_{\psi}^\kappa \) can be proved to never return false, then it must be the case that the overall property \( \Phi \) holds of the initial state \( s \). A proof that \( \mathcal{G}_M^M \Phi \) does not fault implies that \( \Phi \) holds of the initial state \( s \).

The corresponding soundness theorem is as follows:

**Theorem 6.1** (\( \mathcal{E} \) soundness). For an SPL program \( P \) and \( \forall \text{CTL} \) property \( \Phi \),

\[ \exists \text{ finite } \mathcal{M}. \mathcal{E}(P, \mathcal{M}, \Phi) \text{ cannot return false } \Rightarrow P \models_\mathcal{E} \Phi \]

where \( \mathcal{M} \) is, as described earlier, a finite set of ranking functions.
Proof. Specialization of Theorem 4.3. \( \square \)

We abuse notation slightly here, using \( E(P, M, \Phi) \) to mean “\( \forall s \in I. \text{enc}_\Phi(s) \)”.

### 6.2.2 Encoding for program analysis tools

Program analysis tools for safety are effective at discovering a counterexample execution that leads to an assertion violation. We have designed this specialized encoding, which is similar to the representation in Chapter 4, with this in mind. To this end, our design of \( E \) maintains the following invariant:

\[
INV_1 : \forall s, \psi, \kappa. R, s \not\models \psi \text{ implies } \text{enc}_\kappa(s) \text{ can return false}
\]

(Specialization of Lemma 4.1.)

For example, consider the \( \text{enc}_\kappa^\psi \text{ or} \text{enc}_\kappa^{\psi'} \) case from the definition of \( E \). Imagine that \( \psi \equiv x \neq 1 \), and \( \psi' \equiv \text{AG}(x = 0) \). In this case we want to know that one of the subformulae (\( i.e. x \neq 1 \) or \( \text{AG}(x = 0) \)) holds. A procedure call \( \text{enc}_\kappa^\text{L}_{x \neq 1}(s) \) is made to explore whether \( x \neq 1 \) as well as a separate procedure call \( \text{enc}_\kappa^\text{R}_{\text{AG}(x = 0)}(s) \) with the same current state \( s \) to explore \( \text{AG}(x = 0) \). During a symbolic execution of this program, all executions will be considered in a search for a way to cause the program to fail. If it is possible for both procedure calls to return false (\( i.e. \) they satisfy \( INV_1 \)), then there will be an execution in which \( \text{enc}_\kappa^\psi(s) \) can return false (also satisfying \( INV_1 \)). A standard program analysis tool (\( e.g. \) SLAM [BBC+06] or BLAST [BHJM07]) will find this case. By maintaining this invariant in each procedure, a proof that the outermost procedure \( \text{enc}_\Phi \) cannot return false implies that the property \( \Phi \) holds of the program \( P \). There is also a corresponding invariant \( INV_2 \) to Lemma 3.2.

\[
INV_2 : \forall s, \psi, \kappa. \text{enc}_\psi(s) \text{ can return true}
\]

(Specialization of Lemma 3.2.)

### 6.2.3 Finite representation of transition relation

**Sequencing.** Following \( \land, \alpha, \text{and} \lor \) above, one could give the following naïve implementation of the \( \text{AW} \) rule from Chapter 4 as follows:

```cpp
bool enc^c_{\lor \land}(state s) {
    while (true) {
        if (*) return true;
        if (\neg \text{enc}_\land^c(s))
            return enc^c_{\lor}(s);
        s := choose({s' | R(s, s')});
    }
}
```
However, because we are working with a (finite) SPL program text $P$ rather than an arbitrary transition system $M$, we can instead specialize our encoding for $P$ as shown in Figure 6.2. Here, we use the following operator on the program text:

**Definition 6.2 (Command rewrite).** For program text $P$,

\[
\begin{align*}
(P_1 ; P_2)[c/P'] & \equiv (P_1[c/P'] ; P_2[c/P']) \\
(P_1 + P_2)[c/P'] & \equiv (P_1[c/P'] ; P_2[c/P']) \\
P^*[c/P'] & \equiv (P[c/P'])^* \\
\text{assume}(b)[c/P'] & \equiv \text{assume}(b) \\
\text{skip}[c/P'] & \equiv \text{skip} \\
c[c/P'] & \equiv P'
\end{align*}
\]

With this operator, our encoding instruments a fragment of code after each command $c$ in the input program.

**Eventuality.** We use a similar encoding for AF, also shown in Figure 6.2. As in the guarded transition system encoding $G_{M,\Phi}^M$, we use two auxiliary variables within a given enc$^\psi$: dup (i.e. dup$^\psi$), and $s$ (i.e. $s^\psi$). These are procedure-local variables so they need not have unique names.

As in $G_{M,\Phi}^M$, our encoding must allow a program analysis to demonstrate that all paths must eventually reach a state where the subformula holds. While exploring the reachable states in $R$ the encoding may, at every point, nondeterministically decide to capture the current state (setting dup to true and saving $s$ as $s'$). When each subsequent state $s$ is considered, a check is performed that there is some rank function $f \in \mathcal{M}$ that witnesses the well-foundedness of the transitive closure of this particular subset (walk$^\psi_f$) of the transition relation $\mathcal{I}$. In Section 6.4 we describe a method for obtaining a sufficient set of rank functions $\mathcal{M}$ for linear arithmetic programs.

**Partial evaluation.** Our procedural encoding affords us the opportunity to apply a number of static optimizations from abstract interpretation that facilitate the application of current program analysis tools. These optimizations are described in Section 8.1. For example, because the program state is passed on the stack, a procedure call enc$^\psi$ for a subformula $\psi$ will not modify variables in the outer scope, and thus can be treated as skip statements when analyzing the iterations of $R$. Invariants within a given subprocedure can be vital to the pruning, simplification, and partial evaluation required to prepare the output of $\mathcal{E}$ for program analysis.

---

1This is an adaptation of a known technique [CPR06]. However, rather than using assert to check that one of the ranking functions in $\mathcal{M}$ holds, our encoding instead returns false, allowing other possibilities to be considered (if any exist) in outer disjunctive or AF formulae.
Related work. Our work shares some similarities with the finite-state model checking procedure CEX from Figure 6 in Clarke et al. [CJL02]. The difference is that a symbolic model checking tool is used as a sub-procedure within CEX, making CEX a recursively defined model checking procedure. The finiteness of the state-space is crucial to CEX, as in the infinite-state case it would be difficult to find a finite partitioning \emph{a priori} from which to make a finite number of model checking calls when treating temporal operators such as $AG$ and $AF$. Our encoding, by contrast, is not a recursively defined algorithm that calls a model checker at each recursion level, but rather a transformation that produces a procedural program that encodes the proof search-space. This program is constructed such that it can later be symbolically analyzed using (infinite-state) program analysis techniques. When applied to the encoding, the underlying analysis tool is then given the task of finding the necessary finite abstractions and (possibly) procedure summaries.

6.3 Example

Example 6.2 (Acquire/release). The following SPL program is a typical lock acquire/release-style program, where we are interested in proving the $\forall$CTL property $\Phi = AG[(x = 1) \Rightarrow AF(x = 0)]$. Assume that initially $x = 0$.

```
while(*) {
  x := 1;
  n := *;
  while(n>0) {
    n := n - 1;
  }
  x := 0;
}
while(true) { skip }
```

The output of $E(P,M,\Phi)$, after performing several optimizations (discussed above and in Section 3) is given in Figure 6.3. Notice that rather than passing the program counter on the stack, we instead specialize each procedure with respect to the program counter (e.g. we have $\text{enc}_{RG}^{RL}3$ where 3 indicates that execution will begin where $pc = 3$). For every $(\psi, \kappa) \in \text{sub}(\Phi)$ and $pc$ valuation, there is a corresponding method $\text{enc}_{\psi}^{\kappa}_{pc}$. We have omitted many of the procedures which are unneeded. Since we are working with a linear arithmetic program where ranking functions can be given as linear inequalities, integer $<$ is a sufficient ordering for $\prec$. The main procedure in the encoding initializes the program state (i.e. $x, n$) and then asserts that $\text{enc}_{AG(x = 1) \lor AF(x = 0)}^{\kappa}0$ cannot return false. An execution of this program consists of a cascade of calls down the hierarchy of sub-procedures. Each procedure for a subformula maintains invariants $INV_1$ and $INV_2$. This
void main {
  bool x; nat n;
  { x := 0; n := *; }
  assert( enc_{AG,AF,x=0}^L-0(x,n) );
}

bool enc_{AG,AF,x=0}^L-0(bool x, nat n) {
  while(*) {
    x := 1;
    if (~enc_{AF,x=0}^L-3(x,n)) { return false; }
    if (*) return true;
    n := *;
    while(n>0) {
      if (*) return true;
      n–;
    }
    x := 0;
  }
  while(1) { if (*) return true; }
}

bool enc_{AF,x=0}^L-3(bool x, nat n) {
  if (x ≠ 1) return true;
  return enc_{AF,x=0}^L-3(x,n);
}

bool enc_{AF,x=0}^L-3(bool x, nat n) {
  dup2 := dup5 := dup9 := false;
  goto lab_3;
  while(*) {
    if(*) return true;
    if(x==0) return true;
    if(dup2 && ∄ f ∈ M.f(x_2,n_2) > f(x,n)) { return false; }
    if(~(−dup2∧*){dup2:=1;x_2:=x;n_2:=n;})
      x := 1;
    lab_3:
    if (x==0) return true;
    n := *;
    while(n>0) {
      lab_5:
      if(*) return true;
      if(x==0) return true;
      if(dup5 && ∄ f ∈ M.f(x_5,n_5) > f(x,n)) { return false; }
      if(~(−dup5∧*){dup5:=1;x_5:=x;n_5:=n;})
        n–;
    }
    x := 0;
    if (x==0) return true;
  }
  while(1) { if(*) return true;
    if(x==0) return true;
    if(dup9 && ∄ f ∈ M.f(x_9,n_9) > f(x,n)) { return false; }
    if(~(−dup9∧*){dup9:=1;x_9:=x;n_9:=n;})
  }
}

Figure 6.3: Output of the encoding when applied to Example 6.2.

encoding allows us to ask questions of the form “starting now (i.e. from this state) does there exist an execution that violates my property,” and answer them using standard analysis tools.

For example, procedure \( \text{enc}_{AG,AF,x=0}^{\L} \) corresponds to the property \( \text{AG}(x ≠ 1 ∨ \text{AF}(x = 0)) \) and returns \text{false} if there is a reachable state where \( (x ≠ 1 ∨ \text{AF}(x = 0)) \) does not hold. It accomplishes this by calling \( \text{enc}_{AF,x=0}^{\L} \) on each line and passing the current state.

If \( (x ≠ 1 ∨ \text{AF}(x = 0)) \) does not hold from the current state, then there will be a way for \( \text{enc}_{AF,x=0}^{\L} \) to return \text{false}, in which case \( \text{enc}_{AG,AF,x=0}^{\L} \) immediately
returns \texttt{false} (leading to an assertion failure in \texttt{main}). The procedures for disjunction \((\text{\texttt{enc}}^L_{(x \neq 1) \lor \texttt{AF}(x = 0)})\) and atomic propositions \((\text{\texttt{enc}}^L_{x = 1} \text{ and } \text{\texttt{enc}}^L_{x = 0})\) are straight-forward following Fig. 6.2 and also maintain \texttt{INV}_1. We have inlined atomic propositions.

The procedure \(\text{\texttt{enc}}^R_{\text{AF}(x = 0)}\) is, in some sense, the complement of \(\text{AG}\). It is designed to return \texttt{true} whenever there is a path to a state where \(x = 0\) holds, and will return \texttt{false} if there is an infinite execution that never reaches such a state. This is accomplished by checking at each state (i.e. on each line of the program) whether \(\text{\texttt{enc}}^L_{x = 0}\) (which as been inlined) returns \texttt{true}, and returning \texttt{false} if a location is reached multiple times and there is no ranking function in \(\mathcal{M}\) that is decreasing.

A program analysis tool will return a counterexample if applied to the program given in Figure 6.3 as we have not found a sufficient finite set of ranking functions \(\mathcal{M}\). We now describe a method for discovering such an \(\mathcal{M}\).

### 6.4 Rank function synthesis

Recall that we must ultimately find a finite set of ranking functions \(\mathcal{M}\) such that a program analysis can prove for every \(s \in I\) that \(\text{\texttt{enc}}^L_{\Phi}(s)\) does not return \texttt{false}. Our top-level algorithm adapts a known method [CPR06] in order to iteratively find a sufficient \(\mathcal{M}\) as follows:

**Algorithm 6.2. \(\forall\text{CTL verification}\)** We can verify that \(\forall\text{CTL property } \Phi\) holds of a program \(P\) with the following rank function refinement procedure:

```plaintext
let prove(P, \Phi) =
    let \(\mathcal{E}_\Phi = \text{PEval}(\mathcal{E}, \Phi, P)\) in
    \(\mathcal{M} := \emptyset\)
    while (\(\mathcal{E}_\Phi(\mathcal{M})\) can return false) do
        let \(\chi\) be a counterexample in
        if \(\exists\) lasso path fragment \(\chi'\) from \(\chi\) then
            if \(\exists\) witness \(f\) showing \(\chi'\) w.f. then
                \(\mathcal{M} := \mathcal{M} \cup \{f\}\)
            else
                return \(\chi\)
        else
            return \(\chi\)
        done
    return \text{Success}
```

This algorithm begins with the empty set for \(\mathcal{M}\), and constructs the set of procedures, partially evaluating them with \text{PEval} (see Section 8.1). Then, in our implementation, new ranking functions are automatically synthesized by examining counterexamples.
**Definition 6.3.** (*∀ CTL counterexample*) A counterexample $\chi$ to an $\forall$ CTL property $\Phi$ is given by the following grammar:

$$\chi ::= \text{cex}_\alpha \text{ of } S$$

$$| \text{cex}_{L\alpha} \text{ of } \chi$$

$$| \text{cex}_{R\alpha} \text{ of } \chi$$

$$| \text{cex}_\kappa \text{ of } \chi \times \chi$$

$$| \text{cex}_{AF} \text{ of } \varpi \times \varpi \times \chi$$

$$| \text{cex}_{W} \text{ of } \varpi \times \chi \times \chi$$

where the parameter $\kappa$ denotes the path through the formula, and $(\varpi : S \text{ list})$ is a finite path through the encoding $E$.

A counterexample for an $AG$ subformula is a special case of $AW$ and can be represented as $\text{cex}_{AG}$ of $\varpi \times \chi$. Note that often tools will not report a concrete trace but rather a *path*, i.e. a sequence of program counter values corresponding to a class of traces (in rare instances paths may be reported that are spurious). The counterexample structure for an atomic proposition $\text{cex}_\alpha$ is simply a state in which $\alpha$ does not hold. Counterexamples for conjunction and disjunction are as expected. A counterexample to an $AG$ property is a path to a place where there is a counterexample to the sub-property. A counterexample to an $AF$ property is a “lasso”—a stem path to a particular program location, then a cycle which returns to the same program location, and a sub-counterexample along that cycle in which the sub-property does not hold. Finally, an $AW$ counterexample is a path to a place where there is a sub-counterexample to the first property as well as a sub-counterexample to the second property.

In our encoding we obtain these tree-shaped counterexamples effectively for free with program analysis tools (*e.g.* SLAM or BLAST) that report stack-based traces for assertion failures. Information about the stack depth available in the counterexamples allows us to re-construct the tree counterexamples. That is, by walking backward over the stack trace, we can determine the tree-shape of the counterexample. Consider, for example, the case of $AF$. The counterexample found by the underlying tool will visit commands through the encoding of $E$, including points where $\text{dup}$ is set to $\text{true}$. The commands from the input program can be used to populate an instance of $\chi$.

When a counterexample is reported that contains an instance of $\text{cex}_{AF}$ (*i.e.* a “lasso fragment”) it is possible that the property still holds, but that we have simply not found a sufficient ranking function to witness the termination of the lasso. In this case our algorithm finds the lasso fragments and attempts to enlarge the set of ranking functions $M$.

**Example.** We return again to Example 6.2, and apply Algorithm 6.2. Initially we let $M \equiv \emptyset$. Running a refinement-based safety prover will yield a counterexample pertaining
to line lab_5 of \texttt{enc}^{RL_{AF(x=0)}} where we denote a state as \([\frac{n}{pc}]\) and we denote transition relations as \([\begin{array}{c} \frac{x}{pc} \\ \frac{n}{pc} \end{array}]\):

\[
\begin{align*}
(cex_{AG}[ & \begin{array}[c]{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}] :: [\begin{array}[c]{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 5 \end{array}] :: [\begin{array}[c]{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 5 \end{array}]) \\
(cex_{\lor} (cex_{\land} [ & \begin{array}[c]{c} x \\ n \\ pc \end{array}] )) \\
(cex_{AF} [ & \begin{array}[c]{c} 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \end{array}] :: [\begin{array}[c]{c} 5 \\ pc \cdot n + 1 \\ pc \cdot n + pc \end{array}] :: [\begin{array}[c]{c} 5 \\ 5 \\ 5 \\ 5 \\ 5 \\ 5 \end{array}])
\end{align*}
\]

This counterexample appears because we have not found a finite \(\mathcal{M}\) such that in \(\texttt{enc}^{RL_{AF(x=0)}}\) the check that \(\exists f \in \mathcal{M}. f(x_5,n_5) > f(x,n)\) always holds.

In our implementation we then use a rank function synthesis tool on this counterexample (as described by Cook et al. \cite{CPR06}), find that ranking can be done on \(n\), and obtain a new \(\mathcal{M} \equiv \{\lambda s. s(n)\}\). With this new \(\mathcal{M}\) in place, \(\texttt{enc}^{\land _{AG((x=1) \lor AF(x=0))}}\) always returns \texttt{true}, and consequently, by Theorem 6.1, \(\Phi\) holds of the original program.
Determinization refinement

In Chapter 5, we described a novel prediction technique that enables us to prove LTL properties of a program using an $\forall$CTL proof procedure, when provided an appropriate vector of decision predicates. However, we assumed that the decision predicates had been given to us by an oracle.

In this chapter we describe our implementation of a counterexample-guided refinement strategy for iteratively improving the vector of decision predicates. In Section 7.1 we describe the overall refinement loop: we begin with an empty vector of decision predicates and apply the symbolic determinization procedure from Chapter 5. We then examine an $\forall$CTL counterexample and determine whether it is a valid LTL counterexample and, if not, extracts a symbolic representation of the nondeterminism (Section 7.2). Finally, in Section 7.3 we describe a decision predicate synthesis procedure. From a symbolic representation of branching in an $\forall$CTL counterexample we can apply Farkas’ lemma [Far02] and an SMT solver (e.g. Z3 [Z3] or Yices [DDM06] or CVC [BT07]) to generate decision predicates in the form of linear arithmetic expressions.

7.1 Iterated algorithm

In this section we describe a counterexample-guided refinement proof procedure for LTL.

Algorithm 7.1 (Determinization refinement). For all $M$ and LTL property $\varphi$,
\begin{verbatim}
prove_{LTL}(M, \varphi) \equiv \\
\Omega := \emptyset \\
let \Phi = \eta(\varphi) in \\
while true do \\
let M^\Omega = \text{Determinize}(M, \Omega) in \\
match \text{Prove}_{\forall CTL}(M^\Omega, \Phi) with \\
| Succeed -> return Succeed \\
| Fail(\chi) -> \\
let \Omega' = \text{Refine}(\chi) in \\
if(\Omega' = \emptyset) \\
let \pi \in \chi in \\
return Fail(\pi) \\
else \\
\Omega := \Omega \cup \Omega' \\
done
\end{verbatim}

The algorithm is designed to iteratively find a sufficient set of decision predicates \( \Omega \) such that proof tools for CTL (such as in Chapters 4 and 5) can be used to prove an LTL property \( \varphi \) of the system \( M \).

The algorithm proceeds as follows. When \( \Omega = \emptyset \), \text{Determinize}(M, \Omega) = M. Thus, on the first iteration of the loop our procedure is attempting to prove \( \varphi \) via a simple approximation \( \Phi \) together with the original system \( M \). When given a non-empty set of decision predicates, \text{Determinize} builds \( M^\Omega \) by conjoining the original transition relation of \( M \) with a relation that specifies the behavior of a prophecy variable for each decision predicate (Chapter 5). For a set of decision predicates \( \Omega \), if \( \Phi \) holds, then \( \varphi \) also holds. Thus, whenever we find a sufficient set of predicates to prove \( \Phi \), we have proved \( \varphi \).

\text{Refine} is used to determine if an \( \forall \)CTL counterexample found by \text{Prove}_{\forall CTL} \) represents a real LTL counterexample or something spurious (Section 7.2). At first glance there is a formidable semantic gap between the two types of counterexamples: \( \forall \)CTL counterexamples are trees, whereas LTL counterexamples are traces. However, if all of the paths through the counterexample \( \chi \) represent the same path or its prefixes, then every one of these paths is a legitimate counterexample to \( \varphi \). In this case \text{Refine} returns \( \emptyset \). Otherwise, if \( \chi \) represents more than one path in the program, \text{Refine} returns a non-empty set of new decision predicates that characterize the non-determinism that distinguishes between the different paths.

After examining an \( \forall \)CTL counterexample, discovering new decision predicates, and performing \text{Determinize} again, we find that the counterexample no longer appears. We may obtain a new \( \forall \)CTL counterexample and need to iterate but in practice after a few iterations we find that the \( \forall \)CTL prover succeeds (see Chapter 8 for experimental results).
Thus, we are able to prove LTL properties with CTL proof techniques in cases where this strategy would have previously failed.

7.2 Decision predicate refinement

In this section we describe the implementation of a routine \textit{Refine} that examines counterexamples from a branching-time verification tool and discovers predicates which characterize the nondeterministic branching within them if any nondeterminism exists.

7.2.1 Correctness criteria for \textit{Refine}

We begin with some definitions:

\textbf{Definition 7.1 (\(\chi\)-elimination).} For all \(M, \Phi\) and counterexample \(\chi\) to \(M \models \chi\Phi\),

\[
\text{elim } M \Phi \chi \Omega \equiv \text{\(\chi\) is not a valid counterexample to } M^\Omega \models \Phi
\]

Note that we will use this definition iteratively, so the notation \((M^\Omega)^\Omega' \equiv M^{\Omega \cup \Omega'}\).

7.2.2 Implementation

The procedure \textit{Refine} given Figure 7.1 consumes an \(\forall\)CTL counterexample and returns sets of predicates which distinguish nondeterministic branching. Our implementation of \textit{Refine} begins by constructing a corresponding counterexample flow-graph (CEFG) \(G\) that represents all paths in the counterexample using the procedure \textit{cefg}. In our representation of a counterexample flow graphs we use a standard graph-based notation, where nodes \(\text{nd} \in \mathbb{N}\) correspond to states in the counterexample, and edges are triples \((\text{nd}_1, r, \text{nd}_2)\) consisting of a starting node, a transition relation \(r\) from the counterexample and a destination node.

\textit{Refine} then walks down the counterexample flow-graph, at each step simultaneously exploring all possible next steps. If a pair of possible next steps are distinguishable via a predicate from \(\text{PSynth}_D\) then that predicate is immediately returned. \textit{Refine} depends on the availability of a predicate synthesis mechanism defined as follows:

\textbf{Definition 7.2 (Predicate synthesis).} For all \(R, R'\), and abstract domain \(D\),

\[
\text{PSynth}_D(R, R') \equiv \begin{cases} 
\{(a, b), (a, \neg b)\} & \text{such that } R \subseteq [a \land b']^R \text{ and } R' \subseteq [a \land \neg b']^R \\
\emptyset & \text{if no such } a, b \text{ exist}
\end{cases}
\]
The procedure \( \text{PSynth}_{D}(r,r') \) consumes two transition relations \( r,r' \) and returns two pairs of decision predicates. The implementation of \( \text{PSynth}_{D} \) will differ, depending on the context (i.e. finite-state systems expressed at the bit-level, infinite-state systems expressed over linear arithmetic, etc). We assume that for a given domain \( D \) (a) that \( D \) is capable of distinguishing two states and (b) that \( \text{PSynth}_{D} \) is capable of discovering sufficient elements in \( D \) to do so. If these assumptions do not hold then in some instances our technique may be unable to sufficiently determinize. In our implementation, described in Section 8.3, we use constraint-solving techniques to find predicates which are monomials.
over linear inequalities.

### 7.2.3 Example

**Example 7.1.** Consider applying Algorithm 7.1 on Example 5.1 from Chapter 5. As we described, the system $M$ does not respect the property $\Phi$, and we obtain the counterexample $\chi$ as seen before:

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
```

Our implementation of **Refine** simultaneously symbolically simulates all possible paths through this graph and tries to unify them into a single path through $M$. In this case it would begin its execution by visiting first $pc = \ell_1$ and then $pc = \ell_2$, after which it would discover that, for all paths of the graph to represent the same path, it must unify $pc = \ell_2$ and $pc = \ell_3$, which cannot be done. Thus, in this case, the $\forall$CTL counterexample $\chi$ will be deemed spurious to the LTL property and refinement $\Omega'$ will include the decision predicates $(pc = \ell_2, pc = \ell_3)$ and $(pc = \ell_2, pc = \neg \ell_3)$. The decision predicate $(pc = \ell_2, pc = \ell_3)$ characterizes the choice: “when $pc = \ell_2$, will $pc' = \ell_3$ or not?” Notice also that, in this particular case, the predicates selected are over program locations, but this is not true in general (see Example 7.3 in Section 7.2).

For this example, an $\forall$CTL prover may generate the following counterexample:

$$(\text{cex}_\text{AF} \left[ \begin{array}{c} 1 \\ 2 \\ 3 \\ 2 \\ 1 \end{array} \right] :: \left[ \begin{array}{c} 3 \\ 2 \\ 1 \end{array} \right] :: \left[ \begin{array}{c} 3 \\ 2 \end{array} \right], \text{cex}_\text{af} \left[ \begin{array}{c} 5 \\ 6 \end{array} \right])$$

where a state is represented as $\left[ \begin{array}{c} n \end{array} \right]$. From this counterexample, we use **cefg** to construct the counterexample flow-graph $\Gamma$ given in Section 7.1. Each arc represents a possible transition within the counterexample tree. The procedure **Refine** then walks all possible paths of the control-flow graph simultaneously, starting from the first node as follows:

- **Iteration 1:** $N = \{n0\}$, $S = \emptyset$
- **Iteration 2:** $N = \{n1\}$, $S = \{n0,n1\}$
- **Iteration 3:** $N = \{n2,n3\}$, $S = \{n0,n1,n2,n3\}$
After the first and second iterations PSynthD does not discover a predicate to distinguish the two branches, but after the third call to Refine, the predicate pairs \((pc = \ell_2, pc = \ell_3)\) and \((pc = \ell_2, pc \neq \ell_3)\) are discovered, which distinguish paths that remain in the loop or exit the loop. A new machine is then constructed with prophecy variables corresponding to these decisions, and for this new machine an ∀CTL verifier can prove that the property holds.

### 7.2.4 Progress

We now discuss the issue of progress. After applying a determinization from a counterexample, we would like to know that, at least, the counterexample will not reappear. We denote by \(\chi_{\text{ext}}\) the counterexample which consists of the external projection of paths in all components.

We now show that for a given counterexample \(\chi\), if Refine discovers predicates, then our algorithm produces a new machine for which \(\chi_{\text{ext}}\) is not a counterexample.

**Lemma 7.2.** *(Counterexample elimination)* For a counterexample \(\chi\) to \(M \models \Phi\),

\[
\text{elim } M \Phi \chi \text{ Refine}(\chi).
\]

where \(\text{Refine}(\chi) \neq \emptyset\).

**Proof.** (Sketch) Assume \(\chi\) is a counterexample to \(R^{\Omega'} \models \Phi\). Let \((a_i, b_i) \in \text{Refine}(\chi)\). By definition of \(\text{Refine}\) in Figure 7.1 this predicate pair must have come from a subcomponent of the counterexample \(\chi\) flow graph of the form \((nd, r, nd')\), \((nd, r', nd'')\). Moreover \(a_i((r)|_1), b_i((r)|_2)\) and \(\neg b_i((r')|_2)\). Now, in the new machine the prophecy vector is augmented with a new element \(\rho_i\). So the set of states denoted \(((r)|_1, \rho)\) have either \(\rho_i = 0\) or \(\rho_i \in \{1,1,2,...\}\). According to \(R^{\Omega'}\), either \(((r)|_1, \rho), (\{(r)|_2, \rho)\) is enabled or else \(((r)|_1, \rho), (\{(r)|_2, \rho)\) is enabled, but not both. Hence, \(\chi\) is not valid in the new machine. \(\square\)

**Remark on completeness.** Our algorithm is sound but not complete. The lack of a completeness result is for a couple of reasons. First, for a given ∀CTL counterexample \(\chi\), the routine PSynthD must be able to discover predicates to characterize nondeterminism in \(\chi\). However, since we use approximation *(e.g. with linear arithmetic)*, it will not always be able to discover sufficient predicates when the exist.

Second, even when we have a perfect PSynthD routine, some ∀CTL counterexamples may be spurious, as the underlying ∀CTL also supports only over-approximation in linear arithmetic. Consequently, when \(\text{Refine}(\chi) = \emptyset\) we cannot necessarily claim that we have a valid LTL counterexample. Furthermore, as mentioned previously, there are some
non-terminating programs that do not have a infinitely-repeated “lasso path.” In these instances, the ∀CTL tool itself will either hang or return spurious counterexamples. All of the above issues are the subject of ongoing investigation.

### 7.3 Decision predicate synthesis

In Section 7.2, we have assumed the existence of a predicate synthesis mechanism \( \text{PSynth}_D \) that met the constraints given in Definition 7.2. Depending on the configuration of the systems considered by the tool, \( \text{PSynth}_D \) will need to be implemented in different ways. Here we describe a particular method of synthesizing predicates for counterexamples drawn from the style of programs typically accepted by modern model checking tools for infinite-state programs.

As is true in many symbolic model checking tools for software, we will assume that counterexamples are sequences of commands drawn from a path in the program. We will assume that these commands are over a finite set of arithmetic variables, and that the conditional checks and assignment statements only use linear arithmetic. Given this context, an implementation can represent the relations passed to \( \text{PSynth}_D \) as conjunctions of inequalities using variables. For example, the command sequence

\[
\ell_{41} : x := x - 1;
\ell_{21} : \text{assume}(x > 0);
\ell_{10} : y := x;
\]

which might represent a piece of a counterexample can be represented as a relation from valuations on \((x, y, pc)\) to valuations on \((x', y', pc')\) where

\[
\exists x_1, x_0, y_1, y_0.
\] 

\[
\land \left\{ \begin{array}{l}
pc = \ell_{41} \land pc' = \ell_{10} \land x = x_0 \land x' = x_1 \land y = y_0 \land y' = y_1 \\
\land x_1 = x_0 - 1 \land x_1 > 0 \land y_1 = x_1
\end{array} \right.
\]

We can reduce the search for predicates in this setting to the search for functions satisfying a set of constraints. In this instance we hope to find families of affine functions \( f \) and \( g \) such that the following conditions are true

1. \((\exists V'.R_1 \land \exists V'.R_2) \Rightarrow \land_{i \in \text{dom}(f)} f_i(V) > 0\)
2. \(R_1 \Rightarrow \land_{i \in \text{dom}(g)} g_i(V') > 0\)
3. \(R_2 \Rightarrow -(\land_{i \in \text{dom}(g)} . g_i(V') > 0)\)

The set of pre-states common to both relations \( R_1 \) and \( R_2 \) are given \( S \equiv \exists V'.R_1 \land \exists V'.R_2 \), *i.e.* we are existentially quantifying out the post-states by quantifying out the variables
Examples

Consider the following program for which we would like to prove \( \varphi = (FG y = 1) \lor (Fx \geq t) \):
\( \ell_0: \ x = y = 0; \ t = *; \)
\( \text{while}(\ast) \)
\( \ell_1: \ x++; \)
\( \ell_2: \ t = *; \)
\( \ell_3: \ \text{if} \ (x < t) \)
\( \ell_4: \ y = 1; \)
\( \text{while} \ (\text{true}) \)
\( \ell_5: \ \text{skip}; \)

The machine representing this program can be encoded as follows:

\[
S = \begin{bmatrix} N & N & N & L \end{bmatrix} \quad \text{denoted} \quad \begin{bmatrix} x & y & t \end{bmatrix}_{\text{pc}} \\
I = \begin{bmatrix} N & N & N & N \end{bmatrix} \\
R = \left[ \begin{array}{c}
(pc = \ell_0 \land pc' = \ell_1 \land x' = x \land y' = y \land t' = t) \lor \\
(pc = \ell_0 \land pc' = \ell_2 \land x' = x \land y' = y \land t' = t) \lor \\
(pc = \ell_1 \land pc' = \ell_1 \land x' = x + 1 \land y' = y \land t' = t) \lor \\
(pc = \ell_1 \land pc' = \ell_2 \land x' = x + 1 \land y' = y \land t' = t) \lor \\
(pc = \ell_2 \land pc' = \ell_3 \land x' = x \land y' = y \land t' \in \mathbb{N}) \lor \\
(pc = \ell_3 \land pc' = \ell_4 \land x < t \land x' = x \land y' = y \land t' = t) \lor \\
(pc = \ell_3 \land pc' = \ell_5 \land x \geq t \land x' = x \land y' = y \land t' = t) \lor \\
(pc = \ell_4 \land pc' = \ell_5 \land x' = x \land y' = 1 \land t' = t) \lor \\
(pc = \ell_5 \land pc' = \ell_5 \land x' = x \land y' = y \land t' = t) \right]^{R}
\]

Using an \(\forall\)CTL prover, we may obtain the first counterexample in Figure 7.2. From this counterexample, we use \text{cefg} to construct the first counterexample flow graph in Figure 7.3. Each arc represents a possible transition within the counterexample tree. The procedure \text{Refine} then walks all possible paths of the control-flow graph simultaneously, starting from \text{n0} as follows:

\[
\text{Iteration 1: } N = \{n0\}, \quad S = \emptyset
\]

In this iteration, \text{Refine} finds that \(N' = \{n1,n2\}\) and that \(R = \{(\ell_0, \ell_1), (\ell_0, \ell_2)\}\). Taking the (only) pair of relations from \(R\), PSYNTHD generates the predicate pairs \((pc = \ell_0, pc = \ell_1)\) and \((pc = \ell_0, pc \neq \ell_1)\). Corresponding prophecy variables are created, and the \(\forall\)CTL verifier is used on the newly constructed machine, resulting in the next counterexample in Figure 7.2. We then get the second counterexample flow graph in Figure 7.3 and the \text{Refine} explores it as follows:
After the first iteration, PSYNTHD does not discover any predicates to distinguish the two branches, but after the second iteration the predicate pairs \((pc = \ell_1, pc = \ell_1)\) and \((pc = \ell_1, pc \neq \ell_1)\) are discovered, which distinguish paths that remain in the loop or exit the loop. The \(\forall\)CTL verifier is executed once again, resulting in the third counterexample in Figure 7.2. The counterexample flow-graph is given in Figure 7.3 and REFINE explores it as follows:

\[
\begin{align*}
\text{Iteration 1:} & \quad N = \{n0\}, \quad S = \emptyset \\
\text{Iteration 2:} & \quad N = \{n1,n2\}, \quad S = \{n0,n1,n2\}
\end{align*}
\]

In the final iteration, PSYNTHD discovers the predicate pairs \((pc = \ell_2, t \geq x)\) and \((pc = \ell_2, t < x)\). Notice that the second predicate is over a program variable other than \(pc\) – in the next example we will see that \(pc\) is not always sufficient to distinguish paths. Running the \(\forall\)CTL verifier one more time yields no counterexamples. Hence the original LTL property holds.

**Example 7.3.** In the examples above, almost all predicates were over the program counter variable \(pc\). In many cases, the program counter serves as a convenient way of distinguishing paths through the program. However, this is not always the case. Consider proving the property \((G x = 0) \lor (F x = 20)\) for the following program:
\begin{verbatim}
ℓ0: x = 0;
while(x<20)
ℓ1: x := (x==0)*\{0,1\} + (x==1)*20;
while(true)
ℓ2: skip
\end{verbatim}

The notation \{0, 1\} represents nondeterministic choice between 0 or 1. The LTL property holds because in traces where this nondeterministic choice is always 0, the property \(G x = 0\) holds. For any trace in which the nondeterministic choice is 1, the property \(F x = 20\) holds.

We shall represent the state as \([x]\) where \(x \in \mathbb{N}\). An \(\forall\)CTL prover will generate the following counterexample to \((AG x = 0) \lor (AF x = 20)\):

\[
(\text{cex}_\varnothing, (\text{cex}_{AG}[\ell_0] : [\ell_1]^0 : [\ell_1]^1, (\text{cex}_\alpha[\ell_1]^1)))
(\text{cex}_{AF}[\ell_0] : [\ell_1]^0 : [\ell_1]^0, \text{Id}, (\text{cex}_\alpha[\ell_1]^0)))
\]

For this counterexample \textsc{Refine} would explore the corresponding CEFG, and discover the decision predicates \((x = 0, x = 1)\) which distinguishes the transition \([\ell_1]^0, [\ell_1]^1\) from \([\ell_1]^0, [\ell_1]^0\). Importantly, there is no predicate over the program counter variable alone which distinguishes these two transitions. We can now synthesize a prophecy variable corresponding to this decision predicate and an \(\forall\)CTL prover will discover a proof of the \(\forall\)CTL property, implying that the original LTL property holds.
Using the techniques described earlier, we have developed a prototype implementation for verifying LTL and ∀CTL properties of programs. Our tool is written in F#, and uses the CIL compiler infrastructure front-end for (a subset of) C programs. Our implementation parses an input program and an LTL property, and performs a source-to-source translation, implementing the transformation $E$ from Chapter 6.

Recall the overall shape of the tool from Figure 1.3. The outer level PROVE_{LTL} involves the iterative determinization refinement procedure from Chapter 5. At the inner level PROVE_{∀CTL} implements the encoding $E$ (Chapter 6). We use SLAM [BBC+06] as our implementation of the safety prover, and RANKFINDER [PR04a] as the rank function synthesis tool. In this Chapter we will explain our optimizations, benchmarks, and report on experiments with the tool.

8.1 Implementation and optimizations

PROVE_{∀CTL} operates by encoding the ∀CTL temporal verification question $P \models \Phi$ as a program analysis task (Chapter 6). We will now demonstrate some optimizations as they pertains to Example 6.2, transforming the encoding in Figure 8.1 into the optimized encoding in Figure 6.3.

Program counter specialization. For even modest-sized programs, encoding the program counter in the state $s$ leads to a transformed program which requires an enormous amount of disjunction to reason about each procedure. Since $pc$ is taken from a finite domain $L$, we can specialize the encoding with respect to the program counter. Consider the following example:
bool enc^κ_0(pc, s) {
    switch(pc) {
        case 0: goto lab_enc^κ_0;  
        case 1: goto lab_enc^κ_1;  
        ...  
        case n: goto lab_enc^κ_n;  
    }
    pc = 0;
    lab_enc^κ_0:  ...;
    pc = 1;
    lab_enc^κ_1:  ...;
    ...  
    pc = n;
    lab_enc^κ_n:  ...;
}

bool enc^κ_1(s) {
    goto lab_enc^κ_1;
    ...  
    lab_enc^κ_1:  ...;
}

...  

bool enc^κ_n(s) {
    goto lab_enc^κ_n;
    ...  
    lab_enc^κ_n:  ...;
}

Where we previously had the procedure on the left involving a case-split over \( pc \), we can instead specialize the procedure for each value of \( pc \) as shown to the right. Call sites are modified to call the appropriate procedure depending on their program location.

This specialization shifts the need for elaborate procedure summaries for a few procedures to compact per-procedure summaries for many more procedures. In practice (as discussed below) this is far preferable. Note that as future work we hope to implement a more lazy expansion and partial evaluation \( \text{á la IMPACT} \) [McM06]. This specialization can be seen when comparing, for example, \( \text{enc}^{RLF}_A(x=0) \) in Figure 8.1 with \( \text{enc}^{RLF}_A(x=0) \) in Figure 6.3.

Inlining. Sub-procedures of \( E \) for non-temporal formulae can be inlined. For example, rather than the following procedures for the property \( AG(x = 1 \lor (y > 0 \land z < 0)) \):

\[
\begin{align*}
\text{bool enc}^L_{RLF}(y>0) & (int a, x, y, z) \{ \text{return } (y > 0); \} \\
\text{bool enc}^R_{RLF}(z<0) & (int a, x, y, z) \{ \text{return } (z < 0); \} \\
\text{bool enc}^L_{Lr}(x=1) & (int a, x, y, z) \{ \text{return } (x==1); \} \\
\text{bool enc}^R_{Lr}(y>0 \land z<0) & (int a, x, y, z) \{ \ldots \} \\
\text{bool enc}^L_{L}(x=1 \lor (y>0 \land z<0)) & (int a, x, y, z) \{ \ldots \} \\
\text{bool enc}^L_{AG}(x=1 \lor (y>0 \land z<0)) & (int a, x, y, z) \{ \ldots \} 
\end{align*}
\]

We can instead inline the non-temporal procedures and obtain:
**CHAPTER 8. EVALUATION**

void main {
    x := 0; n := *;
    assert(enc_{AG(x=1)}(\ell_1,x,n) = false);
}

bool enc_{AG(x=1)}(\ell_1,x,n) (int pc, x, n) {
    if (pc == \ell_1) goto lab_1;
    ...
    lab_1:
    if (- enc_{L(x=1)}(\ell_1,x,n)) { return false; }
    if (*) return true;
    while(*) {
        if (- enc_{L(x=1)}(\ell_2,x,n)) { return false; }
        if (*) return true;
        x := 1;
        if (- enc_{L(x=1)}(\ell_3,x,n)) { return false; }
        if (*) return true;
        n := *;
        if (- enc_{L(x=1)}(\ell_4,x,n)) { return false; }
        if (*) return true;
        while(n>0) {
            if (- enc_{L(x=1)}(\ell_5,x,n)) { return false; }
            if (*) return true;
            n - -;
            if (- enc_{L(x=1)}(\ell_7,x,n)) { return false; }
            if (*) return true;
        }
    }
    if (- enc_{L(x=1)}(\ell_8,x,n)) { return false; }
    if (*) return true;
}

bool enc_{L(x=1)}(\ell_1,x,n) (int pc, x, n) {
    if (enc_{LL(x=1)}(pc,x,n)) return true;
    return enc_{RL(x=1)}(pc,x,n);
}

bool enc_{LL(x=1)}(int pc, x, n) {
    return (x \neq 1 ? true : false);
}

bool enc_{RL(x=1)}(int pc, x, n) {
    return (x==0 ? true : false);
}
bool encRL( int pc, x, n) {
    ...
    if (pc == ℓ3) goto lab3;
    ...
    Dup1 := false; ...
    if(*) return true;
    if (encRL(1,x,n)) return true;
    if(dup1 && ∃f ∈ M.f(x1,n1) > f(x,n)) return false;
    if(!dup1 &&*) (dup1:=1;x1:=x;n1:=n;
    while(*) {
        if(*) return true;
        if (encRL(2,x,n)) return true;
        if(dup2 && ∃f ∈ M.f(x2,n2) > f(x,n)) return false;
        if(!dup2 &&*) (dup2:=1;x2:=x;n2:=n;
        x := 1;
        lab3:...
        n := *
        if(*) return true;
        if (encRL(3,x,n)) return true;
        if(dup3 && ∃f ∈ M.f(x3,n3) > f(x,n)) return false;
        if(!dup3 &&*) (dup3:=1;x3:=x;n3:=n;
        n := *;
        if(*) return true;
        if (encRL(4,x,n)) return true;
        if(dup4 && ∃f ∈ M.f(x4,n4) > f(x,n)) return false;
        if(!dup4 &&*) (dup4:=1;x4:=x;n4:=n;
        while(n>0) {
            lab5:...
            n := *;
            if(*) return true;
            if (encRL(5,x,n)) return true;
            if(dup5 && ∃f ∈ M.f(x5,n5) > f(x,n)) return false;
            if(!dup5 &&*) (dup5:=1;x5:=x;n5:=n;
            n := *;
            if(*) return true;
            if (encRL(6,x,n)) return true;
            if(dup6 && ∃f ∈ M.f(x6,n6) > f(x,n)) return false;
            if(!dup6 &&*) (dup6:=1;x6:=x;n6:=n;
            x := 0;
            if(*) return true;
            if (encRL(7,x,n)) return true;
            if(dup7 && ∃f ∈ M.f(x7,n7) > f(x,n)) return false;
            if(!dup7 &&*) (dup7:=1;x7:=x;n7:=n;
            x := 0;
            if(*) return true;
            if (encRL(8,x,n)) return true;
            if(dup8 && ∃f ∈ M.f(x8,n8) > f(x,n)) return false;
            if(!dup8 &&*) (dup8:=1;x8:=x;n8:=n;
            while(1) {
                if(*) return true;
                if (encRL(9,x,n)) return true;
                if(dup9 && ∃f ∈ M.f(x9,n9) > f(x,n)) return false;
                if(!dup9 &&*) (dup9:=1;x9:=x;n9:=n;
            )
        )
    )
}
}

Figure 8.1: The encoding E of Example 6.2 (before the partial evaluation has been applied to obtain the output in Figure 6.3).
bool enc\textsubscript{AG}^{k}(x=1\lor(y>0\land z<0))(\text{int } a, x, y, z) \{ 
  ...
  if (x == 1) return true;
  if (\neg y<0) return false;
  return (z<0 ? true : false);
  ...
\}

For example, in Figure 6.3 we have inlined \text{enc}\textsubscript{LR}^{L}(x=0) within the body of \text{enc}\textsubscript{AF}(x=0).

Ordering disjunction. For a disjunctive property \(\Phi \lor \Phi'\), our encoding in \(\mathcal{E}\) has a choice as to the order in which the sub-procedures are invoked. For example, let us say that the property is \((\text{AGAF } y = 1) \lor (x = 1)\). Clearly in most cases it is easier to show that the sub-procedure corresponding to the atomic proposition \((x = 1)\) cannot return \text{false} rather than showing that the \((\text{AGAF } y = 1)\) cannot return \text{false}. We use a simple cost metric to order sub-procedure calls in disjunctive instances of \(\mathcal{E}\) based on depth of nesting in each subformula. We have already done this optimization in \text{enc}\textsubscript{L}(x=1)\lor\text{AF}(x=0) in Figure 8.1.

Intra-procedural analysis. In our treatment of \text{AF} and \text{AW}, \(\mathcal{E}\) injects a call to the sub-procedure on each line. This can be costly and unnecessary when a statement does not impact the truth value of the subformula. Consider the property \(\text{AG}(x = 1\lor(y > 0\land z < 0))\) and the following fragment of \text{enc}\textsubscript{AG}(x=1\lor(y>0\land z<0)):

\begin{verbatim}
  bool enc\textsubscript{AG}^{k}(x=1\lor(y>0\land z<0))(\text{int } a, x, y, z, int) \{ 
    ...
    if (\neg enc\textsubscript{LR}^{L}(x=1\lor(y>0\land z<0))(a,x,y,z)) return false;
    a := 56;
    if (\neg enc\textsubscript{LR}^{L}(x=1\lor(y>0\land z<0))(a,x,y,z)) return false;
    ...
  }
\end{verbatim}

Clearly, the assignment \(a:=56\) does not impact the truth value of \((x = 1\lor(y > 0\land z < 0))\), so if \text{false} can be returned on Line 3, then \text{false} can also be returned on Line 5. Also, if \text{false} cannot be returned on Line 3, then \text{false} cannot be returned on Line 5. We apply a simple \textit{intra}procedural analysis to remove superfluous calls such as the one on Line 5.

This optimization can be seen in \text{enc}\textsubscript{AG}(x=1)\lor\text{AF}(x=0) and in \text{enc}\textsubscript{AF}(x=0) in Figure 6.3

8.2 Benchmarks

We have drawn out a set of both \(\forall\text{CTL}\) and LTL liveness property challenge problems from industrial code bases. Examples were taken from the I/O subsystem of the Windows
OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. In order to make these examples self-contained we have, by hand, abstracted away the unnecessary functions and struct definitions. We also include a few toy examples, as well as the example from Figure 8 in [CGP+07]. Sources of examples can be found at the following URLs:

http://www.cl.cam.ac.uk/~ejk39/ltl/
http://www.cl.cam.ac.uk/~ejk39/actl/

Heap commands from the original sources have been abstracted away using the approach due to Magill et al. [MBCC07]. This abstraction introduces new arithmetic variables that track the sizes of recursive predicates found as a byproduct of a successful memory safety analysis using an abstract domain based on separation logic [ORY01]. Support for variables that range over the natural numbers is crucial for this abstraction.

8.3 Experiments

As previously mentioned in Chapter 2, there are several available tools for verifying state-based properties of general purpose (infinite-state) programs. Neither the authors of this paper, nor the developer of Yasm [GWC06] were able to apply Yasm to the challenge problems in a meaningful way, due to bugs in the tool. Note that we expect Yasm would have failed in many cases [Gur10], as it is primarily designed to work for unnested existential properties (e.g. $EGp$ or $EFp$). We have also implemented the approach due to Chaki et al. [CCG+05]. The difficulty with applying this approach to the challenge problems is that the programs must first be abstracted to finite-state before branching-time proof methods are applied. Because the challenge problems focus on liveness, we have used transition predicate abstraction [PR05] as the abstraction method. However, because abstraction must happen first, predicates must be chosen ahead of time either by hand or using heuristics. In practice we found that our heuristics for choosing an abstraction a priori could not be easily tuned to lead to useful results.

Because the examples are infinite-state systems, popular CTL-proving tools such as Cadence SMV [cad] or NuSMV [CCG+02] are not directly applicable. When applied to finite instantiations of the programs these tools run out of memory.

The tool described in Cook et al. [CGP+07] can be used to prove LTL properties if used in combination with an LTL to Büchi automata conversion tool (e.g. [GO01]). To compare our approach to this tool we have used two sets of experiments: Figure 8.2 displays the results on challenge problems in $\forall$CTL verification; Figure 8.3 contains results on LTL verification. Experiments were run using Windows Vista and an Intel 2.66GHz processor. In both figures, the code example is given in the first column, and a note as to whether it contains a bug. We also give a count of the lines of code and the shape of the temporal
### Figure 8.2: Comparison between our implementation of Algorithm 6.2 and Cook et al. [CGP+07] on ∀CTL verification benchmarks. All of the above ∀CTL properties have equivalent corresponding LTL properties so they are suitable for direct comparison with the LTL tool [CGP+07].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Acq/rel</td>
<td>14</td>
<td>AG(a ⇒ AFb)</td>
<td>103.48</td>
<td>✓</td>
<td>14</td>
<td>AG(a ⇒ AFb)</td>
<td>103.48</td>
<td>✓</td>
</tr>
<tr>
<td>Fig. 8 of [CGP+07]</td>
<td>34</td>
<td>AG(p ⇒ AFq)</td>
<td>209.64</td>
<td>✓</td>
<td>27.94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toy linear arith. 1</td>
<td>13</td>
<td>p ⇒ AFq</td>
<td>126.86</td>
<td>✓</td>
<td>34.51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toy linear arith. 2</td>
<td>13</td>
<td>p ⇒ AFq</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>6.74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL stmsrv</td>
<td>259</td>
<td>AG(p ⇒ AFAGq)</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>9.56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL stmsrv+bug</td>
<td>259</td>
<td>AG(p ⇒ AFAGq)</td>
<td>87.31</td>
<td>✓</td>
<td>47.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL pgarch</td>
<td>61</td>
<td>AFAGp</td>
<td>31.50</td>
<td>✓</td>
<td>15.20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache progress</td>
<td>314</td>
<td>AG(p⇒(AF∧AF))</td>
<td>685.34</td>
<td>✓</td>
<td>684.24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 1</td>
<td>180</td>
<td>AG(p ⇒ AFq)</td>
<td>901.81</td>
<td>✓</td>
<td>539.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 4</td>
<td>327</td>
<td>AG(p ⇒ AFq)</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>1,114.18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 4+bug</td>
<td>327</td>
<td>(AFa)∨(AFb)</td>
<td>1,223.96</td>
<td>✓</td>
<td>100.68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 5</td>
<td>648</td>
<td>AG(p ⇒ AFq)</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>&gt;14400.00</td>
<td>???</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 7</td>
<td>13</td>
<td>AGAFp</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>55.77</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 8.3: Comparison between our tool and Cook et al. [CGP+07] on LTL benchmarks. For our tool, we use the iterative symbolic determinization strategy from Algorithm 7.1 to prove LTL properties by using Algorithm 6.2 as the underlying ∀CTL proof technique. The number of iterations is reported in the # column.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex. from [CK11]</td>
<td>5</td>
<td>FGp</td>
<td>2.32</td>
<td>✓</td>
<td>1.98</td>
<td>2</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL dropbuf</td>
<td>152</td>
<td>G(p ⇒ Fq)</td>
<td>53.99</td>
<td>✓</td>
<td>27.54</td>
<td>3</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Apache accept liveness</td>
<td>314</td>
<td>Gp ⇒ GFq</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>197.41</td>
<td>3</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 2</td>
<td>158</td>
<td>FGp</td>
<td>16.47</td>
<td>✓</td>
<td>52.10</td>
<td>4</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 2+bug</td>
<td>158</td>
<td>FGp</td>
<td>26.15</td>
<td>✓</td>
<td>30.37</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 3</td>
<td>14</td>
<td>FGp</td>
<td>4.21</td>
<td>✓</td>
<td>15.75</td>
<td>2</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 6</td>
<td>13</td>
<td>FGp</td>
<td>149.41</td>
<td>✓</td>
<td>59.56</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 6+bug</td>
<td>13</td>
<td>FGp</td>
<td>6.06</td>
<td>✓</td>
<td>22.12</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Windows OS fragment 8</td>
<td>181</td>
<td>FGp</td>
<td>&gt;14400.00</td>
<td>???</td>
<td>5.24</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

property where p and q are atomic propositions specific to the program. For both the tools we report the total time (in seconds) and the result for each of the benchmarks. A ✓ indicates that a tool proved the property, and χ is used to denote cases where bugs were found (and a counterexample returned). In the case that a tool exceeded the timeout threshold of 4 hours, “>14400.00” is used to represent the time, and the result is listed as “???”.

When comparing approaches on ∀CTL properties (Figure 8.2) we have chosen properties
that are equivalent in ∀CTL and LTL and then directly compared our Algorithm 6.2 to the tool in Cook et al. [CGP^07]. When comparing approaches on LTL verification problems (Figure 8.3) we have used an iterative symbolic determinization strategy in Algorithm 7.1 which uses Algorithm 6.2 on successively refined ∀CTL verification problems. The number of such iterations is given as column “#.” in Figure 8.3. For example, in the case of benchmark Windows OS fragment 3, our procedure was called twice while attempting to prove a property of the form \( FGp \).

Figure 8.4: Comparison of the times (logarithmic scale) required to prove the property of each benchmark with the previous technique [CGP^07] versus our technique. The diagonal line indicates where the tools would have the same performance. The data points in the circle on the right are cases where the previous technique timed out after 4 hours.

A visual comparison is given in Figure 8.4. Using a logarithmic scale, we compare the times required to prove the property of each benchmark with the previous technique [CGP^07] (on the x-axis) versus our technique (on the y-axis). Our tool is superior whenever a benchmark falls in the bottom-right half of the plot. Timeouts are plotted at 10,000s (seen in the circled area to the right) though they may have run much longer if we had not stopped them. Our technique was able to prove or disprove all but one example, usually in a fraction of a minute. The competing tool fails on over 25% of the benchmarks.


9.1 Summary

We have described novel methods of verifying temporal logic properties of programs. Rather than applying a general construction such as those from automata theory, we instead decompose the problem in a way that permits more tractable implementations. We began by observing that we can develop methods based on state partitioning (rather than state enumeration). This type of reasoning is characterized by a proof system in Chapter 3.

We then described an encoding that is constructed inductively over the temporal property. This encoding is a novel characterization of temporal verification as two tandem program analysis tasks: one for safety, one for liveness. With the encoding, modern abstraction and termination tools can be used directly, and we leverage their capabilities to perform what is necessary for temporal verification (e.g. backtracking, eventuality checking, tree counterexamples for branching-time properties, etc.).

The effectiveness of the above techniques is due, in part, to the fact that it uses state-based reasoning (in particular, ∀CTL). We then gave a recipe for proving trace-based properties. When approximating LTL properties with ∀CTL, we sometimes obtain an ∀CTL counterexample which is spurious to the original LTL property. This counterexample arises from the fact that there are nondeterministic choices made by the program. We give an algorithm for recognizing these choices in a ∀CTL counterexample, characterizing them as decision predicates, and introducing new hidden variables into the original program that predict the outcomes of these decisions. We showed that this symbolic, partial determinization preserves trace equivalence with the original machine. By performing this symbolic determinization iteratively, we can often prove LTL properties at speeds typically associated with state-based verification tools.

We have developed a prototype implementation of the methods in this dissertation. We show that our algorithms are able to prove trace-based properties of fragments of imperative code, drawn from software systems such as the Apache web server, the PostgreSQL...
database server, and the Windows operating system. We have proved properties which previoulsy could not be proved using existing automata-theoretic techniques.

9.2 Next steps

Our work opens the door for several important areas of future work:

Existential branching properties. We have given a method for proving LTL properties, as well as a method for proving properties in the universal fragment of Computation Tree Logic (\(\forall\text{CTL}\)). Left unexplored are extensions of our work to explore the existential fragment and, indeed, more universal logics such as \(\text{CTL}^*\) and the modal \(\mu\)-calculus.

The extension to the existential fragment is nontrivial. Our \(\forall\text{CTL}\) verification algorithm requires that we prove universality properties of guarded transition systems:

\[
\forall M, G^M \not\exists \Phi
\]

Proving this kind of behavior typically involves over-approximating the reachable states of the guarded transition system to see that the over-approximated \(G^M \not\exists \Phi\) cannot fault. For existential behaviors, we would likely need to prove properties such as

\[
\forall M, G^M \exists \Phi
\]

Here we might try to under-approximate the reachable states to find an abstraction that succeeds.

Then, of course, the two pieces must be able to fit together. We are no longer in the alternation-free fragment of \(\mu\)-calculus, and so we must combine the two types of abstraction.

Non-arithmetic and finite-state theories. Many of the algorithms in this thesis apply to general transition systems (Chapters 2, 4, 5), though some techniques have been specialized for linear arithmetic (Chapter 6, 7). We would like to explore determinization refinement and decision predicate synthesis for other theories such as bit-level verification or arrays or for programs that manipulate the heap.

We would also like to explore how our work could be extended to apply to finite-state verification algorithms. The extension to finite-state is not straight-forward. Introducing prophecy variables greatly increases the number of state-holding elements required in usual finite-state encodings: Each prophecy variable must be capable of counting up to a number larger than the system’s diameter. The problem is further exacerbated when we introduce multiple prophecy variables, as the \(n\)th prophecy variable must range over
values as large as the diameter of the system which has been augmented with the first $n-1$ prophecy variables. By contrast, when using proof tools for infinite-state systems the performance cost for adding additional infinite-state variables is usually low.

**Completeness of determinization refinement.** It is unclear whether, in general, the iterated determinization discussed in Chapter 7 terminates. We can easily show that there exists an infinite set of decision predicates, allowing us to sufficiently determinize. However, we would like to know if there exists a finite set of decision predicates that allows us to prove the property. In all practical cases (see Chapter 8) we can find such a finite set of predicates, but it is not clear that one exists in all cases. Additionally, our decision predicate synthesis method is designed strictly for linear arithmetic programs. Our choice of this domain imposes some limitations of applicability of the techniques described in this chapter. However, since determinization (Chapter 5) is sound, unsound approximations to predicate synthesis could potentially be used in instances where the systems considered do not meet these constraints.

Finally, we have presented a naïve version of the algorithm and optimizations are possible. For example, rather than iteratively beginning the proof in a ∀CTL prover from scratch, we could keep track of completed portions of the proof and re-use them.


Proofs of (de)composability

We first define some more notation and prove two lemmas before giving the proofs of (de)composability. We use the notation \((X \mapsto FO(X))\) to denote the set such that the first order formula \(FO\) holds of \(X\). We say that a given \(p \in FO\) is monotone provided that \(\forall X, Y. X \subseteq Y \Rightarrow p(X) \subseteq p(Y)\). If \(p\) is monotone, then a least fixedpoint \(\text{lf}(X \mapsto FO(X))\) exists.

**Lemma A.1** (Reachable set). For all \(I, F\), there exists a set \(S\) such that \(s' \in S\) if and only if \(\exists s \in I\) such that \((s, s') \in (\text{walk}^F_I)^*\).

*Proof.* (Coq: mk_allreach) Let \(S \equiv \text{lf}(X \mapsto I \cup \{s' | s \in X \wedge (s, s') \in \text{walk}^F_I\})\). This least fixedpoint exists because the expression is monotone. ■

The following lemma says that if a path from \(I\) never reaches the frontier, then all transitions along that path are also in \(\text{walk}^F_I\).

**Lemma A.2** (Avoiding \(F\)). For all \(R, I, F, \pi \in \Pi(S, R, I)\),

\[
\forall n \geq 0. \pi^n \notin F \Rightarrow \forall n \geq 0. \text{walk}^F_I (\pi^n_0, \pi^{n+1}_0)
\]

*Proof.* (Coq: Cktl.no_front_walk) By induction on \(n\). ■

**Definition A.1** (Frontier of \(S\)). If \(S \vdash AF\Phi\), then we denote by \(F_S\) the corresponding frontier needed to satisfy \(S \vdash AF\Phi\). That is, the greatest fixedpoint: \(\nu s. \langle R, \{s\} \rangle \vdash \Phi\).

**Proof of Decomposability Lemma 3.4.** (Coq: Cktl.cktl_subset) By induction on \(\Phi\). The AF case requires the fact that if \(R\) is well-founded, then every subset of \(R\) is also well-founded. The other cases are straight-forward. ■
Proof of Composability Lemma 3.5: \(\text{(Cog: Cktl.combine_cktl)}\) By induction on \(\Phi\).

**Cases** \(\alpha, \land\): Trivial.

**Case** \(\Phi' \lor \Phi''\): By the axiom of choice, we can choose a partitioning \(I', I''\) of \(I\) (i.e. \(I \subseteq I' \cup I''\)) such that for all \(s\), if \(\langle R, \{s\} \rangle \vdash \Phi'\) that \(s \in I'\) and otherwise \((\langle R, \{s\} \rangle \vdash \Phi'')\) \(s \in I''\). Reasoning is then straightforward, using Lemma 3.4 and ind. hyp.

**Case** \(\text{AF} \Phi'\): We use the axiom of choice to define a combined frontier (to show \(\langle R, I \rangle \vdash \text{AF} \Phi'\)) from a collection of frontiers (each \(\langle R, \{s\} \rangle \vdash \text{AF} \Phi'\)):

\[
F \equiv \{ t \mid \exists s. s \in I \land t \in F_s \} 
\]

Now \(\text{walk}_F^I\) is well-founded (by Theorem 2.5) and \(F \vdash \Phi'\) (Lemma 3.4).

**Case** \(\text{A}(\Phi' \land \Phi'')\): Once again, we use the axiom of choice to define a combined frontier:

\[
F \equiv \{ s \mid \exists s_0. s_0 \in I. s \in F_{s_0} \land \forall t, t'. \text{walk}_{F_{s_0}}^I (t, t') \Rightarrow \langle R, \{t\} \rangle \vdash \Phi' \} 
\]

Now, following the \(\text{AW}\) semantics, we must show two things:

- \(\forall t, t'. \text{walk}_F^I (t, t') \Rightarrow \langle R, \{t\} \rangle \vdash \Phi'\). By induction we show \(\exists s \in I. \text{walk}_{F_s}^I (t, t')\). We know that \(\forall s', s''. \text{walk}_{F_s}^I (s', s'') \Rightarrow \langle R, \{s'\} \rangle \vdash \Phi',\) so \(\langle R, \{t\} \rangle \vdash \Phi'\).
- \(\langle R, F \rangle \vdash \Phi''\). Follows from the definition of \(F\) above, and Lemma 3.4. \(\blacksquare\)
We now give the correctness of the encoding described in Chapter 4. We will show that if $G^M_{M,\Phi}$ does not fault, then $M \models \Phi$.

## B.1 Lemmas

Before proceeding to the main Lemma B.5, we first must prove the following lemmas.

**Lemma B.1** ($n$ decidability). For every $n_1, n_2 \in N_\Phi$, $n_1 = n_2$ is decidable.

*Proof.* Follows from the fact that both $\kappa_1 = \kappa_2$ and $\Phi_1 = \Phi_2$ are decidable. We assume that equality of atomic propositions is decidable. $\square$

**Lemma B.2** ($INV_2$). For all $\Phi, M, R, s$, there is a complete run of $G^{(S,R\{s\})}_{M,\Phi}$.

*Proof.* By induction on $\Theta_\Phi$. $\square$

The next two lemmas are needed to make our inductive argument in the main Lemma B.5. For a given guarded transition system $G$ we will need to be able to argue about the fault behavior of subsystems. This requires us to establish a homomorphic mapping between the two, the existence of which is not surprising given that $G$ is defined inductively.

**Lemma B.3** (Homomorphic mapping). For every $M = (S, R, I)$, $\forall$CTL property $\Phi$ and $(\Phi', \kappa) \in isub(\Phi)$, there exists a homomorphic mapping $H$ between $G^M_{M,\Phi'}$ and $G^M_{M,\Phi}$ such that for every pair of configurations $c, c'$,

$$
c(nd), c'(nd) \in (\{en, ex\} \times \kappa \times sub(\psi)) \land \Theta_\Phi(c)(c') \\
\Rightarrow \\
\exists d, d'. \Theta_\Phi^\kappa(d)(d') \land H(c) = d \land H(c') = d'
$$

Informally, $H$ projects out variables in $W_\Phi \setminus W_\Phi'$ and replaces $\kappa (or \overline{\kappa})$ with $\kappa$. 114
Lemma B.4 (G induction). For every $M = (S, R, I)$, $\forall$ CTL property $\Phi$ and $(\Phi', \kappa) \in isub(\Phi)$, if a configuration $c$ where $c(\text{nd}) = (\text{en}, \kappa, \Phi')$ is reachable in $G_{\mathcal{M}, \Phi}^M$, then a complete set of runs for $G_{\mathcal{M}, \Phi'}^{(S.R, c(\sigma_\kappa^c))}$ can be obtained from the set of runs of $G_{\mathcal{M}, \Phi}^M$.

Proof. By using the homomorphic mapping (Lemma B.3) and Lemma B.2.

B.2 Soundness

Lemma B.5. For every $M = (S, R, I)$, $\forall$ CTL property $\Phi$, and corresponding guarded transition system $G_{(\cdot), \Phi}^M$

\[ \exists \mathcal{M}. G_{\mathcal{M}, \Phi}^M \text{ cannot fault } \Rightarrow \langle R, I \rangle \vdash \Phi. \]

Proof. By induction on $\Phi$ and $\Theta_\Phi$, using Lemma B.4 in the inductive steps. In the base case we have that

\[ \exists \mathcal{M}. G_{\mathcal{M}, \alpha}^M \text{ cannot fault } \Rightarrow \langle R, I \rangle \vdash \alpha. \]

Here $\text{nd} \in (\{\text{en}, \text{ex}\} \times \{\epsilon\} \times \{\alpha\})$, i.e. there are only two control points. Every run begins with some $c_0 \in C_0^\alpha$, so $c_0(\text{nd}) = (\text{en}, \epsilon, \alpha)$. By the definition of $\Theta_\alpha$, there are only two classes of transitions from $(\text{en}, \epsilon, \alpha)$, both leading to $(\text{ex}, \epsilon, \alpha)$. So every run must be of length 2. Consider a run $c_1, c_2$. $C_0^\alpha(c_1)$ implies that $c_1(\sigma_\alpha) \in I$. The LHS above indicates that no run faults: $c_2(\text{fault}_\alpha^c) = \text{false}$. This must mean that $c_1(\sigma_\alpha) \in [\alpha]^S$. Consequently, $\alpha$ holds of every initial state, and so $\langle R, I \rangle \vdash \alpha$.

The proof proceeds by induction:

\[ \forall \Phi' \in isub(\Phi'). (\exists \mathcal{M}. G_{\mathcal{M}, \Phi'}^M \text{ cannot fault } \Rightarrow \langle R, I \rangle \vdash \Phi') \]
\[ \Rightarrow \exists \mathcal{M}. G_{\mathcal{M}, \Phi'}^M \text{ cannot fault } \Rightarrow \langle R, I \rangle \vdash \Phi'. \]

Notice that the hypothesis holds for all initial context path $\kappa$. This lets us use a proof of each $G_{\mathcal{M}, \Phi'}^M$ using $\epsilon$, at the next step in the induction where either $(\text{le})$ or $(\text{re})$ is used. The cases are as follows:

Case $\Phi' = \psi_1 \land \psi_2$: By the semantics of $\vdash$, we must show that $\langle R, I \rangle \vdash \psi_1$ and $\langle R, I \rangle \vdash \psi_2$. W.l.o.g. let us consider $\psi_1$. By the definition of $G_{\mathcal{M}, \psi_1 \land \psi_2}^M$, every complete run $c_0, \ldots, c_n$ is such that $C_{\psi_1 \land \psi_2}^0(c_0)$ and that $c_n(\text{fault}_\psi^c) = \text{false}$. Moreover, $c_0(\text{nd}) = (\text{en}, \epsilon, \psi_1 \land \psi_2)$ and $c_n(\text{nd}) = (\text{ex}, \epsilon, \psi_1 \land \psi_2)$.

Consider the subset of all complete runs of $G_{\mathcal{M}, \phi_1 \land \phi_2}^M$:

\[ R \equiv \{c_0, c_1, \ldots, c_{n-1}, c_n \mid c_1(\text{nd}) = (\text{en}, \text{l}, \psi_1) \land c_{n-1}(\text{nd}) = (\text{ex}, \text{l}, \psi_1) \land \Theta_{\psi_1 \land \psi_2}^\epsilon(c_0, c_1) \land \Theta_{\psi_1 \land \psi_2}^\epsilon(c_{n-1}, c_n) \land \forall i \in [1, n-2], \Theta_{\psi_1}^{\text{Le}}(c_i, c_{i+1}) \} \]
By Lemma B.4 there is a homomorphic mapping between the complete set of runs of \( G_{M,\psi_1}^M \) and \( G_{M,\psi_1 \land \psi_2}^M \). The same holds for \( G_{M,\psi_2}^M \). Since \( G_{M,\psi_1}^M \) cannot fault, and \( fault_{\psi_1 \land \psi_2} \) is given by \( fault_{\psi_1} \) in every run in \( R \), it must be the case that \( fault_{\psi_1} = false \) in each \( c_{n-1} \) of a run in \( R \). So every complete run of \( G_{M,\psi_1}^M \) cannot fault and thus (ind. hyp.) \( (R, I) \vdash \psi_1 \).

**Case \( \Phi' = \psi_1 \lor \psi_2 \):** Consider an initial state \( s_0 \in I \). By the definition of \( \Theta_{\psi_1 \lor \psi_2}^\epsilon \), we can partition the complete runs based on the initial value of \( \sigma_{\psi_1 \lor \psi_2}^\epsilon \), and then into two classes:

\[
\begin{align*}
R_{L_{s_0}} &= \{ c_0, c_1, \ldots, c_{n-1}, c_n \mid c_0(\sigma_{\psi_1 \lor \psi_2}^\epsilon) = s_0 \land \hspace{1cm} \\
c_1(\text{nd}) &= (\text{en}, \psi_1, \text{LK}) \land \hspace{1cm} \\
c_{n-1}(\text{nd}) &= (\text{ex}, \psi_1 \lor \psi_2, \kappa) \}
\end{align*}
\]

\[
\begin{align*}
R_{R_{s_0}} &= \{ c_0, c_1, \ldots, c_{n-1}, c_n, c_{n+m-1}, c_{n+m} \mid c_0(\sigma_{\psi_1 \lor \psi_2}^\epsilon) = s_0 \land \hspace{1cm} \\
c_1(\text{nd}) &= (\text{en}, \psi_1, \text{LK}) \land \hspace{1cm} \\
c_{n-1}(\text{nd}) &= (\text{ex}, \psi_1, \text{LK}) \land \hspace{1cm} \\
c_{n+m-1}(\text{nd}) &= (\text{ex}, \psi_2, \text{RK}) \land \hspace{1cm} \\
c_{n+m}(\text{nd}) &= (\text{ex}, \psi_1 \lor \psi_2, \kappa) \}
\end{align*}
\]

**Claim 1:** \( \forall \epsilon \in R_{L_{s_0}} \), \( c_{n-1}(\text{fault}_{\psi_1}^\epsilon) = false \lor \forall \epsilon \in R_{R_{s_0}} \), \( c_{n+m-1}(\text{fault}_{\psi_2}^\epsilon) = false \)

**Pf:** Assume not. \( \exists \epsilon \in R_{L_{s_0}} \), \( c_{n-1}(\text{fault}_{\psi_1}^\epsilon) = true \lor \exists \epsilon \in R_{R_{s_0}} \), \( c_{n+m-1}(\text{fault}_{\psi_2}^\epsilon) = true \). Given the RHS, there exists \( \epsilon \in R_{R_{s_0}} \) such that \( c_{n+m}(\text{fault}_{\psi_1 \lor \psi_2}^\epsilon) = true \) (def. of \( \Theta_{\psi_1 \lor \psi_2}^\epsilon \)). Contradiction (\( G_{M,\psi_1 \lor \psi_2}^M \) cannot fault).

By Claim 1 there are two cases. From the LHS we can show that \( G_{M,\psi_1}^{(S,R,\{s_0\})} \) cannot fault, and from the RHS we can show that \( G_{M,\psi_2}^{(S,R,\{s_0\})} \) cannot fault (using Lemma B.4 in each case). Hence either \( (R, \{s_0\}) \vdash \psi_1 \) or \( (R, \{s_0\}) \vdash \psi_2 \) (ind. hyp.) so \( (R, \{s_0\}) \vdash \psi_1 \lor \psi_2 \). We can use this reasoning for every \( s_0 \in I \) to obtain a partitioning of \( I \) to satisfy \( (R, I) \vdash \psi_1 \lor \psi_2 \).

**Case \( \Phi = AF_{\psi_1} \):** We must show that there exists a set \( \mathcal{F} \) such that \( \text{walk}_{\mathcal{F}}^I \) is well-founded and \( (R, \mathcal{F}) \vdash \psi_1 \). First we define the following:

\[
\mathcal{F} \equiv \{ s \mid G_{M,\psi_1}^{(S,R,\{s\})} \text{ cannot fault} \}
\]

- **Claim:** \( \text{walk}_{\mathcal{F}}^I \) is well-founded.

**Pf.** By showing there are no infinite sequences induced by \( \text{walk}_{\mathcal{F}}^I \). Assume not. Then there is an infinite sequence \( s_0, s_1, \ldots \) such that \( \forall i \geq 0. (s_i, s_{i+1}) \in \text{walk}_{\mathcal{F}}^I \).

By definition of \( \mathcal{F} \) and an inductive argument over \( \text{walk}_{\mathcal{F}}^I \), we can show that \( \forall i \geq 0. G_{M,\psi_1}^{(S,R,\{s_i\})} \) can fault. Given that \( \text{walk}_{\mathcal{F}}^I \subseteq R \), we also know that \( \forall i \geq 0. (s_i, s_{i+1}) \in R \). So by the definition of \( \Theta_{\Phi} \), we can show that there is an infinite run of \( G_{M,\text{AF}_{\psi_1}}^M \):

\[
c_0, c_0', c_0'', c_1, c_1', c_1'', \ldots \text{ s.t. } \forall i \geq 0. \hspace{1cm} \\
c_i(\sigma) = s_i \\
\land c_i(\text{nd}) = (\text{en}, \kappa, \text{AF}_{\psi_1}) \\
\land c_i'(\text{nd}) = (\text{en}, \text{LK}, \psi_1) \\
\land c_i''(\text{nd}) = (\text{ex}, \text{LK}, \psi_1)
\]
Note that for every consecutive pair of states \((s_i, s_{i+1})\), there is also a run in which \(c_{i+1}(\sigma) = s_i\) and \(c_{i+1}(\sigma) = s_{i+1}\). Since \(G_{M,AF,\psi_1}^M\) cannot fault, it must be the case that \(\forall i \geq 0. \exists f \in M. f(s_{i+1}) < f(s)\). Contradiction.

- Claim: \(\langle R, F \rangle \vdash \psi_1\).
  
  Pf. Trivial, given the definition of \(F\) and the inductive hypothesis.

**Case** \(\Phi' = A(\psi_1 W \psi_2)\): We must show that there exists a frontier \(F\) that satisfies the conditions in the AW case. First, a Lemma:

**Lemma B.6.** For every \((s_n, t) \in \text{walk}_F^I (s_n, t)\) there is a sequence \(s_0, s_1, ...\) such that \(s_0 \in I\) and \(\forall i \in [0, n]. (s_i, s_{i+1}) \in R\). Proof. By induction. □

Now we define \(F \equiv \{ s | G_{M,\psi_2}^{(S,R,\{s\})} \text{ cannot fault} \}\). What remains is to show that \(F\) satisfies the conditions of the RAW rule.

- **Claim 1:** \(\forall (s_n, t) \in \text{walk}_F^I. \langle R, \{s_n\} \rangle \vdash \psi_1\).
  
  Pf. Pick some \((s_n, t) \in \text{walk}_F^I\). By Lemma [B.6] there is some sequence of states \(s_0, ..., s_n\) such that \(s_0 \in I\) and \((s_i, s_{i+1}) \in R\).

  **Claim 1.1:** \(\forall i \in [0, n]. \langle R, \{s_i\} \rangle \vdash \psi_1\).
  
  Pf: Follows trivially from the following Claim 1.2. (Claim 1.1 cannot be proved directly by induction because the inductive hypothesis is not strong enough.)

  **Claim 1.2:** \(\forall i \in [0, n]. G_{M,\psi_1}^{(S,R,\{s_i\})} \text{ cannot fault}\).
  
  Pf: By induction on the list \(s_0, ..., s_n\), also maintaining the invariant that \((s_i, s_{i+1}) \in R\) and so

\[
\Theta_F = ... \lor (ex, \psi_1, L\kappa)\{fault_{\psi_1}^{L\kappa} \land \sigma_{A[\psi_1 W \psi_2]}^{\kappa}_A = s_i \} \xrightarrow{\delta_{A[\psi_1 W \psi_2]}^{\kappa}_{M,\psi_1} = s_{i+1}} (en, A[\psi_1 W \psi_2], \kappa)
\]

- Base case \(G_{M,\psi_1}^{(S,R,\{s_0\})} \text{ cannot fault}\):
  
  Define \(R^{s_0} = \{ c_0, ..., c_{n-2}, c_{n-1}, c_n | c_0(\sigma) = c_{n-2}(\sigma) = s_0 \land c_{n-2}(nd) = (ex, \psi_1, L\kappa) \land c_{n-1}(nd) = (en, A[\psi_1 W \psi_2], \kappa) \land c_n(nd) = (ex, A[\psi_1 W \psi_2], \kappa) \}\)

By definition of \(F\) and \(\text{walk}_F^I\), there cannot be a run in which \(c_{n-2}(\text{fault}_{\psi_1}^{L\kappa}) = \text{true}\). So from \(R^{s_0}\), we can obtain a complete set of runs for \(G_{M,\psi_1}^{(S,R,\{s_0\})}\) (Lemma [B.4]), none of which fault.
APPENDIX B. CORRECTNESS OF THE ENCODING

Induction (For $s_0, ..., s_i$, $G_{M, \psi_1}^{(S,R,(s_i))}$ cannot fault $\Rightarrow G_{M, \psi_1}^{(S,R,(s_{i+1}))}$ cannot fault): Consider the following set of runs:

$$\mathcal{R}^{s_{i+1}} = \{ c_0, ..., c_1, c_2, ..., c_4, c_5 | c_0(\sigma) = s_0 \land c_1(\sigma) = s_i \land c_2(\sigma) = s_{i+1} \land c_3(\sigma) = s_{i+1} \land c_4(\sigma) = s_{i+1} \}$$

This set is nonempty. $c_3$ is reachable because the ind. hyp. says that each $G_{M, \psi_1}^{(S,R,(s_i))}$ cannot fault and there is a transition in $\Theta$ for each $(s_i, s_{i+1})$. By definition of $\mathcal{F}$ and walk $\mathcal{F}$, there cannot be a run in which $c_3(\text{fault}^L) = \text{true}$. So from $\mathcal{R}^{s_{i+1}}$, we can obtain a complete set of runs for $G_{M, \psi_1}^{(S,R,(s_{i+1}))}$ (Lemma B.4), none of which fault.

- **Claim 2:** $(R, \mathcal{F}) \vdash \psi_2.$
  
  **Pf.** Consider the following definition:

$$\mathcal{R} = \{ c_0, ..., c_{n-1}, c_n | c_0(\sigma) = (\text{en}, A[\psi_1 W \psi_2], \kappa) \land c_{n-1}(\sigma) = (\text{ex}, \psi_2, R\kappa) \land c_n(\sigma) = (\text{ex}, A[\psi_1 W \psi_2], \kappa) \}$$

The runs in $\mathcal{R}$ cannot fault because $G_{M,A[\psi_1 W \psi_2]}^M$ cannot fault. For every $s \in \mathcal{F}$, we can obtain a complete set of runs for $G_{M, \psi_1}^{(S,R,(s_i))}$ (Lemma B.4) from $\mathcal{R}$. So by the ind. hyp. and Lemma 3.5, we find that $(R, \mathcal{F}) \vdash \psi_2.$

\[\square\]

**Proof of Theorem 4.3.** Follows from Lemma B.5 and Theorem 3.12.

### B.3 Relative completeness

**Lemma B.7** (Relative completeness). For a transition system $M = (S, R, I)$ and $\forall \text{CTL}$ property $\Phi$, and guarded transition system $G_{M,\psi_1}^M$,

$$\exists M. G_{M,\psi_1}^M \text{ cannot fault } \iff (R, I) \vdash \Phi.$$

Provided that each ranking function $f \in M$ is enumerable (e.g. represented as a possibly infinite list of state/rank pairs).
Proof. By induction on $\Phi$ and $\Theta_\Phi$, similar to the proof of Lemma \[B.5\]. Most cases are straightforward. In the the $G_{\mathcal{M}, AF\psi}^M$ case, there must not be a run $c_0, \ldots, c_n, c_{n+1}$ such that

$$c_n(\text{nd}) = (\text{ex}, \psi, \epsilon) \land c_{n+1}(\text{nd}) = (\text{ex}, \text{AF}\psi, \epsilon) \land c_{n+1}(\text{fault}_{AF\psi}^\epsilon) = \text{true}. $$

Given the definition of $\Theta_{AF\psi}$, there is only one transition which would give rise to such a run. This transition will not be taken provided that

$$\exists \mathcal{M}. \forall (s, s') \in \text{walk}_f^\mathcal{F} \exists f \in \mathcal{M}. f(s') < f(s).$$

In order for this to hold, the evaluations $f(s')$ and $f(s)$ must complete in finite time—which will happen if $f$ is enumerable. \qed

Proof of Theorem 4.4. Follows from Theorem 3.12 and Lemma B.7.
Operational semantics with a program counter

Figure [6.1] gives an operation semantics for SPL. We now modify the semantics to show how to incorporate a program counter variable \( pc \) in the state space that maps each subcommand to a unique value in the (finite) domain \( \mathcal{L} \) given by the following grammar:

\[
\mathcal{L} ::= \ell_0 \mid S \mathcal{L} \mid R \mathcal{L}
\]

For simplicity, in this dissertation we will write \( \ell_0, \ell_1, \ell_2, \ldots \) rather than clumsy names such as \( \ell_0, S\ell_0, R\ell_0 \) and so forth. The semantics is given below.

\[
\begin{align*}
\text{(skip } & ; \ C), s \rightarrow C, s[pc \mapsto Ss(pc)] & \quad \text{Skip} \\
C_1, s \rightarrow C_1', s' & \quad \text{Seq} \\
(C_1 \; ; \ C_2), s \rightarrow (C_1' \; ; \ C_2), s'[pc \mapsto Ss'(pc)] & \\
(C_1 + C_2), s \rightarrow C_1, s[pc \mapsto Ss(pc)] & \quad \text{Nd1} \\
(C_1 + C_2), s \rightarrow C_2, s[pc \mapsto Rs(pc)] & \quad \text{Nd2} \\
C^*_1 \; ; \ C_2, s \rightarrow C_2, s[pc \mapsto Rs(pc)] & \quad \text{LEx} \\
C_1, s[pc \mapsto Ss(pc)] \rightarrow^* \text{skip, s'} & \quad \text{LIt} \\
C^*_1 \; ; \ C_2, s \rightarrow C^*_1 \; ; \ C_2, s'[pc \mapsto s(pc)] & \\
\end{align*}
\]

where \( s[pc \mapsto x] \equiv \lambda v. \begin{cases} x & \text{if } v = pc, \\ s(v) & \text{otherwise} \end{cases} \)

The other commands (\( Asm \) and \( Cmd \)) do not change the value of \( pc \). Note that to use this semantics programs must first be modified such that each iterated loop \( C^* \) is replaced with \( C^* ; \text{skip} \) (this is without loss of generality).
To gain higher confidence in the detail of our work, we have proved many lemmas and theorems using the Coq proof assistant [BC04]. There are two theorems below that have been “Admitted” as axioms into Coq without a corresponding proof. Theorem Wf.well_founded is quoted from the literature, so we have not bothered to prove it. Theorem Dpred.mach_equiv is a quotation of Theorem 5.1 which we have proved by hand in Section 5.4. The proof of Theorem 5.1 is a modified version of Abadi and Lamport’s result [AL91]. Reimplementing their entire foundational proof in Coq seemed unnecessary and irrelevant to the main ideas of this thesis. Finally, some theorems and lemmas in this thesis have not been proved in Coq (for example, those in Appendix B) because it did not seem necessary.

D.1 Library boolean

Require Import Bool.

Lemma ne_true_imp_eq_false : \forall b,  
  b \neq true \rightarrow b = false.
Proof. intros. apply not_true_is_false. trivial. Qed.

Lemma negb_b_true : \forall b,  
  negb b = true \rightarrow b = false.
Proof. intros. generalize diff_true_false. intros.  
generalize (no_fixpoint_negb b). intros.  
generalize not_true_is_false. intros.  
eauto.  
Qed.

Lemma my_not_true_is_false : \forall b,  
  b = false \rightarrow b \neq true.
Proof. intros.  
generalize (bool_dec b).  
generalize diff_true_false. eauto. Qed.

Lemma neg_true_imp_eq_false : \forall b,  
  negb b = true \rightarrow b = false.
Proof. destruct b; eauto. Qed.
D.2 Library base

Require Import Arith.
Require Import Bool.
Require Import boolean.
Require Import Decidable.

Parameter state : Set.
Parameter trans : state × state → bool.
Parameter init : state → bool.

Definition atm : Set := state → bool.

Definition isSubset (sts1 sts2 : states) : Prop :=
  ∀ s, isin s sts1 → isin s sts2.

Require Import ClassicalChoice.
Require Import ClassicalFacts.

Lemma isin_choice : ∀ inits,
  ∀ s, isin s inits ∨ ¬isin s inits.
Proof. intros. set (P := isin s inits). apply classic. Qed.

Definition isEmptySet (init : states) : Prop :=
  ∀ s, ¬isin s init.

Definition init_E (s : state) : bool := false.

Lemma is_empty_init_E : isEmptySet init_E.
Proof. unfold isEmptySet. unfold isin. intros. unfold init_E. eauto. Qed.

Require Import ClassicalChoice.

Lemma equiv_init_E : ∀ init1 init2,
  isEmptySet init1 →
  isEmptySet init2 →
  init1 = init2.
Proof.
  unfold isEmptySet.
  intros.
  assert (∀ s, init1 s = init2 s).
  intros. generalize (H s). generalize (H0 s). intros.
  unfold isin in H1.
  unfold isin in H2.
  assert (init2 s = false) eauto.
  generalize (ne_true_imp_eq_false (init1 s) H2). intros.
  generalize (ne_true_imp_eq_false (init2 s) H1). intros.
  replace (init2 s). eauto.
  assert (init2 s = false) eauto.
  generalize (ne_true_imp_eq_false (init1 s) H2). intros.
  generalize (ne_true_imp_eq_false (init2 s) H1). intros.
  replace (init2 s). replace (init1 s). eauto.

Require Import FunctionalExtensionality.
apply functional_extensionality. eauto. Qed.

Lemma empty_case_split : ∀ (inits : states),
  (∀ t, inits t = false) ∨ (∃ t, inits t = true).
Proof.
  intros.
  assert (∀ t, inits t = false) ∨ ¬(∀ t, inits t = false).
  intros. set (P := ∀ t, inits t = false). apply classic.
  elim H; intros.
  left. trivial.
  right.
  apply NNPP. intuition.
  apply H0. intros.
  replace (inits t = false) with (¬ inits t = true).
  intuition.
  apply H1. trivial.
  generalize (diff_true_false). intros.
APPENDIX D. COQ PROOF SCRIPT

123
eauto.
assert(∀ b, b ≠ true → b = false); eauto.
intros.
apply ne_true_imp_eq_false. trivial.
generalize (H3 (inits t)). intros.
firstorder.
Qed.

Lemma notin_subset : ∀ x xx XX,
isSubset xx XX →
¬isin x XX →
¬isin x xx.
Proof. intros. intuition. Qed.

D.3 Library traces

Require Import Streams.
Require Import base.

CoFixpoint nsR_x x : Stream state :=
Cons x (nsR_x (proj1_sig (next_state x))).

Lemma unfold_nsRx :
∀ x,
(nsR_x x) = Cons x (tl (nsR_x x)).
Proof. intros. apply unfold_Stream. Qed.

Lemma def_nsRx :
∀ x,
(nsR_x x) = Cons x (nsR_x (proj1_sig (next_state x))).
Proof. intros. apply unfold_Stream. Qed.

Lemma hast :
∀ x, trans(( hd (nsR_x x)), (hd (tl (nsR_x x)))) = true.
Proof.
intro x.
rewrite def_nsRx.
generalize unfold_Stream.
intros. simpl.
unfold proj1_sig.
generalize (next_state x) as ns; intros. elim ns. eauto.
Qed.

Lemma mk_trace :
∀ x,
let tr0 := (nsR_x x) in
{ tr | hd tr = x ∧ tr = tr0 }.
Proof.
intros. ∃ tr0. split. eauto. eauto.
Qed.

Lemma mk_subtrace :
∀ x,
let tr0 := (nsR_x x) in
{ tr1 | tl tr0 = tr1 }.
Proof.
intros.
generalize next_state as ns; intros; elim (ns x).
intros.
generalize mk_trace as mt; intros; elim (mt x0).
intro. intros [H1 H2]. eauto.
Qed.

CoInductive isRtrace (R:state->state->Prop) : Stream state → Prop :=
| ci : ∀ tr s,
isRtrace R tr →
R s (hd tr) →
isRtrace R (Cons s tr).

Hint Resolve ci : Stream state.

Definition Rprog (s t:state) : Prop := trans( s, t ) = true.

Theorem isRtrace_nsRx :
∀ x, isRtrace nsR_x x.
Proof.
cofix CH.
intro. rewrite unfold nsRx. split. simpl. eauto. Guarded.
generalize next_state as ns; intros; elim (ns x); intros.
simpl. generalize (hast x) as ht; intros. eauto.
Qed.

CoInductive isRtraceF : (state→state→Prop) → Stream state → state → Prop :=
| itF : ∀ R tr s, isRtrace R tr s → (hd tr) = s → isRtraceF R tr s.

D.4 Library wf

Require Import base.
Require Import traces.

Inductive iswf (R : state→state→Prop) : Prop :=.

Theorem well_founded : ∀ R inits, iswf R ↔ (∀ s tr,isin s inits → not (isRtraceF R tr s)).
Proof. Admitted.

Lemma wf_notr : ∀ R inits, iswf R → (∀ s tr,isin s inits → not (isRtraceF R tr s)).

Lemma notr_wf : ∀ R inits, (∀ s tr,isin s inits → not (isRtraceF R tr s)) → iswf R.

D.5 Library actl

Require Import base.
Require Import traces.
Require Import Streams.

Inductive p : Set :=
| p_AF : p → p
| p_AG : p → p
| p_AW : p → p → p
| p_Disj : p → p → p
| p_Conj : p → p → p
| p_AtM : atm → p
with ltl : Set :=
| ltl_F : ltl → ltl
| ltl_G : ltl → ltl
| ltl_W : ltl → ltl → ltl
| ltl_Disj : ltl → ltl → ltl
| ltl_Conj : ltl → ltl → ltl
| ltl_AtM : atm → ltl.

Inductive CTL : state → p → Prop :=
| CTLap : ∀ (s : state) (atm1 : atm), atm1 s = true →
  CTL s (p_AtM atm1)
| CTLand : ∀ (s : state) (p1 p2 : p),
  CTL s p1 →
  CTL s p2 →
  CTL s (p_Conj p1 p2)
| CTLor : ∀ (s : state) (p1 p2 : p),
  (CTL s p1 ∨ CTL s p2) →
  CTL s (p_Disj p1 p2)
| CTLag : ∀ (s : state) (p1 : p),
  (∀ tr, isRtraceF Rprog tr s →
  (∀ n, CTL (Str_nth n tr) p1)) →
APPENDIX D. COQ PROOF SCRIPT

\[CTL\ s\ (p_{AG} p1)\]
\[|\ CTLaF : \forall \ (s:state)\ (p1:p),\]
\[\ (\ \forall \ tr,\]
\[\ \text{isRtraceF} \ Rprog\ tr\ s \rightarrow\]
\[\ (\exists\ n,\ CTL\ (\Str\ nth\ n\ tr)\ p1) \rightarrow\]
\[\ CTL\ s\ (p_{AF} p1)\]
\[|\ CTLaF : \forall \ (s:state)\ (p1\ p2:p),\]
\[\ (\ \forall \ tr,\]
\[\ \text{isRtraceF} \ Rprog\ tr\ s \rightarrow\]
\[\ (\ \forall\ n,\ CTL\ (\Str\ nth\ n\ tr)\ p1) \lor\]
\[\ (\exists\ n,\ CTL\ (\Str\ nth\ n\ tr)\ p1\ p2 \land \ \forall\ m,\ m < n \rightarrow\ CTL\ (\Str\ nth\ m\ tr)\ p1 ))\rightarrow\]
\[\ CTL\ s\ (p_{AW} p1\ p2).\]

Require Import List.

Fixpoint subf (ctlp:p) : list p :=
match ctlp with
| p_Atм _ => nil
| p_Disj p1 p2 => ctlp :: ((subf p1)++(subf p2))
| p_Conj p1 p2 => ctlp :: ((subf p1)++(subf p2))
| p_AF p1 => ctlp :: (subf p1)
| p_AG p1 => ctlp :: (subf p1)
| p_AW p1 p2 => ctlp :: ((subf p1)++(subf p2))
end.

D.6 Library cktl

Require Import base.
Require Import actl.
Require Import traces.
Require Import wf.
Require Import boolean.
Require Import Bool.
Require Import Streams.
Require Import Classical.
Require Import ClassicalFacts.

Inductive Reach : state \rightarrow state \rightarrow nat \rightarrow Prop :=
| ReachRefl : \forall \ s,\]
\[\ Reach\ s\ s\ 0\]
| ReachTrans : \forall \ (st1 st3 st2:state)\ n,\]
\[\ Reach\ st1\ st2\ n \rightarrow\]
\[\ trans\ (\ st2 , \ st3 ) = true \rightarrow\]
\[\ Reach\ st1\ st3\ (S\ n).\]

Inductive Walk (init front:states) : state \rightarrow state \rightarrow Prop :=
| WalkBase : \forall \ (st1 st2:state),\]
\[\ Rprog\ st1\ st2 \rightarrow\]
\[\ not\ (isin\ st1\ front) \rightarrow\]
\[\ isin\ st1\ init \rightarrow\]
\[\ Walk\ init\ front\ st1\ st2\]
| WalkFront : \forall \ (st1 st3 st2:state),\]
\[\ Rprog\ st2\ st3 \rightarrow\]
\[\ not\ (isin\ st2\ front) \rightarrow\]
\[\ Walk\ init\ front\ st1\ st2 \rightarrow\]
\[\ Walk\ init\ front\ st2\ st3.\]

Inductive ReachFrom : states \rightarrow state \rightarrow Prop :=
| rf : \forall \ init,\]
\[\ (\exists\ s,\ isin\ s\ init \land \exists\ n,\ Reach\ s\ t\ n) \rightarrow\ ReachFrom\ init\ t.\]

Inductive Walkn (init front:states) : nat \rightarrow state \rightarrow state \rightarrow Prop :=
| WalknBase : \forall \ (st1 st2:state),\]
\[\ Rprog\ st1\ st2 \rightarrow\]
\[\ not\ (isin\ st1\ front) \rightarrow\]
\[\ isin\ st1\ init \rightarrow\]
\[\ Walkn\ init\ front\ 0\ st1\ st2\]
| WalknFront : \forall \ n\ (st1 st3 st2:state),\]
\[\ Rprog\ st2\ st3 \rightarrow\]
Appendix D. Coq Proof Script

Lemma walkn_imp_walk : \forall inits front, \forall s t. Walk inits front s t \rightarrow \exists n. Walk inits front n s t.
Proof.
  induction 1.
  \exists 0. apply WalkBase; eauto.
  firstorder. \exists (S x). eapply WalkFront; eauto.
Qed.

Lemma walk_imp_walkn : \forall inits front, \forall s t n. Walk inits front n s t \rightarrow Walk inits front s t.
Proof.
  induction 1. apply WalkBase; eauto.
  eapply WalkFront; eauto.
Qed.

Lemma walkn_imp_initE : \forall n inits front s t, isEmptySet inits \rightarrow (\neg Walk inits front n s t).
Proof.
  induction n. firstorder.
  intuition. inversion H0.
  firstorder.
  intuition.
  inversion H0. subst.
  generalize (IHn inits front st1 s H). intro.
  eauto.
Qed.

Lemma walk_initE : \forall n inits front s t, isEmptySet inits \rightarrow (\neg Walk inits front n s t).
Proof.
  generalize walkn_imp_initE;
  generalize walkn_imp_walk;
  generalize walk_imp_walkn;
  eauto. firstorder.
  intuition.
  generalize (H0 inits front s t H3). intros.
  elim H4; intro n Hn.
  generalize (H1 n inits front s t H2 Hn). eauto. Qed.

Lemma false_ne_true : false = true \rightarrow False.
Proof. intros. contradict H. eauto. Qed.

Axiom state_eq : \forall (a b:state), \{a=b\}+\{a<>b\}.

Lemma state_is_fin : \forall (s t:state), \{s=t\} \lor \{s<>t\}.
Proof.
  intros. generalize (state_eq s t) as H; intro H.
  elim H; eauto. Qed.

Definition mk_singleton (s:state) : state \rightarrow bool :=
  fun t, match state_eq s t with
  \mid left _ \Rightarrow true
  \mid right _ \Rightarrow false
  end.

Inductive CKTL : states \rightarrow p \rightarrow Prop :=
  \mid CKTLap : \forall (init:states) (atm1:atm),
  \forall s, isin s init \rightarrow atm1 s = true \rightarrow CKTL init (p_Atm atm1)
  \mid CKTLand : \forall (init:states) (p1 p2:p),
  CKTL init p1 \rightarrow CKTL init p2 \rightarrow CKTL init (p_Conj p1 p2)
  \mid CKTlor : \forall (init:states) (p1 p2:p) (init2 init3:states),
  \forall s, isin s init \rightarrow (\{isin s init2\}+\{isin s init3\}) \rightarrow CKTL init2 p1 \rightarrow CKTL init3 p2 \rightarrow CKTL init (p_Disj p1 p2)
\[\text{Lemma tr\_hd\_trans\_gen} : \forall R,\]
\[\forall tr,\]
\[\text{isRtrace R tr} \rightarrow \]
\[R (\text{hd tr}) (\text{tl (tl tr)})].\]

**Proof.** intros. inversion H. simpl. trivial. Qed.

\[\text{Lemma tr\_nth\_tr\_gen} : \forall R,\]
\[\forall n, \forall tr,\]
\[\text{isRtrace R tr} \rightarrow \]
\[\text{tr} (\text{Str\_nth\_tl (S n) tr})].\]

**Proof.**
induction n.
intros. inversion H. eauto.
intros. inversion H.
replace (\text{Str\_nth\_tl (S (S n)) (Cons s tr0)})
with (\text{Str\_nth\_tl (S n) tr0}).
apply IHn. eauto. eauto. Qed.

\[\text{Lemma tr\_nth\_trans\_gen} : \forall R,\]
\[\forall tr n,\]
\[\text{isRtrace R tr} \rightarrow \]
\[R (\text{Str\_nth (S n) tr}) (\text{Str\_nth (S n) tr})].\]

**Proof.**
intros.
induction n.
replace (\text{Str\_nth 0 tr}) with (\text{hd tr}).
replace (\text{Str\_nth 1 tr}) with (\text{hd (tl tr)}).
also replace (\text{Str\_nth (S (S n) tr}) with (\text{hd (Str\_nth\_tl (S n) tr)})).
replace (\text{Str\_nth (S (S n) tr}) with (\text{hd (tl (Str\_nth\_tl (S n) tr)})� 
set (\text{trp := (Str\_nth\_tl (S n) tr)}).
generalize (\text{tr\_nth\_tr\_gen R n tr H}).
intro. apply (\text{tr\_hd\_trans\_gen R trp}). eauto.
replace (\text{tl (Str\_nth\_tl (S n) tr}) with (\text{Str\_nth\_tl (S (S n)) tr}).
eaauto generalize tl\_nth\_tl; intros.
replace (\text{tl (Str\_nth\_tl (S n) tr}) with (\text{Str\_nth\_tl (S n) (tl tr)}).
eaauto eauto. eauto. Qed.

\[\text{Lemma tr\_hd\_trans} :\]
\[\forall tr,\]
\[\text{isRtrace Rprog tr} \rightarrow \]
\[\text{trans (hd tr), (hd (tl tr))) = true.}\]

**Proof.** intros. inversion H. simpl. trivial. Qed.

\[\text{Lemma tr\_nth\_tr} :\]
\[\forall n, \forall tr,\]
\[\text{isRtrace Rprog tr} \rightarrow \]
\[\text{isRtrace Rprog (Str\_nth\_tl (S n) tr)}.\]

**Proof.**
induction n.
intros. inversion H. eauto.
intros. inversion H.
replace (\text{Str\_nth\_tl (S (S n)) (Cons s tr0)})
with (\textit{Str\_nth\_tl} (\textit{S n}) tr0).
apply \textit{IHn} eauto eauto.
Qed.

\textbf{Lemma \textit{tr\_nth\_trans}} :
\forall tr n,
isRtrace \textit{Rprog} tr \rightarrow
\textit{trans} (\textit{Str\_nth} n tr), (\textit{Str\_nth} (\textit{S n}) tr) = true.
\textbf{Proof}.
intros.
induction \textit{n}.
replace (\textit{Str\_nth} 0 tr) with (\textit{hd} tr).
replace (\textit{Str\_nth} 1 tr) with (\textit{hd} (\textit{tl} tr)).
apply \textit{tr\_hd\_trans}. trivial eauto eauto.
replace (\textit{Str\_nth} (\textit{S n}) tr) with (\textit{hd} (\textit{Str\_nth\_tl} (\textit{S n}) tr)).
replace (\textit{Str\_nth} (\textit{S n}) tr) with (\textit{hd} (\textit{tl} (\textit{Str\_nth\_tl} (\textit{S n}) tr)))).
set \textit{(tr := (\textit{Str\_nth\_tl} (\textit{S n}) tr))}.
generalize (\textit{tr\_nth\_tr n tr H}).
intro apply (\textit{tr\_hd\_trans trp}). eauto.
replace (\textit{tl} (\textit{Str\_nth\_tl} (\textit{S n}) tr)) with (\textit{Str\_nth\_tl} (\textit{S n}) tr).
eauto generalize \textit{tl\_nth\_tl}; intros.
replace (\textit{tl} (\textit{Str\_nth\_tl} (\textit{S n}) tr)) with (\textit{Str\_nth\_tl} (\textit{S n}) (\textit{tl} tr)).
eauto eauto.
Qed.

\textbf{Lemma \textit{can\_reach}} :
\forall \textit{n}, \forall \textit{tr s},
isRtraceF \textit{Rprog} tr s \rightarrow
\textit{Reach s} (\textit{Str\_nth n tr}) \textit{n}.
\textbf{Proof}.
induction \textit{n}.
intro s. replace (\textit{Str\_nth} 0 tr) with \textit{s}. apply \textit{ReachRefl}. inversion \textit{H}. eauto.
intros.
set \textit{(u := (\textit{Str\_nth n tr}))}.
set \textit{(t := (\textit{Str\_nth S n tr}))}.
apply (\textit{ReachTrans s t u}).
unfold \textit{u}. apply \textit{IHn}. trivial.
eauto apply \textit{tr\_nth\_trans}. inversion \textit{H}. trivial.
Qed.

\textbf{Lemma \textit{sing\_implies\_isin}} :
\forall \textit{t}, \isin t \textit{1} (\textit{mk\_singleton} \textit{t})
\textbf{Proof}.
intros unfold \isin. compute. elim \textit{state\_eq}; eauto. Qed.

\textbf{Theorem \textit{soundness}} :
\forall \textit{p1}.
\forall \textit{init}.
CKTL \textit{init} \textit{p1} \rightarrow
(\forall \textit{s}, \isin s \isin \textit{init} \rightarrow \textit{CTL} \textit{s p1}).
\textbf{Proof}.
induction \textit{p1}.
intro s. inversion \textit{H}. subst.
generalize \textit{CTLaf} as \textit{af}; intros.
apply \textit{af}.
intro \textit{tr G1}.
generalize (\textit{wf\_notr} (\textit{Walk init front}) \textit{init} \textit{H2 s tr}) as \textit{notr}; intros.
contradiction.
intro s. inversion \textit{H}. subst.
generalize \textit{CTLag} as \textit{ag}; intros. apply \textit{ag}. intros \textit{tr J1 n}.
aply (\textit{IHp1 s s \textit{H}_4 (Str\_nth n tr))}.
aply (\textit{H2 (Str\_nth n tr)}).
aply \textit{rf} \exists \textit{s}. split. trivial. \exists \textit{n}. apply \textit{can\_reach n tr s}. trivial.
firstorder.
inversion \textit{H}. generalize \textit{H3 s}. intros.
aply \textit{CTLaw}. subst.
intro \textit{tr Htr}.
generalize \textit{(H5 tr H0 Htr)}. intros.
elim \textit{H1}; intros; clear \textit{H1}. left.
intro \textit{n}.
aply (\textit{IHp1 \textit{I} (mk\_singleton (Str\_nth n tr)))}. eauto.
aply \textit{sing\_implies\_isin}. right.
elim H2, intros n [G1 G2].
exists n, split.
apply (IHp1_2 (mk_sing (Str_nth n tr))). eauto.
apply sing_implies_isin.
intros m Hm.
generalize (G2 m Hm). intros.
apply (IHp1_1 (mk_sing (Str_nth m tr))). trivial.
apply sing_implies_isin.
intros. inversion H. subst.
subst. generalize (H3 s H0). intro CH.
clear H0.
apply CTLor.
intro. left. apply (IHp1_1 init2). trivial. trivial.
intro. right. apply (IHp1_2 init3). trivial. trivial.
intros. inversion H. subst. apply CTLand. eauto. eauto.
intros. inversion H. subst. apply CTLap. eauto.
Qed.

Lemma sing_iff_eq : \forall s t, isin t (mk_sing s) ↔ s = t.
Proof.
intros s t. unfold isin. compute.
elim state_eq. eauto. intros.
generalize false_ne_true as Hft. intro.
split; contradiction.
Qed.

Lemma eq_implies_sing : \forall s t, s = t → isin t (mk_sing s).
Proof.
generalize eq_implies_isin as H1.
generalize eq_implies_eq as H2.
generalize eq_implies_sing as H3.
intros. unfold singleton. split. eauto.
Qed.

Definition singleton (s : state) (init : states) : Prop :=
isin s init ∧ \forall t, s = t ↔ isin t init.

Lemma sing_is_singleton : \forall s,
singleton s (mk_sing s).
Proof.
generalize sing_implies_isin as H1.
generalize sing_implies_eq as H2.
generalize eq_implies_sing as H3.
intros. unfold singleton. split. eauto.
Qed.

Lemma biimp_implies_not : \forall P Q,
(P ↔ Q) → ¬P → ¬Q.
Proof. intros. intuition. Qed.

Lemma sing_implies_notin : \forall s ss t,
singleton s ss →
t ∉ s →
¬isin t ss.
Proof.
intros. unfold singleton in H.
generalize H; intros [H1 H2].
generalize (H2 t). intros.
apply (biimp_implies_not (s=t) (isin t ss)). trivial.
eauto. Qed.

Lemma sing_eq_sing : \forall s ss,
singleton s ss →
mk_sing s = ss.
Proof.
intros.
assert(∀ t, ss t = (mk_sing s) t).
intro.
assert([t=s]+{t<>s}). apply state_eq.
elim H0; intros; clear H0.
subst.
replace (ss s) with true.
replace ((mk_singleton s) s) with true.
eauto.
generalize (sing_implies_isin s). intro. unfold isin in H0. symmetry. eauto.
unfold singleton in H.
generalize H; intros [H1 H2].
generalize (H2 s). intros [H3 H4]. eauto.
replace (ss t) with false.
replace ((mk_singleton s) t) with false.
eauto.
generalize sing_implies_isin s.
intro.
unfold isin in H0.
symmetry.
eauto.
unfold singleton in H.
generalize H; intros [H1 H2].
generalize (H2 s). intros. eauto.
replace (ss t) with false.
replace ((mk_singleton s) t) with false.
eauto.
gerelax mk singleton.
symmetry. elim state_eq. intros. intuition in a. firstorder. eauto.
generalize (sing_implies_notin s ss t H).
generalize (H0 b). intros. unfold isin in H1.
symmetry.
apply ne_true_imp_eq_false.
trivial.
Require Import FunctionalExtensionality.
apply functional_extensionality.
eauto.
Qed.

Lemma singleton_subset :
\forall s init a,
isin s init \rightarrow
isSubset init a.
Proof.
unfoldsingleton.
intros s init a [H1 H2] H3.
unfold isSubset.
generalize sing.is_singleton as H4.
intros. elim (H2 s0). intros.
genralize (H5 H).
itros. subst. trivial.
Qed.

Require Import ClassicalChoice.
Lemma is_choice :
\forall P : state \rightarrow bool \rightarrow Prop,
(\forall s, \exists b, P s b)
\rightarrow
\exists f. (\forall s, P s (f s)).
Proof. apply choice. Qed.

Definition b_reach (inits:state->bool) : state \rightarrow bool \rightarrow Prop :=
fun t b \Rightarrow
match b with
| true \Rightarrow ReachFrom inits t
| false \Rightarrow \neg ReachFrom inits t
end.

Lemma mk_allreach :
\forall inits.
\exists front. \forall t, b_reach inits t (front t).
Proof. intros.
avert(\forall t, \exists b, b_reach inits t b).
itros. unfold b_reach.
assert(ReachFrom inits t \lor \neg ReachFrom inits t).
set (P:=ReachFrom inits t). apply classic.
elim H; introts. clear H.
set(b1:=true). \exists b1. eauto.
set(b1:=false). \exists b1. eauto.
apply choice.
eauto. Qed.

Lemma escape_front :
\forall tr inits front n,
isin (Str_nth n tr) front \rightarrow
\neg Walk inits front (Str_nth n tr) (Str_nth (S n) tr).
Proof. intros.
induction n.
intuition.
inversion H0. subst. eauto. subst. eauto.
inversion H0. subst. eauto. subst. eauto.
Qed.

Definition isn frontier (s:state) (p1:p) (t:state) : Prop :=
\[(\exists \, tr, \exists \, n,\quad t = (\text{Str}_{\text{n}} n \, tr) \land \text{isRtraceF} \, \text{Rprop} \, tr \, s \land \text{CTL} (\text{Str}_{\text{n}} n \, tr) \, p1).\]

**Definition** \(Pimp \, (s:\text{state}) \, (p1:p) \, (t:\text{state}) \, (P:\text{Stream} \, \text{state} \rightarrow \text{Prop} \rightarrow \text{nat} \rightarrow \text{Prop}) : \text{Prop} := \exists \, tr, \exists \, n, \quad P \, tr \, (\text{isRtraceF} \, \text{Rprop} \, tr \, s) \, n.\)

**Axiom** \(\text{front_imp_dec} : \forall \, s \, p1 \, t \, P, \quad \{(Pimp \, s \, p1 \, t \, P)\} + \{(Pimp \, s \, p1 \, t \, P)\}.\)

**Definition** \(\text{frontier} \, (s:\text{state}) \, (p1:p) \, (P:\text{Stream} \, \text{state} \rightarrow \text{Prop} \rightarrow \text{nat} \rightarrow \text{Prop}) : \text{state} \rightarrow \text{bool} := \)

\[
\text{fun} \, t \Rightarrow \text{match} \, (\text{front_imp_dec} \, s \, p1 \, t \, P) \, \text{with} \\
| \text{left} \_ \Rightarrow \text{true} \\
| \text{right} \_ \Rightarrow \text{false} \\
\text{end}.\]

**Axiom** \(\text{fff} : \forall \, s \, p1 \, t, \quad \{(\text{isin_frontier} \, s \, p1 \, t)\} + \{(\text{isin_frontier} \, s \, p1 \, t)\}.\)

**Lemma** \(\text{Walk_has_Rprop} : \quad \forall \, \text{init front}, \quad \forall \, s \, t, \text{Walk} \, \text{init front} \, s \, t \rightarrow \text{trans}(s, t) = \text{true}.\)

**Proof.** intros. induction \(H.\) trivial. inversion \(H.\) trivial. Qed.

**Lemma** \(\text{walk_is_R}2 : \quad \forall \, tr \, \text{init front}, \quad \text{isRtrace} \, (\text{Walk} \, \text{init front}) \, tr \rightarrow \text{isRtraceF} \, \text{Rprop} \, tr.\)

**Proof.** cofix \(CH.\) intros.

inversion \(H.\) split. apply \((CH \, tr0 \, \text{init front}).\) trivial.

inversion \(H.\) inversion \(H1.\) trivial. trivial. Qed.

**Lemma** \(\text{walk_is_RF} : \forall \, \text{inits front} \, s, \quad \forall \, tr, \quad \text{isRtraceF} \, (\text{Walk} \, \text{init front}) \, tr \, s \rightarrow \text{isRtraceF} \, \text{Rprop} \, tr \, s.\)

**Proof.** intros. inversion \(H.\) apply \(itF.\)

apply \((\text{walk_is_R}2 \, tr \, \text{inits front}).\) trivial. trivial. Qed.

**Lemma** \(\text{walk_is_R} : \quad \forall \, s \, tr \, \text{init front}, \quad \text{singleton} \, s \, \text{init} \rightarrow \text{isRtraceF} \, (\text{Walk} \, \text{init front}) \, tr \, s \rightarrow \text{isRtraceF} \, \text{Rprop} \, tr \, s.\)

**Proof.** intros. split. apply \((\text{walk_is_R}2 \, tr \, \text{init front}).\)

inversion \(H0.\) trivial. inversion \(H0.\) trivial. Qed.

**Lemma** \(\text{reach_step} : \forall \, s \, t, \quad \text{Rprop} \, s \, t \rightarrow \exists \, tr, \text{isRtraceF} \, \text{Rprop} \, tr \, s \land t = (\text{Str}_{\text{n}} n \, 1 \, tr).\)

**Proof.** intros.

assert\((\exists \, tr0, \text{isRtraceF} \, \text{Rprop} \, tr0 \, t).\)

generalize \((mk_{\text{trace}} \, t).\) intros. elim \(H0.\) intros \(tr0 \, G1 \, G2.\) \(\exists \, tr0.\)

apply \(itF.\) generalize \((\text{isRtraceF_{nxt}} \, t).\) as \(HR.\) intro. replace \(tr0.\) trivial. trivial.

elim \(H0.\)

intros \(tr0 \, H1.\)

set\((tr1 := \text{Cons} \, s \, tr0).\)

\(\exists \, tr1.\) split. unfold \(tr1.\)

apply \(itF.\) apply cong. inversion \(H1.\) trivial.

replace \((\text{hd} \, tr0) \text{ with} \, t.\) trivial. inversion \(H1.\) symmetry. trivial. simpl. trivial.

unfold \(tr1.\) inversion \(H1.\) eauto.

Qed.

**Definition** \(\text{isRtraceFTn} \quad \text{(R:state} \rightarrow \text{state} \rightarrow \text{Prop} \text{) \, (tr:Stream} \, \text{state} \text{) \, (s \, t: \text{state}) \, (n: \text{nat}) : \text{Prop} := \text{isRtraceF} \, R \, tr \, s \land t = (\text{Str}_{\text{n}} n \, tr).\)

**Lemma** \(\text{isRFTn_imp_head} : \forall \, R \, s \, t \, tr \, n, \quad \text{isRtraceFTn} \, R \, tr \, s \, t \, n \rightarrow \text{hd} \, tr = s.\)

**Proof.** intros. inversion \(H.\) inversion \(H0.\) trivial. Qed.

**Lemma** \(\text{isRfnt_same} : \forall \, tr \, s \, t,\)
isRtraceFTn Rprog tr s t 0 → s = t.
Proof. intros. inversion H. inversion H0. replace s. replace t. subst. eauto. Qed.

Fixpoint glue_streams (n: nat) (tr: Stream state) (tr’: Stream state) : Stream state :=
match n with
| 0 ⇒ Cons (hd tr) tr’
| S m ⇒ (Cons (hd tr) (glue_streams m (tl tr) tr’))
end.

Lemma glue_unfold : ∀ m tr tr’,
(glue_streams (S m) tr tr’) = (Cons (hd tr) (glue_streams m (tl tr) tr’)).
Proof. induction m. intros. eauto. intros. eauto. Qed.

Lemma glue_unfold3 : ∀ m tt tr tr’,
Cons (hd tt) (glue_streams m (tl tt) tr’) =
(glue_streams m tt (Cons (Str_nth (S m) tt) tr’)).
Proof. induction m. intros. eauto. intros. simpl.
generalize glue_unfold as Hgu. intro.
replace (glue_streams m (tl tt) (Cons (Str_nth (S m) tt) tr’))
with (Cons (hd (tl tt)) (glue_streams m (tl (tl tt)) tr’)). trivial.
apply IHm. Qed.

Lemma glue_unfold2 : ∀ m tt tr tr’,
(glue_streams (S m) tr tr’) = (glue_streams m tr (Cons (Str_nth (S m) tr) tr’)).
Proof. induction m. intros. simpl. eauto.
intros. simpl.
replace (glue_streams m (tl tr) (Cons (Str_nth (S m) tr) tr’))
with (Cons (hd (tl tr)) (glue_streams m (tl (tl tr)) tr’)). trivial.
set (tt := (tl tr)).
replace (Str_nth (S (S m)) tr) with (Str_nth (S m) tt).
apply glue_unfold3. unfold tt. eauto. Qed.

Lemma glue_help : ∀ m tr tr’,
isRtrace Rprog tr →
isRtrace Rprog tr’ →
(Rprog (Str_nth (S m) tr) (hd tr’)) →
isRtrace Rprog (Cons (Str_nth (S m) tr) tr’).
Proof. induction m. intros. apply ci. trivial. trivial.
intros. apply ci; trivial. Qed.

Lemma glue_isRtrace : ∀ m, ∀ tr tr’,
isRtrace Rprog tr →
isRtrace Rprog tr’ →
(Rprog (Str_nth m tr) (hd tr’)) →
isRtrace Rprog (glue_streams m tr tr’).
Proof. induction m.
intros tr tr’ H1 H2. unfold glue_streams. intros.
apply ci. trivial. eauto.
intros.
generalize (glue_unfold m tr tr’) as Hgu. intro.
generalize (glue_unfold2 m tr’ tr’) as Hg2. intro.
replace (glue Streams (S m) tr tr’)
with (glue_streams m tr (Cons (Str_nth (S m) tr) tr’)).
apply (IHm tr (Cons (Str_nth (S m) tr) tr’)). trivial.
apply glue_help; trivial. simpl.
apply tr_nth_trans. trivial. Qed.

Lemma glue_head : ∀ n tr tr’,
hd (glue_streams n tr tr’) = hd tr.
Proof. induction n; intros; eauto. Qed.

Lemma glue_nth : ∀ n tr tr’,
(Cons (Str_nth (S n) tr) (glue_streams n tr tr’)) = (hd tr’).
Proof. induction n; intros; eauto. simpl. eapply IHn. Qed.

Lemma reach_imp_trace : ∀ n, ∀ s t,
Reach s t n →
∃ tr, isRtraceFTn Rprog tr s t n.
Proof.
induction n.
intros.
generalize (mk_trace s). intro Hm. elim Hm. intros tr [K1 K2]. ∃ tr.
unfold isRtraceFTn. split. eapply itF. generalize (isRtrace nsrx s).
intro. replace tr. trivial. trivial.
assert(s=t). inversion H. eauto. replace t. eauto.
intros.

inversion H.
generalize (IHn s st2 H1) as IHc. intro. elim IHc. intros tr0 Htr0. subst.
generalize (mk_trace t). intro Hm. elim Hm. intros tt [K1 K2].
set (TT:=(glue_streams n tr0 tt)).
existsTT. unfold TT.
unfold isRtraceFTn. split.
apply ifF.
apply glue_isRtrace. inversion Htr0. inversion H0. trivial.
replace tt. apply (isRtrace_neq t).
replace (hd tt).

inversion Htr0. replace (Str_nth n tr0) with st2. eauto.
inversion Htr0.
replace (hd (glue_streams n tr0 tt)) with (hd tr0).

inversion H0. eauto. symmetry. apply glue_head.
replace (Str_nth (S n) (glue_streams n tr0 tt)) with (hd tt).
symmetry. trivial. symmetry. apply glue_nth.
Qed.

Lemma wf_trace_has_n : \forall front s,
(\forall tr,
  isRtraceF Rprog tr s \rightarrow
  \exists n,isin (Str_nth n tr) \rightarrow
  iswf (Walk (mk_singleton s) front)).
Proof.
intras.
generalize (notr_wf (Walk (mk_singleton s) front) (mk_singleton s)) as wf. intros.
apply wf. intros.
assert(s=s0).
apply (sing_implies_eq s s0).
trivial.
symmetry.
intras.
assert(isRtraceF Rprog tr s0 \lor ~isRtraceF Rprog tr s0) as Hc.
set (P:=isRtraceF Rprog tr s0). apply classic.
elim Hc; intros; clear Hc.
assert(isin s0 (mk_singleton s0) \lor ~isin s0 (mk_singleton s0)).
apply classic.
elim H2; intros; clear H2.
genralize (H tr H1). intros.
elim H2; intros n Hn.
genralize (escape_front tr (mk_singleton s0) front n Hn). intros.
genralize (tr_nth_trans_gen (Walk (mk_singleton s0) front) tr n). intros.
intuition.
inversion H6. eauto.
intuition.
genralize (walk_is_RF (mk_singleton s0) front s0 tr). intros. eauto.
Qed.

Lemma iswf_subset : \forall inits subinits front,
iswf (Walk inits front) \rightarrow
isSubset subinits inits \rightarrow
iswf (Walk subinits front).
Proof.
intras.
genralize (wf_notr (Walk inits front) inits H). intros.
apply (notr_wf (Walk subinits front) subinits).
intras s tr.
genralize (H1 s tr). intros.
intuition. Qed.

Lemma iswf_initE : \forall inits front,
isEmptySet inits \rightarrow
iswf (Walk inits front).
Proof. intros.
apply (notr_wf (Walk inits front) inits).
intras.
intuition.
assert(~isin (hd tr) initE); eauto.

unfold isin. unfold initE. generalize diff_true_false. intros. eauto.
inversion H1. subst.
inversion H3. subst.
assert(∀ u v, ¬ Walk inits front u v).
  intros.
  generalize (walk_initE inits front u v H). eauto.
  generalize (H6 s (bd tr0)). intros. eauto.
Qed.

Lemma initE_trivial : ∀ p1, CKTL initE p1.
Proof.
  generalize ies_initE as Hies.
  intros. induction p1; eauto.
  apply (CKTLaf initE p1 initE); trivial.
  apply (isuf_initE); trivial.
  apply (CKTLag initE p1 initE).
  intros. inversion H. elim H0. intros s [A1 A2]. subst.
  contradict A1. unfold isin. unfold initE. eauto. trivial.
  apply (CKTLor initE p1_1 p1_2 initE initE). intros. left. trivial. trivial. trivial.
  apply (CKTLand initE p1_1 p1_2); trivial.
  apply (CKTLap). intros. contradict H. unfold isin. unfold initE. eauto.
Qed.

Lemma ckl_subset : ∀ p1 inits s, CKTL inits p1 → isin s inits →
CKTL (mk_singleton s) p1.
Proof. induction p1; intros; inversion H; subst.
apply (CKTLaf (mk_singleton s) p1 (ckl_front s)).
apply (isuf_subset inits (mk_singleton s) front). trivial.
unfold isSubset. intros.
assert(s0=s). generalize (sing_implies_eq s). intro G1. generalize (G1 s0).
  intros. symmetry. eauto. subst. trivial. trivial. trivial.
  apply (CKTLag (mk_singleton s) p1 s02).
  intro t. generalize (H2 t). intro G1. intro G2.
  apply G1. inversion G2. elim H1. intros s0 [K1 K2]. subst.
  assert(s0=s). generalize (sing_implies_eq s). intro G3. generalize (G3 s0).
  intros. symmetry. eauto. subst.
  apply rf s split; eauto. trivial.
  apply (CKTLaw (mk_singleton s) p1 l p1 l_2).
  intros. generalize (H3 s tr).
  assert(s0=s). generalize (sing_implies_eq s). intro G3. generalize (G3 s0).
  intros. symmetry. eauto. subst.
  intros.
  generalize (H4 H0 H2).
  intros.
  elim H5. intros; clear H5.
  left. eauto.
  right. eauto.
  generalize (H3 s H0) as G1. intro. elim G1; intro.
  apply (CKTLor (mk_singleton s) p1_1 p1 l_2 (mk_singleton s) initE). intros; eauto.
  apply (H lp l_1 (mk_singleton s) s). eauto. apply (sing_implies_isin).
  (initE_trivial).
  apply (CKTLor (mk_singleton s) p1_1 p1_2 initE (mk_singleton s)). intros; eauto.
  apply (initE_trivial).
  apply (H lp l_2 (mk_singleton s) s). eauto. apply (sing_implies_isin).
  (CKTLand (mk_singleton s) p1_1 p1 l_2).
  apply (H lp l_1 inits s H4 H0).
  apply (H lp l_2 inits s H5 H0).
  apply (CKTLap (mk_singleton s)).
  intros s0 G1.
  assert(s0=s). generalize (sing_implies_eq s). intro G2. generalize (G2 s0).
  intros. symmetry. eauto. subst. apply H3. trivial.
Qed.

Require Import Classical_Pred_Type.

Definition use_choice (p1 p2:p) (inits:state->bool) : state → bool → Prop :=
  fun s b ⇒
  match b with
  | true ⇒ isin s inits → CKTL (mk_singleton s) p1
  | false ⇒ isin s inits → CKTL (mk_singleton s) p2
end.
Lemma is_choice2 : \forall inits p1 p2,  
\( (\forall s, \exists b, use_choice p1 p2 inits s b) \rightarrow \exists f, (\forall s, use_choice p1 p2 inits s (f s)). \)  
Proof. intros. apply is_choice. eauto. Qed.

Lemma must_choose : \forall ch:state->bool,  
\( \forall s, \{ch s = true\} + \{ch s = false\} \).  
Proof. intros. generalize (bool_dec (ch s) true). intros. elim H; intros; clear H. eauto.  
right. generalize (not_true_is_false (ch s)). eauto. Qed.

Definition t_in_s_front (p1:p) (inits:state->bool) : state \rightarrow Prop :=  
fun t =>  
exists s, \exists front s,  
isin s inits \land  
isuf (Walk (mk_singleton s) front s) \land  
CKTL front s p1 \land  
isin t front s.

Lemma choice_t_in_s_front : \forall t p1 inits,  
t_in_s_front p1 inits t \lor \neg (t_in_s_front p1 inits t).  
Proof. intros. generalize (t_in_s_front p1 inits t). apply classic. Qed.

Definition front_choice (p1:p) (inits:state->bool) : state \rightarrow bool \rightarrow Prop :=  
fun t b =>  
match b with  
| true \Rightarrow t_in_s_front p1 inits t  
| false \Rightarrow \neg t_in_s_front p1 inits t  
end.

Lemma front_choice2 : \forall inits p1,  
(\forall s, \exists b, front_choice p1 inits s b) \rightarrow \exists f, (\forall s, front_choice p1 inits s (f s)).  
Proof. intros. apply is_choice. trivial. Qed.

Lemma trivial_CKTL :  
\forall inits p1, isEmptySet inits \rightarrow CKTL inits p1.  
Proof. intros. induction p1.  
apply (CKTLaf inits p1 inits).  
apply (isuf_initE inits inits). trivial. trivial.  
apply (CKTLag inits p1 inits).  
intros. inversion H0. subst.  
elim H1. intros s [G1 G2].  
unfold isEmptySet in H.  
generalize (H s). intros. firstorder. trivial.  
apply (CKTLaw inits p1_1 p1_2); eauto. intros.  
firstorder.  
apply (CKTLor inits p1_1 p1_2 inits inits); eauto.  
apply (CKTLand inits p1_1 p1_2); eauto.  
apply CKTLap. intros.  
unfold isEmptySet in H.  
generalize (H s). intros. firstorder.  
Qed.

Lemma combine_ckt :  
\forall p1 init,  
(\forall t,isin t init \rightarrow CKTL (mk_singleton t) p1) \rightarrow CKTL init p1.  
Proof. induction p1; intros.  
generalize (empty_case_split init) as H. intros.  
elim H; clear H; intros.  
apply (CKTLaf init p1 initE).  
assert(isEmptySet init).  
unfold isEmptySet. intros.  
generalize(H0 s). intros. unfold isin.  
apply my_not_true_is_false. trivial.  
apply isuf_initE. trivial.  
apply initE_trivial.
assert(∀ s, ∃ b, front_choice p1 init s b) as HM.

intro t. unfold front_choice.

generalize (choice t in s front t p1 init) as Hch. intros.

elim Hch; intros.

set(b1:=true). ∃ b1. replace b1 with true.

unfold t in s front in H1.

elim H1. intros s Hs. elim Hs. intros front_s. intros [G1 [G2 [G3 G4]]].

∃ s. ∃ front_s. split. eauto.

split. trivial. split; trivial. eauto.

set(b1:=false). ∃ b1. replace b1 with false. trivial. eauto.

generalize (front_choice2 init p1 HM as HM1. intros.

elem HM1. intros fronts Hf.

apply (CKTLag init p1 fronts).

apply (notr_wf (Walk init fronts) init).

intros s tr Hs.

generalize (H s Hs). intro. inversion H1. subst.

generalize (uf_notr (Walk (mk_singleton s) front)). intros.

generalize (H2 init H3 s tr Hs). intro.

intuition.

apply (IHp1 fronts). intros s Hs.

generalize (Hf s). intros.

unfold front_choice in H1.

unfold in s front in H1.

elim H1. intros s0 Hs0. elim Hs0. intros front_s [G1 [G2 [G3 G4]]].

apply (ckt_subset p1 front_s s); eauto.

generalize (mk_allreach init) as HA; intros; elim HA; intros front G1.

apply (CKTLag init p1 front).

intros t G2.

generalize (G1 t) as G3. intro.

unfold b_reach in G3.

intuition.

assert(∀ t, isin t front ∨ ~isin t front).

intro. apply classic.

generalize (H0 t) as H1; intros. elim H1; intros; clear H1. trivial.

unfold isin in H2.

replace (front t) with false in G3. intuition.

symmetry.

generalize diff_true_false. intros.

apply ne_true_imp_eq_false. eauto.

apply IHp1.

intros t H1.

generalize (G1 t). intros G2.

unfold b_reach in G2. replace (front t) with true in G2.

inversion G2. elim H0. intros s [K1 K2]. subst.

generalize (H s K1) as G3. intros.

inversion G3. subst.

assert (ReachFrom (mk_singleton s) t).

split. ∃ s. split; eauto. apply (sing_implies_isin).

generalize (H2 t H1). intros.

apply (ckt_subset p1 st s2 t); trivial.

generalize (empty_case_split init) as Hc. intros.

elim Hc. intros.

apply (CKTLaw init p1_1 p1_2).

intros.

generalize (H0 s). intro. unfold isin in H1.

clear H. clear Hc. clear H0. clear IHp1.1. clear IHp1.2.

generalize (diff_true_false). intros.

intuition. replace (init s) with false in H1.

symmetry in H1. fistorder.

clear Hc. intro.

apply (CKTLaw init p1_1 p1_2).

intro s.

generalize (H s) as G1. intro.

assert(isin s init ∨ ~isin s init). apply classic.

elem H1; intro Hs. clear H1.

generalize (G1 Hs) as G2.

intro.

inversion G2. subst.
generalize \((H3 \ s)\). intro. clear \(H3\).
intros tr \(Hs1\) \(Htr\).
assert(\(isin\ s\ (mk\_singleton\ s))\). apply sing\_implies\_isin.
generalize \((H1\ tr\ H2\ Htr)\) as \(Hch\). intro.
elim \(Hch\); intros; clear \(Hch\).
left. intro. generalize \((H3\ n)\). intro. trivial.
right. elim \(H3\). intros \(a\ [K1\ K2]\). \(n\) split; eauto.
intros. intuition.
generalize \((empty\_case\_split\ init)\) as \(Hc\). intros.
elim \(Hc\); intros.
apply \((CKTLor\ init\ p1\_1\ p1\_2\ initE\ initE)\).
intros. generalize \((H0\ s)\). intros.
unfold isin. eauto. apply initE\_trivial. apply initE\_trivial.
clear \(Hc\).
assert(\(\forall\ s, 3\ b, use\_choice\ p1\_1\ p1\_2\ init\ s\ b\) as \(Ha\).
intros. unfold use\_choice.
generalize \((H\ s)\). intros.
generalize \((isin\_choice\ init)\) as \(H2\). intros.
generalize \((H2\ s)\). intros.
elim \(H3\).
intros. generalize \((H1\ H4)\). intros. inversion \(H5\). subst.
generalize \((H8\ s)\). intros.
generalize \((sin\_implies\_isin\ s)\) as \(Hsi\). intros.
generalize \((H6\ Hsi)\). intros.
elim \(H7\); intros.
set(b1 := true). \(3\ b1\). simpl. intros.
apply \((cktl\_subset\ p1\_1\ init2\ s)\). trivial. trivial.
set(b1 := false). \(3\ b1\). simpl. intros.
apply \((cktl\_subset\ p1\_2\ init3\ s)\). trivial. trivial.
intros.
set(b1 := true). \(3\ b1\). simpl. contradiction.
generalize \((is\_choice2\ init\ p1\_1\ p1\_2\ Ha)\). intros.
elim \(H1\). intros. choice \(Hch\).
set(initL := fun \(x \Rightarrow init\ x\ &\& choice\ x\))
set(initR := fun \(x \Rightarrow init\ x\ &\& negb\ (choice\ x))\).
apply \((CKTLor\ init\ p1\_1\ p1\_2\ initL\ initR)\).
intros \(s\ Hs\).
generalize \((must\_choose\ choice\ s)\) as \(Hmc\). intros.
elim \(Hmc\); intros.
left. unfold isin. unfold initL.
generalize (andb\_true\_iff (init\ s) (choice\ s)). intros \[G1\ G2\].
apply \(G2\). eauto.
right. unfold isin. unfold initR. replace (choice\ s).
generalize (andb\_true\_iff (init\ s) (negb false)). intros \[G1\ G2\].
apply \(G2\). eauto.
apply (IHpl\_1\ initL). intros \(s\ Hs\).
assert(\(isin\ s\ init\)) as \(Hs0\). unfold isin.
unfold isin in \(Hs\).
unfold initL in \(Hs\).
generalize (andb\_true\_iff (init\ s) (choice\ s)). intros.
elim \(H2\). intros \(G5\ G6\).
generalize \((G5\ Hs)\). intros \[G7\ G8\]. trivial.
generalize \((Hch\ s)\) as \(G1\). unfold use\_choice. intros.
assert(choice\ s = true).
unfold initL in \(Hs\).
unfold isin in \(Hs\).
generalize (andb\_true\_iff (init\ s) (choice\ s)). intros.
elim \(H2\). intros \(G5\ G6\).
generalize \((G5\ Hs)\). intros \[G7\ G8\]. trivial.
replace (choice\ s) with true in \(G1\). eauto.
apply (IHpl\_2\ initR).
intros.
generalize \((Hch\ t)\). intros. unfold use\_choice in \(H3\).
unfold initR in \(H2\). unfold isin in \(H2\).
apply andb\_true\_iff in \(H2\). generalize \(H2\). intros \[K1\ K2\].
assert(choice\ t = false). apply negb\_b\_true. trivial.
replace (choice\ t) in \(H3\).
unfold isin in \(H3\).
apply (H3 K1).
apply (CKTLand init p1_1 p1_2).
apply IHp1_1. intros t Hi. generalize (H t Hi) as G1. intro.
inversion G1. trivial.
apply IHp1_2. intros t Hi. generalize (H t Hi) as G1. intro.
inversion G1. trivial.
apply CKTLap. intros s Hi.
generalize (H s Hi) as G1. intro. inversion G1. subst.
apply H2. apply sing_implies_isin.
Qed.

Theorem completeness :
\forall p1 init,
\forall s,
\quad \seconds p1 \rightarrow
\quad \{\text{singleton s init} \} \rightarrow
\quad \CKTL \init p1.

Proof.
induction p1.
intros. inversion H. subst.
set (front := (\fun t \Rightarrow
\quad \{\text{fff \ s \ p1 \ t} \} \text{ with }
\quad \left\{ \begin{array}{l}
\text{left} \Rightarrow \text{true} \\
\text{right} \Rightarrow \text{false}
\end{array} \right.
\quad \text{end})).
apply (CKTLap init p1 front).
generalize (wf_trace_has_n front s). intros.
replace init with (mk_singleton s). apply H1.
unfold singleton in H0. generalize H0. intros [Hs A1].
intros tr J1.
generalize (H3 tr J1). intro K3.
elim K3. intros n K5 \exists n.
unfold isin.
assert (isin_frontier s p1 (Str \_nth n tr)).
unfold isin_frontier. \exists tr \exists n.
split. trivial. split. trivial. trivial.
unfold front. elim fff. intro. trivial.
intro. contradiction.
generalize (sing_is_singleton s). intros.
apply (sing_eq_sing s init). trivial.
apply (combineckt1 p1 front). intros t G1.
apply (IHp1 (mk_singleton t) t).
assert (front t = \text{true}). eauto.
assert (isin_frontier s p1 t).
Focus 2.
unfold isin_frontier in H2.
elim H2. intros tr K1. elim K1. intros n [K2 [K3 K4]].
replace t. trivial.
unfold isin_frontier.
unfold front in H1.
unfold front in G1.
assert (front t = \text{true}). eauto.
assert (isin_frontier s p1 t).
Focus 2. unfold isin_frontier in H4.
elim H4. intros tr L1. elim L1. intros n [L2 [L3 L4]].
\exists tr \exists n. split. trivial. split. trivial. trivial. trivial.
generalize H1. elim fff. intros. trivial.
intros. contradict H4. eauto.
apply sing_is_singleton.
intros.
generalize (mk_allreach init) as HA; intros; elim HA; intros front G1.
assert (\forall \text{tr n,}
\quad \text{isRtraceF Rprog tr s} \Rightarrow
\quad \CKTL (\text{mk_singleton (Str nth n tr)}) \text{ p1}) \rightarrow
\quad \CKTL \text{ front p1}.
Focus 2.
apply (CKTLag init p1 front). eauto.
assert (\forall t, ReachFrom init t \rightarrow \text{isin t front}).
intros t K1. generalize (G1 t). intro GG.
unfold b_reach in GG.
assert(∀ t, isin t front → ∃ t front).
intro. apply classic.
generalize (H2 t) as KK; intros. elim KK; intros; clear KK. trivial.
unfold isin in H3.
replace (front t) with false in GG. intuition.
symmetry.
apply ne_true_false. intros.
eauto.
eauto.
apply H1.
intros.
apply (IHp1 (mk_singleton (Str_nth n tr)) (Str_nth n tr)).
inversion H. subst. apply H5. trivial.
apply sing_is_singleton.
intros.
assert(∀ t, isin t front → ReachFrom init t).
intros t K1.
generalize (G1 t). intros. unfold b_reach in H2.
replace (front t) with true in H2. trivial.
apply (combine_cktl p1 front).
intros t K1.
apply (H3 t K1) as G4. intro.
inversion G4. elim H3. subst.
intros s0 [L1 L2].
assert(s=s0). unfold singleton in H0. elim H0.
intros M1 M2. elim (M2 s0). eauto.
apply (sing_is_singleton).
intros init s HCTL Hsing. inversion HCTL. subst.
apply (CKTLaw init p1_1 p1_2).
intros.
assert(s=s0). unfold singleton in Hsing. elim Hsing.
intros M1 M2. elim (M2 s0). eauto.
apply (CKTLaw init p1_1 p1_2).
intros.
apply (CKTLor init p1_1 p1_2 initE). intros. left. trivial.
apply (IHp1_1 init s). trivial. trivial.
apply (trivial_CKTL initiE p1_2). trivial. apply ies_initE.
apply (CKTLor init p1_1 p1_2 initE init). intros. right. trivial.
apply (trivial_CKTL initiE p1_1). trivial. apply ies_initE.
apply (IHp1_2 init s). trivial. trivial.
inversion H1.
intro. apply (CKTLap init a). inversion Hsing.
inversion HCTL. subst.
apply (CKTLap init a). inversion Hsing.
inversion HCTL. subst.
apply (singleton_subset s init a). trivial.
D.7 Library mach

Require Import boolean.
Require Import base.
Definition pstate : Set := nat->nat.
Record Mach (X:Type) : Type := mkMach { ms : X->bool; mr : X*X->bool; mi : X->bool; mproj : X->state }.
Require Import Streams.
CoInductive isMtrace {X:Type} (m:Mach X) : Stream X → Prop :=
| ismt : ∀ tr s, isMtrace m tr → (mr X m) (s,(hd tr)) = true →
  isMtrace m (Cons s tr).

Definition isMtraceF {X:Type} (m:Mach X) (tr:Stream X) (s:X) : Prop :=
  isMtrace m tr ∧ (hd tr) = s.

Definition isMtraceI {X:Type} (m:Mach X) (tr:Stream X) : Prop :=
  (mi X m) (hd tr) = true ∧ isMtrace m tr.

D.8 Library mctl

Require Import base.
Require Import Streams.
Require import mach.
Inductive ctl : Type :=
| ctl_AF : ctl → ctl
| ctl_AG : ctl → ctl
| ctl_RW : ctl → ctl → ctl
| ctl_Disj : ctl → ctl → ctl
| ctl_Conj : ctl → ctl → ctl
| ctl_Atm : (state->bool) → ctl.

Inductive CTLm {X:Type} (m:Mach X) : X → ctl → Prop :=
| CTLmap : ∀ (s:X) (ap:state->bool),
  ap ((mproj X m) s) = true →
    CTLm m s (ctl_Atm ap)
| CTLmand : ∀ (s:X) (p1 p2:ctl),
  CTLm m s p1 →
  CTLm m s p2 →
  CTLm m s (ctl_Conj p1 p2)
| CTLmor : ∀ (s:X) (p1 p2:ctl),
  (CTLm m s p1 ∨ CTLm m s p2) →
  CTLm m s (ctl_Disj p1 p2)
| CTLmag : ∀ (s:X) (p1:ctl),
  (∀ tr,
    isMtraceF m tr s →
    (∀ n, CTLm m (Str_nth n tr) p1)) →
    CTLm m s (ctl_AG p1)
| CTLmaf : ∀ (s:X) (p1:ctl),
  (∀ tr,
    isMtraceF m tr s →
    (∃ n, CTLm m (Str_nth n tr) p1)) →
    CTLm m s (ctl_AF p1)
| CTLmaw : ∀ (s:X) (p1 p2:ctl),
  (∀ tr,

isMtraceF m tr s →
( (∀ n, CTLm m (Str_nth n tr) p1) ∨
(3 n, CTLm m (Str_nth n tr) p2 ∧ ∀ n', n' < n → CTLm m (Str nth n' tr) p1)) →
CTLm m s (ctl_AW p1 p2).

Definition CTL {X:Type} (m:Mach X) (ctlp:ctl) : Prop :=
∀ s, (mi X m) s = true → CTLm m s ctlp.

D.9 Library ltl

Require Import Bool.
Require Import base.
Require Import mach.
Require Import mctl.
Require Import Streams.

Inductive ltl : Type :=
| ltl_F : ltl → ltl
| ltl_G : ltl → ltl
| ltl_W : ltl → ltl → ltl
| ltl_Disj : ltl → ltl → ltl
| ltl_Conj : ltl → ltl → ltl
| ltl_Atm : (state->bool) → ltl.

Inductive LTLm {X:Type} (m:Mach X) : Stream X → ltl → Prop :=
| LTLmap : ∀ (tr:Stream X) (atm1:state->bool),
(∀ i, LTLm m (str_nth_tl i tr) p1) →
LTLm m tr (ltl_Atm atm1)
| LTLmap1 : ∀ (tr:Stream X) (p1 p2:ltl),
LTLm m tr p1 →
LTLm m tr p2 →
LTLm m tr (ltl_Conj p1 p2)
| LTLmorA : ∀ (tr:Stream X) (p1 p2:ltl),
LTLm m tr p1 →
LTLm m tr (ltl_Disj p1 p2)
| LTLmorB : ∀ (tr:Stream X) (p1 p2:ltl),
LTLm m tr p2 →
LTLm m tr (ltl_Disj p1 p2)
| LTLmg : ∀ (tr:Stream X) (p1:ltl),
(∀ i, LTLm m (str_nth_tl i tr) p1) →
LTLm m tr (ltl_G p1)
| LTLmf : ∀ (tr:Stream X) (p1:ltl),
(∃ i, LTLm m (str_nth_tl i tr) p1) →
LTLm m tr (ltl_F p1)
| LTLmwG : ∀ (tr:Stream X) (p1 p2:ltl),
LTLm m tr (ltl_G p1) →
LTLm m tr (ltl_W p1 p2)
| LTLmwF : ∀ (tr:Stream X) (p1 p2:ltl),
(∃ i,
(∃ j, j < i → LTLm m (str_nth_tl j tr) p1)) →
LTLm m tr (ltl_W p1 p2).

Definition LTL {X:Type} (m:Mach X) (ltlp:ltl) : Prop :=
∀ s tr,
(mi X m) s = true →
isMtraceF m tr s →
LTLm m tr ltlp.

Inductive Matcher : ctl → ltl → Prop :=
| m_Atm : ∀ (p:state->bool),
Matcher (ctl_Atm p) (ltl_Atm p)
| m_F : ∀ ctlp ltlp,
Matcher (ctl_AW ctlp) (ltl_F ltlp)
| m_G : ∀ ctlp ltlp,
Matcher (ctl_AF ctlp) (ltl_G ltlp)
| m_W : ∀ ctlp ltlp,
Matcher (ctl_AW ctlp) (ltl_W ltlp)
Matcher (ctl_AW ctlp) (ltl_W ltlp) →
Matcher (ctl\_AG ctlp) (ltl\_G ltlp)
\[ m\_W : \forall \; ctlp1\; ltlp1\; ctlp2\; ltlp2, \\
\quad \text{Matcher}\; ctlp1\; ltlp1 \rightarrow \\
\quad \text{Matcher}\; ctlp2\; ltlp2 \rightarrow \\
\quad \text{Matcher}\; (ctl\_AW\; ctlp1\; ctlp2)\; (ltl\_W\; ltlp1\; ltlp2) \]
\[ m\_Disj : \forall \; ctlp1\; ltlp1\; ctlp2\; ltlp2, \\
\quad \text{Matcher}\; ctlp1\; ltlp1 \rightarrow \\
\quad \text{Matcher}\; ctlp2\; ltlp2 \rightarrow \\
\quad \text{Matcher}\; (ctl\_Disj\; ctlp1\; ctlp2)\; (ltl\_Disj\; ltlp1\; ltlp2) \]
\[ m\_Conj : \forall \; ctlp1\; ltlp1\; ctlp2\; ltlp2, \\
\quad \text{Matcher}\; ctlp1\; ltlp1 \rightarrow \\
\quad \text{Matcher}\; ctlp2\; ltlp2 \rightarrow \\
\quad \text{Matcher}\; (ctl\_Conj\; ctlp1\; ctlp2)\; (ltl\_Conj\; ltlp1\; ltlp2) \]

Lemma bigger_is_mtrace : \forall \; (X:\text{Type}) \; m \; s \; s' \; tr, \\
\quad isMtraceF\; m\; tr\; s' \rightarrow \\
\quad isMtrace\; m\; (\text{Cons}\; s\; tr) \; s.
Proof.
intros. split. inversion H.
eapply. eauto. replace (hd\; tr) \; with \; s'; eauto. eauto.
Qed.

Lemma bigger_is_mtrace2 : \forall \; (X:\text{Type}) \; m \; s \; tr, \\
\quad isMtrace\; m\; tr \rightarrow \\
\quad isMtrace\; m\; (\text{Cons}\; s\; tr).
Proof.
intros. split. inversion H.
eapply. eauto. eauto.
eauto.
Qed.

Lemma tl_is_rtrace : \forall \; (X:\text{Type}) \; (m:\text{Mach}\; X) \; tr, \\
\quad isMtrace\; m\; tr \rightarrow \\
\quad isMtrace\; m\; (tl\; tr).
Proof. cofix CH.
inros. inversion H. eauto. Qed.

Lemma nth_is_rtrace : \forall \; (X:\text{Type}) \; n \; (m:\text{Mach}\; X) \; tr, \\
\quad isMtrace\; m\; tr \rightarrow \\
\quad isMtrace\; m\; (\text{Str}_{\text{nth}}\; tl\; n\; tr).
Proof.
induction n. eauto.
inros.
generalize. (tl_is_rtrace\; X\; m\; (\text{Str}_{\text{nth}}\; tl\; n\; tr)).
inros.
assert. ((\text{Str}_{\text{nth}}\; tl\; (S\; n)\; tr) = (tl\; (\text{Str}_{\text{nth}}\; tl\; n\; tr))).
simpl. generalize. (tl\_nth\_tl\; n\; tr).
eauto. eauto.
replace. (\text{Str}_{\text{nth}}\; tl\; (S\; n)\; tr).
eauto.
Qed.

Theorem ltl_imp_actl : \forall \; (X:\text{Type}) \; (m:\text{Mach}\; X) \; ltlp\; ctlp, \\
\quad Matcher\; ctlp\; ltlp \rightarrow \\
\quad \forall\; s, \\
\quad CTLm\; m\; s\; ctlp \rightarrow \\
\quad (\forall\; tr, \text{isMtraceF}\; m\; tr\; s \rightarrow LTLm\; m\; tr\; ltlp).
Proof.
inros. intro X m ltlp ctlp.
induction 1. intro s Hc tr Histr.
eapply. LTLmap.
inversion Hc.
inversion Histr. subst. eauto.
apply. LTLmf.
inversion Hc.
generalize. (H2\; tr\; Histr) as G2. intro.
elim. G2. intro n G3.
exists n. intro.
inversion Histr.
generalize. (nth_is_rtrace\; X\; m\; tr) as Hsuf.
inros.
generalize. (IHMatcher\; (\text{Str}_{\text{nth}}\; n\; tr)\; G3\; (\text{Str}_{\text{nth}}\; tl\; n\; tr)).
inros. apply. H5.
split. eauto.
apply. LTLmg.
inversion Hc. subst. intro n.
generalize. (H2\; tr\; Histr\; n) as G1. intro.
(unfold inverison ltl apply Ispdecn mach
Require Import Record ltl Require Import List Streams
apply LTLlmsG. apply LTLmg.
intro n. generalize (H1 n) as G1. intros.
generalize (nth_is_rtrace X n m tr) as Hsuf. intros.
generalize (IHMatcher1 (Str_nth n tr) G1 (Str_nth_tl n tr)) as G2.
intro s. apply G2. split; eauto. apply Hsuf. inversion Histr. eauto.
imerson Hc. subst.
generalize (H3 tr Histr) as Hcase. intro.
elim Hcase; intros; clear Hcase; clear H3.
apply LTLLsigr. apply LTLmg.
intro n. generalize (H1 n) as G1. intros.
generalize (nth_is_rtrace X n m tr) as Hsuf. intros.
generalize (IHMatcher2 (Str_nth n tr) G1 (Str_nth_tl n tr)) as G3. intro.
apply G3. split; eauto. apply Hsuf. inversion Histr. eauto.
imerson j Hj.
generalize (G2 j Hj) as G4. intros.
apply (IHMatcher1 (Str_nth j tr) G4 (Str_nth_tl j tr)).
split; eauto.
inversion Histr. apply (nth_is_rtrace X j m tr H2).
imerson Hc. subst. elim H3; clear H3; intros.
apply LTLLmsrA.
apply (IHMatcher1 s H1 tr Histr).
apply LTLLsrrB.
applay (IHMatcher2 s H1 tr Histr).
applay LTLLlma; inversion Hc; subst.
applay (IHMatcher1 s H4 tr Histr).
applay (IHMatcher2 s H5 tr Histr).
Qed.

Set Printing All.
Theorem m_ltl_imp_actl : ∀ (X:Type) (m:Mach X) ltlp ctp.
Matcher ctp ltlp →
CTL m ctp →
LTL m ltlp. Proof.
unfold CTL; unfold LTL; intros.
eapply ltl_imp_actl . eauto. eauto. eauto.
Qed.

D.10 Library dpred

Require Import base.
Require Import boolean.
Require Import mct.
Require Import mach.
Require Import lit.
Require Import List.
Require Import mach.
Require Import Streams.

Record Dpreds : Set := mkDpreds
{ dp : list ((state->bool)→(state->bool)) }.

Definition Ispdecn (p p':pstate) (aidx:nat) (n:nat) : Prop :=
∀ i, IF i = aidx then p' i = n else p' i = p i.

Definition PredDet (apred bpred:state->bool) (s s':state) (p p':pstate) (aidx:nat) : Prop :=
(apred s = true →
(match (p aidx) with
| O ⇒
| S O ⇒
| S n ⇒
| (bpred s' = true ∧ Ispdecn p p' aidx n) end) ∧
(apred s = false → True).

Inductive RisDet (s s':state) (p p':pstate) :

APPENDIX D. COQ PROOF SCRIPT

\textbf{Lemma} \texttt{mk_trace} : \forall (X:\text{Type}) (m:\text{Mach} X),
\{ tr | \text{hd} tr = a \land tr = tr0 \}.

\textbf{Proof.}
\texttt{intros. \exists tr0. split. eauto. eauto. Qed.}

\textbf{Lemma} \texttt{unfold_nsr_a} :
\forall (X:\text{Type}) (a:X) m,
(nsR_a a m a) = Cons a (tl (nsR_a m a a)).

\textbf{Proof.}
\texttt{intros. apply unfold_Stream. Qed.}

\textbf{Lemma} \texttt{hdStrip_ishdproj} :
\forall X \text{ tr m}, \text{hd} (\text{trStrip m tr}) = mproj X m \text{ (hd tr)}.

\textbf{Proof.}
\texttt{intros. eauto. Qed.}

\textbf{Lemma} \texttt{def_nsr} :
\forall (X:\text{Type}) (a:X) (m:\text{Mach} X),
(nsR_a m a) = Cons a (nsR_a m (proj1sig (nextM m X m a))).

\textbf{Proof.}
\texttt{intros. apply unfold_Stream. Qed.}

\textbf{Lemma} \texttt{hast_a} :
\forall (X:\text{Type}) (a:X) (m:\text{Mach} X),
\text{mr} X m \text{ (hd (nsR_a m a a)), (hd (tl (nsR_a m a a)) ) = true.}

\textbf{Proof.}
\texttt{intros X a m.}
\texttt{rewrite def_nsr.}
\texttt{generalize unfold_Stream.}
\texttt{intros. simpl.}
\texttt{unfold proj1sig.}
\texttt{generalize (nextM m X m a) as ns; intros. elim ns. eauto. Qed.}

\textbf{Unset Printing All.}

\textbf{Lemma} \texttt{nsR_a_isMtrace} : \forall (X:\text{Type}) (a:X) m,
isMtrace m (nsR_a m a).

\textbf{Proof.}
cofix CH.
\texttt{intros X a m. rewrite unfold_nsr_a. split. simpl. eauto.}
\texttt{generalize nextM m a as ns; intros; elim (ns X m a); intros.}
\texttt{simpl. generalize (hast_a X a m) as h; intros. eauto. Qed.}

\textbf{Lemma} \texttt{nsR_a_MtraceI} : \forall (X:\text{Type}) (a:X) m,
(mi X m) a = true \rightarrow
\text{isMtraceI m (nsR_a m m a)}.

\textbf{Proof.}
\texttt{intros X a m.}
\texttt{unfold isMtraceI. split. eauto.}
Lemma hd_tr_in_l : \forall \ (X : Type) \ m tr,
   isMtraceI m tr \rightarrow (\forall X \ m) (hd tr) = true.
Proof. intros unfold isMtraceI in H. firstorder. Qed.

Lemma Eq_tl : \forall \ (X : Type) \ (tr1 tr2 : Stream X),
   EqSt tr1 tr2 \rightarrow EqSt (tl tr1) (tl tr2).
Proof. cofix CH. split. inversion H. inversion H1. trivial.
   inversion H. apply (CH X (tl tr1) (tl tr2)). trivial. Qed.

Lemma tl_outside : \forall \ (X : Type) \ (m : Mach X) tr,
   tl (tr_strip m (tl tr)) = tl (tr_strip m tr).
Proof. generalize Eq_tl.
   generalize tl_outside. eauto. Qed.

Lemma asdf : \forall \ (X : Type) \ (m : Mach X) tr,
   hd (tr_strip m (tl tr)) = hd (tl (tr_strip m tr)).
Proof. generalize Eq_tl.
   eauto. Qed.

Lemma asdf2 : \forall \ (X : Type) \ (m : Mach X) tr,
   tl (tr_strip m tr) = tr_strip m (tl tr).
Proof. generalize Eq_tl.
   eauto. Qed.

Lemma tl_eq : \forall \ (X Y : Type) \ (m1 : Mach X) \ (m2 : Mach Y) tr1 tr2,
   EqSt (tr_strip m1 tr1) (tr_strip m2 tr2) \rightarrow
   EqSt (tl (tr_strip m1 tr1)) (tl (tr_strip m2 tr2)).
Proof. generalize Eq_tl. eauto. Qed.

Lemma tl_eq_n : \forall \ (X Y : Type) \ (m1 : Mach X) \ (m2 : Mach Y) n,
   \forall tr1 tr2,
   EqSt (tr_strip m1 tr1) (tr_strip m2 tr2) \rightarrow
   EqSt (tr_strip m1 (Str_nth_tl n tr1)) (tr_strip m2 (Str_nth_tl n tr2)).
Proof. intros X Y m1 m2. induction n ; intros.
   simpl. trivial.
   generalize Eq_tl. intro.
   simpl.
   replace (Str_nth_tl n (tl tr1)) with (tl (Str_nth_tl n tr1)).
   replace (Str_nth_tl n (tl tr2)) with (tl (Str_nth_tl n tr2)).
   replace (tr_strip m1 (tl (Str_nth_tl n tr1)))
     with (tl (tr_strip m1 (Str_nth_tl n tr1))).
   replace (tr_strip m2 (tl (Str_nth_tl n tr2)))
     with (tl (tr_strip m2 (Str_nth_tl n tr2))).
   eapply tl_eq. eauto.
   eauto. eauto. apply tl_nth_tl. apply tl_nth_tl. Qed.

Lemma equiv_ltl_tr : \forall \ (X Y : Type) \ (m1 : Mach X) \ (m2 : Mach Y) ltlp,
   \forall tr ie1 tr ie2,
   isMtrace m1 tr ie1 \rightarrow
   isMtrace m2 tr ie2 \rightarrow
   EqSt (tr_strip m1 tr ie1) (tr_strip m2 tr ie2) \rightarrow
   LTLm m1 tr ie1 ltlp \rightarrow
   LTLm m2 tr ie2 ltlp.
Proof. intros X Y m1 m2 ltlp.
   generalize nth_is_trace as H1rt. intro.
   induction ltlp ; intros tr ie1 tr ie2 Hist1 Hist2 Ht1 ltlp1 ; inversion Ht1.
   eapply LTLmf. elim H1. intros n Hn. \exists n. n.
APPENDIX D. COQ PROOF SCRIPT

apply (IHltlp (Str_nth_tl n tr_ie1) (Str_nth_tl n tr_ie2)); eauto.
apply tl_eq_n. trivial.
apply LTLmor. intro n. generalize (H1 n) as G1. intro.
apply (IHltlp1 (Str_nth_tl n tr_ie1) (Str_nth_tl n tr_ie2)); eauto.
apply tl_eq_n. trivial.
apply LTLmorG.
apply LTLmor. intro n. inversion H1. generalize (H5 n) as G1. intro.
apply (IHltlp1 (Str_nth_tl n tr_ie1) (Str_nth_tl n tr_ie2)); eauto.
apply tl_eq_n. trivial.
apply LTLmorF. elim H1. intros n [G1 G2].
exists n. split.
apply (IHltlp2 (Str_nth_tl n tr_ie1) (Str_nth_tl n tr_ie2)); eauto.
apply tl_eq_n. trivial.
apply LTLmorA. eauto.
apply LTLmorB. eauto.
apply LTLmand. eauto. eauto.
apply LTLmap. subst.
assert (mproj X m1 (hd tr_ie1) = mproj Y m2 (hd tr_ie2)).
inversion Heq. eauto.
replace (mproj Y m2 (hd tr_ie2)). trivial.
Qed.

Lemma equiv_tl : \forall (X Y : Type) (m1 : Mach X) (m2 : Mach Y),
isTraceEq m1 m2 →
(\forall ltl, LTL m1 ltlp → LTL m2 ltlp).
Proof.
intros X Y m1 m2 Heq ltlp. Hm1.
unfold LTL. intros a2 tr_ie2 Ha2 Htr2.
unfold isTraceEq in Heq.
generalize Heq. intros [Heq1 Heq2].
inversion Htr2.
assert (isMtraceF m1 m2 tr_ie2) as Hkk.
unfold isMtraceF. split; eauto. replace (hd tr_ie2). trivial.
generalize (Heq2 tr_ie2 Hkk) as Htr2. intro.
assert (\exists tr_ie1, isMtraceF m1 tr_ie1 (hd tr_ie1)
∧ EqSt (tr_strip m1 tr_ie1) (tr_strip m2 tr_ie2)
∧ (mi X m1) (hd tr_ie1) = true) as Hm.
unfold is_state_trace in Htr2. elim Htr2.
intros tt [Gtt1 Gtt2].
exists tt. split. unfold isMtraceF. inversion Gtt1. split; eauto.
split; eauto. inversion Gtt1. trivial.
elim Hm. intros tr_ie1 [Htr1m [Heqst Hm1]].
unfold LTL in Hm1.
inversion Htr1m as [G1 G2].
assert (isMtraceF m1 tr_ie1) as GG1.
split; eauto.

generalize (hd_tr_in_I X m1 tr_ie1 GG1) as Ghd. intro.
assert (isMtraceF m1 tr_ie1 (hd tr_ie1)) as G4.
 unfold isMtraceF. split; eauto.
 generalize (Hm1 (hd tr_ie1) tr_ie1 Ghd G4) as G3. intros.
apply equiv_tl_tr; eauto.
Qed.

Definition sameIP (X Y : Type) (m1 : Mach X) (m2 : Mach Y) : Prop :=
(\forall s,
(\exists a1, (mi X m1) a1 = true ∧ (mproj X m1) a1 = s)
→
(\exists a2, (mi Y m2) a2 = true ∧ (mproj Y m2) a2 = s)).

Definition sameSP (X Y : Type) (m1 : Mach X) (m2 : Mach Y) : Prop :=
(\forall s,
(\exists a1, (ms X m1) a1 = true ∧ (mproj X m1) a1 = s)
→
(\exists a2, (ms Y m2) a2 = true ∧ (mproj Y m2) a2 = s)).

Definition sameRP (X Y : Type) (m1 : Mach X) (m2 : Mach Y) : Prop :=
(\forall s s',
(\exists a1, \exists a1', (mr X m1) (a1,a1') = true ∧
(\text{mproj}\ X\ m1\ a1 = s \land (\text{mproj}\ X\ m1\ a1' = s'))

(\exists\ a2, \exists\ a2'.\ (\text{mr}\ Y\ m2\ (a2,a2') = \text{true} \land 
(\text{mproj}\ Y\ m2\ a2 = s \land (\text{mproj}\ Y\ m2\ a2' = s'))).

\textbf{Definition} MisDet (m:\text{Mach}\ (\text{state}^*\ \text{pstate}))\ (\text{preds}:\text{Dpreds}) : \text{Prop} := \\
\forall\ a\ a'.
(mr\ (\text{state}^*\ \text{pstate})\ m\ (a,a') = \text{true} \rightarrow 
\text{match}\ (a,a')\ \text{with} \\
| ((s,p),(s',p')) \Rightarrow 
\text{RisDet}\ s\ s'\ p\ p'\ (dp\ \text{preds})\ \text{end}.

\textbf{Definition} isMtoMD (m1:\text{Mach}\ \text{state})\ (m2:\text{Mach}\ (\text{state}^*\ \text{pstate}))\ (\text{preds}:\text{Dpreds}) : \text{Prop} := \\
\text{sameSP}\ \text{state}\ (\text{state}^*\ \text{pstate})\ m1\ m2 \land 
\text{sameRP}\ \text{state}\ (\text{state}^*\ \text{pstate})\ m1\ m2 \land 
\text{sameIP}\ \text{state}\ (\text{state}^*\ \text{pstate})\ m1\ m2 \land 
\text{MisDet}\ m2\ \text{preds}.

\textbf{Theorem} mach_equiv : \forall\ (m1:\text{Mach}\ \text{state})\ (m2:\text{Mach}\ (\text{state}^*\ \text{pstate}))\ \text{preds}, 
\text{isMtoMD}\ m1\ m2\ \text{preds} \rightarrow 
\text{isTraceEq}\ m1\ m2.
Admitted.

\textbf{Lemma} tr_eq_sym : \forall\ (X\ Y:\text{Type})\ (m1:\text{Mach}\ X)\ (m2:\text{Mach}\ Y), 
\text{isTraceEq}\ m1\ m2 \rightarrow \text{isTraceEq}\ m2\ m1.
\text{Proof}.
unfold\ \text{isTraceEq}.
intros\ X\ Y\ m1\ m2\ [G1\ G2].\ \text{split};\ \text{intros}.
apply\ G2.\ \text{eauto}.
apply\ G1.\ \text{eauto}.
Qed.

Set \text{Printing All}.

\textbf{Theorem} four_two : \forall\ \text{ctlp}\ \text{ltlp}, 
\forall\ (m1:\text{Mach}\ \text{state})\ (m2:\text{Mach}\ (\text{state}^*\ \text{pstate}))\ \text{preds}, 
\text{isMtoMD}\ m1\ m2\ \text{preds} \rightarrow 
\text{Matcher}\ \text{ctlp}\ \text{ltlp} \rightarrow 
\text{CTL}\ m2\ \text{ctlp} \rightarrow 
\text{LTL}\ m1\ \text{ltlp}.
\text{Proof}.
intros\ \text{ctlp}\ \text{ltlp}\ m1\ m2\ \text{preds}\ \text{Hfun}.
generalize\ (\text{mach_equiv}\ m1\ m2\ \text{preds}\ \text{Hfun})\ \text{as}\ G1.\ \text{intro}.
intros\ Hmat\ Hc.
apply\ (\text{equiv_ltl}\ (\text{state}^*\ \text{pstate})\ \text{state}\ m2\ m1).
apply\ tr_eq_sym;\ \text{eauto}.
apply\ (\text{m_ltl_imp_actl}\ (\text{state}^*\ \text{pstate})\ m2\ \text{ltlp}\ \text{ctlp});\ \text{eauto}.
Qed.