Technical Report

UCAM-CL-TR-788
ISSN 1476-2986

Number 788

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Branching-time reasoning for programs (extended version)

Byron Cook, Eric Koskinen, Moshe Vardi

July 2011

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

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

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

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

ISSN 1476-2986

Branching-time reasoning for programs

(Extended Version)

Byron Cook[1] **Eric Koskinen**[2] **Moshe Vardi**[3]

July 2011

Abstract

We describe a reduction from temporal property verification to a program analysis problem. Our reduction is 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, tree counterexamples for branching-time properties, abstraction refinement, etc.). Our reduction is state-based in nature but also forms the basis of an efficient algorithm for verifying trace-based properties, when combined with an iterative symbolic determinization technique, due to Cook and Koskinen [15].

In this extended version of [17], we formalize our encoding as a guarded transition system $\mathcal{G}$ parameterized by a finite set of ranking functions and the temporal logic property. We establish soundness between a safety property of $\mathcal{G}$ and the validity of a branching-time temporal logic property $\forall$CTL. $\forall$CTL is a sufficient logic for proving properties written in the trace-based Linear Temporal Logic via the iterative algorithm [15].

Finally, using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

[1]Microsoft Research and Queen Mary University of London
[2]University of Cambridge
[3]Rice University

Chapter 1

Introduction

In this report we elaborate on our previous results [17]. We describe a novel method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of procedures and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known interprocedural safety analysis tools (*e.g.* [2, 5, 8, 25, 34]) together with techniques for discovering termination arguments (*e.g.* [3, 6, 18]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to methods drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique yields speeds ups by multiple orders of magnitude for ∀CTL. Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL). Existential reasoning would also be possible, but care is required to ensure that the underlying program analysis tools appropriately use universal abstractions ("may" transitions) as well as existential abstractions ("must" transitions). Finally, our method works best when properties do not involve complex nesting of temporal operators. In order to better support these more complex properties our implementation would need to mix the construction of the program analysis problem with the analysis itself in the spirit of Impact [29], as invariants proved during a lazy unrolling could be used to prune away much of the work. As presented here, our approach instead creates a single encoding up front before performing program analysis.

1.1 Related work

There is a relationship between temporal logic verification and the problem of finding winning strategies in finite-state games or game-like structures such as alternating automata [4, 26, 36]. The technique presented in this paper can be viewed as a generalization of prior work to games over infinite state spaces.

Other previous tools and techniques are known for proving temporal properties of finite-state systems (*e.g.* [7, 11, 26]) or classes of infinite-state systems with specific structure (*e.g.* pushdown systems [38, 39] or parameterized systems [20]). Our proposal works for arbitrary transition systems, including programs.

A previous tool proves only trace-based (*i.e.* linear-time) properties of programs [14] using an adaptation of the traditional automata-theoretic approach [37]. By contrast, the reduction to program analysis given here promotes a state-based (*e.g.* branching-time) approach. Trace-based properties can be proved with our tool using a recently described iterative symbolic determinization technique [15]. In most cases our new approach is faster for LTL verification than [14] by several orders of magnitude.

When applying traditional bottom-up based methods for state-based logics (*e.g.* [12, 19, 21]) to infinite-state transition systems, one important challenge is to track reachability when considering relevant subformulae from the property. In contrast to the standard method of directly tracking the valuations of subformulae in the property with additional variables, we instead use procedure calls to encode the checking of subformulae as a program analysis problem. 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 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, φ where $R, s \vDash \varphi$ 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 $p \wedge q$ holds, where p, q are atomic propositions. Our encoding explores the cases $s_0 \vDash p \wedge q$, $s_0 \vDash p$, $s_0 \vDash q$, but not the cases $s_1 \vDash p \wedge q$, $s_2 \vDash p \wedge q$, $s_1 \vDash p$, $s_1 \vDash q$, $s_2 \vDash p$, $s_2 \vDash q$. A bottom-up algorithm will explore these superfluous cases.

Chaki *et al.* [9] attempt to address the same problem of subformulae and reachability for infinite-state transition systems by first computing a finite abstraction of the system *a priori* that is never refined again. Standard finite-state 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 [35] take a similar tack.

The tool YASM [24] 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 AFAGp). With our encoding the underlying tools are performing abstraction-refinement over tree counterexamples. Moreover, YASM is primarily designed to work for unnested existential properties [23] (*e.g.* EFp or EGp), whereas our focus is on precise support for arbitrary (possibly nested) universal properties.

Our encoding shares some similarities with the finite-state model checking procedure CEX from Figure 6 in Clarke *et al.* [13]. 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 *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.

1.2 Preliminaries

We begin with some definitions and terminology. Our results apply to state transition systems. For convenience however, we will begin by defining an imperative `while` language (used in all of our examples) and give its operation semantics in the form of a transition system. Our results can then be directly applied.

Programs. We assume that programs are written (or can be compiled to) the standard simple programming language (SPL [28]) given as follows:

C	::=	C ; C	Sequential composition
	|	$C + C$	Nondeterministic choice
	|	C^*	Looping
	|	`assume`(b)	Assume
	|	c	Basic command
	|	`skip`	Skip

SPL is parameterized by the set of basic commands c. 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 report, however, we will assume we are working with states that map (stack) variables $Vars$ to integers (and expressions from the domain of linear arithmetic). Hence the following definitions:

$$
\begin{array}{lcl}
c & ::= & x \texttt{ := } e \\
e & ::= & z \in \mathbf{Z} \mid v \in Vars \mid e + e \mid e - e \mid e \times e \mid e \div e \mid *_e \\
b & ::= & \mathsf{true} \mid \neg b \mid *_b \mid b \texttt{ \&\& } b \mid e \texttt{ == } e \mid e > e \mid e \geq e
\end{array}
$$

$$\frac{}{(\texttt{skip}\ ;\ C), s \longrightarrow C, s}\ Skip \qquad \frac{C_1, s \longrightarrow C_1', s'}{(C_1\ ;\ C_2), s \longrightarrow (C_1'\ ;\ C_2), s'}\ Seq$$

$$\frac{}{(C_1 + C_2), s \longrightarrow C_1, s}\ Nd1 \qquad \frac{}{(C_1 + C_2), s \longrightarrow C_2, s}\ Nd1$$

$$\frac{}{C^*, s \longrightarrow (\texttt{skip} + (C\ ;\ C^*)), s}\ Loop \qquad \frac{[\![b]\!]}{\texttt{assume}(b), s \longrightarrow \texttt{skip}, s}\ Asm$$

$$\frac{s' \in [\![c]\!]s}{c, s \longrightarrow \texttt{skip}, s'}\ Cmd$$

Figure 1.1: Operational semantics of SPL

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:

$$\begin{aligned} \texttt{while}\ (b)\ \{\ C\ \} &\equiv (\texttt{assume}(b)\ ;\ C)^*\ ;\ \texttt{assume}(\neg b) \\ \texttt{if}\ (b)\ \{\ C_1\ \}\ \texttt{else}\ \{C_2\ \} &\equiv (\texttt{assume}(b)\ ;\ C_1) + (\texttt{assume}(\neg b)\ ;\ C_2) \\ \texttt{if}\ (b)\ \{\ C_1\ \} &\equiv (\texttt{assume}(b)\ ;\ C_1) + (\texttt{assume}(\neg b)\ ;\ \texttt{skip}) \end{aligned}$$

Standard expressions (e.g. $\mathsf{false}, <, \leq, \neq, ||$) can also be derived. The above language is sufficient to model many C, C++, and Java programs (after some compilation and transformation).

Operational semantics. The small-step operational semantics for SPL are given in Figure 1.1. Configurations consist of the program text C and the current state s from the set of states S_C and the initial states $I_C \subseteq S_C$. We will assume the state s is a mapping from (typed) variables $Vars$ to values. We assume that state equality is decidable:

Axiom 1.1 (Distinguishability)**.** *For all $s, s' \in S$ either $s = s'$ or $s \neq s'$, and this can be determined in finite time.*

Program counter. A program text C is finite. We assume a special variable denoted pc which maps each subcommand of C to a unique element in a finite domain $\mathcal{L}$. The operational semantics in Figure 1.1 can be easily modified to explicitly show that the value of pc is updated accordingly in each step.

Definition 1.1 (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 $I \subseteq S$.*

The transition system for a program text C and set of variables $Vars$, $M_C = (S_C, R_C, I_C)$ (usually written simply M) requires that we define the transition relation:

$$R_C \equiv \{(s,t) \mid s \in S \wedge \exists C' \in \mathsf{sub}(C).\ \exists C''.\ C', s \longrightarrow C'', t\}$$

where $\mathsf{sub}(C)$ is defined inductively on C in the natural way.

Executions (Streams). For convenience, we do not allow finite traces—the transition relation must be such that every state s has at least one successor state:

Axiom 1.2. $\forall s.s \in S \Rightarrow \exists t.\ (s,t) \in R.$

The above axiom is without a loss of generality, as final states in systems with finite traces can be encoded as states that loop back to themselves in the transition relation. Note that we assume S does not include unreachable states, so we do not need to define a successor for them to ensure Axiom 1.2 holds.
We define a *trace* to be a stream (an infinite sequence) of states:

Definition 1.2 (Trace)**.** *For a transition relation R, we say that $\pi : \mathsf{Stream}\ S$ is a* trace *provided that $\mathsf{isTrace}\ \pi$ holds, defined coinductively as follows:*

$$\frac{\mathsf{isTrace}\ \pi' \quad (s, \mathsf{hd}\ \pi') \in R}{\mathsf{isTrace}\ s :: \pi'}$$

where $::$ and hd are the standard stream operations. In the remainder of this report, when we use the notation π, we assume that $\mathsf{isTrace}\ \pi$ holds. We use the notation π^i to mean the ith tail of π, and the notation π_0 to mean the head of π. Note that tail binds tighter than head, i.e. $\pi_0^i = (\pi^i)_0$. With coinductive reasoning we can show that there exists an execution from every state.

Lemma 1.3 (Trace existence)**.** *For all $M = (S, R, I)$,* $\forall s.\ s \in S \Rightarrow \exists \pi.\ \mathsf{isTrace}\ \pi.$

Proof. First construct a cofixedpoint, using Axiom 1.2 to witness a next state, denoted $(\texttt{next}\ s)$[1], to obtain an execution π as follows:

$$\texttt{traceFrom}\ s \ \equiv\ s :: (\texttt{traceFrom}\ (\texttt{next}\ s))$$

Then, via coinduction, unfold π and show that $\mathsf{isTrace}\ \pi$ holds. □

Definition 1.3 (Execution)**.** *For all $M = (S, R, I)$, π is an* execution *of M, if $\pi_0 \in I$.*

Definition 1.4 (Executions)**.** *For all $M = (S, R, I)$,* $\Pi_M \ \equiv\ \{\pi \mid \pi\ \mathit{is\ an\ execution\ of}\ M\}$.

Ranking functions. For a state space S, a ranking function f is a total map from S to a well ordered set with ordering relation $\prec$. A relation $R \subseteq S \times S$ is *well-founded* if and only if there exists a ranking function f such that $\forall (s, s') \in R.\ f(s') \prec f(s)$. We denote a finite set of ranking functions (or *measures*) as $\mathcal{M}$. Note that the existence of a finite set of ranking functions for a relation R is equivalent to containment of R within a finite union of well-founded relations [32]. In other words, a set of ranking functions $\{f_1, ..., f_n\}$ can denote the disjunctively well-founded relation $\{(s, s') \mid f_1(s') \prec f_1(s) \vee ... \vee f_n(s') \prec f_n(s)\}$.

Theorem 1.4. *For all R, I, if R is well-founded, then from every state $s \in I$ there are no infinite traces in R. Proof. Trivial.* □

[1] To construct this in Coq, we used the sigma operator `proj1_sig`, which is an existential operator that yields a whitness.

$$\frac{\alpha(s)}{R, s \vDash \alpha} \qquad \frac{R, s \vDash \varphi_1 \quad R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \wedge \varphi_2} \qquad \frac{R, s \vDash \varphi_1 \ \vee \ R, s \vDash \varphi_2}{R, s \vDash \varphi_1 \vee \varphi_2}$$

$$\frac{\forall(s_0, s_1, ...).\ s_0 = s \Rightarrow \exists i \geq 0.\ R, s_i \vDash \varphi}{R, s \vDash \mathsf{AF}\varphi}$$

$$\frac{\begin{array}{c}\forall(s_0, s_1, ...).\ s = s_0 \Rightarrow (\forall i \geq 0.\ R, s_i \vDash \varphi_1) \vee \\ (\exists j \geq 0.\ R, s_j \vDash \varphi_2 \wedge \forall i \in [0, j).\ R, s_i \vDash \varphi_1)\end{array}}{R, s \vDash \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]}$$

Figure 1.2: Semantics of ∀CTL: $\vDash$

1.3 Temporal logic

We are concerned with verifying temporal properties that may be written either as trace-based properties in LTL or as state-based properties in the existential-free fragment of computation tree logic (∀CTL). The encoding we describe in this paper is state-based in nature and, as such, is readily suitable to ∀CTL properties. To prove LTL properties we combine a recently described iterative symbolic determinization technique [15] with the ∀CTL proving technique described here.

The syntax of a ∀CTL formula is

$$\varphi ::= \alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi]$$

The standard semantics of ∀CTL are given in Fig. 1.2. α is an atomic proposition. ∀CTL's temporal operators are state-based in structure. The operator $\mathsf{AF}\varphi$ specifies that, across all computation sequences from the current state, a state in which φ holds must be reached. The $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ operator specifies that φ_1 holds in every state where φ_2 does not yet hold. Notice that the operator $\mathsf{AG}\varphi$, which specifies that φ globally holds in all reachable future states, can be derived: $\mathsf{A}[\varphi \ \mathsf{W} \ \mathsf{false}]$. We include the AG operator in Chapter 2 for illustrative purposes, but omit this redundancy in Chapter 3.

We use AF and AW as our base operators (as opposed to the more standard U and R), as each corresponds to a distinct form of proof: AF to termination and AW to safety. We omit the next state operator AX. Formulae with U and R can be expressed in ∀CTL. We assume that formulae are written in negation normal form, in which negation only occurs next to atomic propositions (we also assume that the domain of atomic propositions is closed under negation). A formula that is not in negation normal form can be easily normalized.

Definition 1.5 (∀CTL-lifting)**.** *For all $M = (S, R, I)$ and ∀CTL property φ,*

$$M \vDash \varphi \ \equiv \ \forall s \in I.\ R, s \vDash \varphi.$$

Subformulae. We will need to enumerate subformulae, taking care to uniquely identify each one. To this end, our definition of subformulae mantains a context path:

$$\kappa \ \equiv \ \bot \mid \mathsf{L} \ \kappa \mid \mathsf{R} \ \kappa$$

which indicates the path from the root $\bot$ (the outermost property φ), to the particular subproperty φ of interest, at each step taking either the left or right subformula ($\mathsf{L}\,\kappa$ or $\mathsf{R}\,\kappa$). Consequently, the set of subformulae is a set of (φ, κ) pairs, and begins with the root context element $\mathsf{sub}(\varphi, \bot)$:

Definition 1.6 (Subformulae)**.** *For an $\forall CTL$ property φ,*

$$\begin{aligned}
\mathit{sub}(\alpha, \kappa) &\equiv \{(\alpha, \kappa)\} \\
\mathit{sub}(\varphi \vee \varphi', \kappa) &\equiv \{(\varphi \vee \varphi', \kappa)\} \cup \mathit{sub}(\varphi, \mathsf{L}\,\kappa) \cup \mathit{sub}(\varphi', \mathsf{R}\,\kappa) \\
\mathit{sub}(\varphi \wedge \varphi', \kappa) &\equiv \{(\varphi \wedge \varphi', \kappa)\} \cup \mathit{sub}(\varphi, \mathsf{L}\,\kappa) \cup \mathit{sub}(\varphi', \mathsf{R}\,\kappa) \\
\mathit{sub}(\mathit{AF}\varphi, \kappa) &\equiv \{(\mathit{AF}\varphi, \kappa)\} \cup \mathit{sub}(\varphi, \mathsf{L}\,\kappa) \\
\mathit{sub}(\mathit{A}[\varphi\, \mathit{W} \varphi'], \kappa) &\equiv \{(\mathit{A}[\varphi\, \mathit{W} \varphi'], \kappa)\} \cup \mathit{sub}(\varphi, \mathsf{L}\,\kappa) \cup \mathit{sub}(\varphi', \mathsf{R}\,\kappa)
\end{aligned}$$

Definition 1.7 (Immediate subformulae)**.** *For an $\forall CTL$ property φ,*

$$\begin{aligned}
\mathit{isub}(\alpha) &\equiv \varnothing \\
\mathit{isub}(\varphi \vee \varphi') &\equiv \{\varphi, \varphi'\} \\
\mathit{isub}(\varphi \wedge \varphi') &\equiv \{\varphi, \varphi'\} \\
\mathit{isub}(\mathit{AF}\varphi) &\equiv \{\varphi\} \\
\mathit{isub}(\mathit{A}[\varphi\, \mathit{W} \varphi']) &\equiv \{\varphi, \varphi'\}
\end{aligned}$$

Chapter 2

From temporal logic to program analysis

In this section we introduce a reduction which, when given a transition system M and a temporal logic property φ, generates a program that encodes the search for the proof that φ holds of M. Existing program analysis tools can then be used to reason about the validity of the property. The encoding is state-based in nature but can be used in combination with an iterative symbolic determinization procedure, due to Cook and Koskinen [15], to obtain an efficient algorithm for verifying trace-based properties.

Example 2.1 (Acquire/release)**.** *We will use this example (written in the* `while` *language from Section 1.2) throughout this chapter. It is a typical lock acquire/release-style program, where we are interested in proving the* $\forall$*CTL property* $\varphi = \mathsf{AG}[(x = 1) \Rightarrow \mathsf{AF}(x = 0)]$*. Assume that initially* $x = 0$*.*

```
1  while(*) {
2     x := 1;
3     n := *;
4     while(n>0) {
5        n := n - 1;
6     }
7     x := 0;
8  }
9  while(true) { skip }
```

2.1 Encoding

Our encoding $\mathcal{E}$ is given in Fig. 2.1. When given a transition relation system $M = (S, R, I)$, a finite set of ranking functions $\mathcal{M}$, and $\forall$CTL property φ, $\mathcal{E}$ returns a set of procedures. There is a procedure denoted $\mathtt{enc}^{\kappa}_{\psi} : s \to \mathbf{B}$ for every (ψ, κ)-subformula of φ, including a root procedure denoted $\mathtt{enc}^{\perp}_{\varphi}$. For Example 2.1, $\mathcal{E}$ returns the following set of procedures:

$$\left\{ \mathtt{enc}^{\perp}_{\mathsf{AG}[(x\neq 1)\vee\mathsf{AF}(x=0)]},\ \mathtt{enc}^{\mathsf{L}\perp}_{[(x\neq 1)\vee\mathsf{AF}(x=0)]},\ \mathtt{enc}^{\mathsf{L}\mathsf{L}\perp}_{(x\neq 1)},\ \mathtt{enc}^{\mathsf{R}\mathsf{L}\perp}_{\mathsf{AF}(x=0)},\ \mathtt{enc}^{\mathsf{L}\mathsf{R}\mathsf{L}\perp}_{(x=0)} \right\}$$

Together, these procedures encode the search for the proof that φ holds of M.

Executions of the procedures explore the $S \times \mathsf{sub}(\varphi)$ state space in a depth-first manner, passing the current state $s \in S$ on the stack, starting with the root procedure $\mathtt{enc}^{\perp}_{\varphi}$ and an initial state $s_0 \in I$. At each successive procedure call $\mathtt{enc}^{\kappa}_{\psi}(s)$, the encoding is attempting to determine whether subformula ψ holds of a particular state s. Rather than explicitly tracking this information, however, $\mathtt{enc}^{\kappa}_{\psi}(s)$ returns false whenever ψ does not hold of s. Consequently if, at the root level, $\mathtt{enc}^{\perp}_{\varphi}$ can be proved to never return false, it must be the case that the overall property φ holds of the initial state s (we discuss the termination of $\mathcal{E}$ below). This is the intution behind the following main theorem:

Theorem 2.1 (Soundness)**.** *For a machine $M = (S, R, I)$ and $\forall$CTL property φ,*

$$\exists \textit{ finite } \mathcal{M}.\ \mathcal{E}(M, \mathcal{M}, \varphi) \textit{ cannot return } \mathsf{false} \quad \Rightarrow \quad M \models \varphi$$

Proof. See Chapter 3. □

where $\mathcal{M}$ is, as described earlier, a finite set of ranking functions. We abuse notation slightly here, using $\mathcal{E}(M, \mathcal{M}, \varphi)$ to mean $\forall s \in I.\ \mathtt{enc}^{\perp}_{\varphi}(s)$. We formally define "cannot return false" by giving $\mathcal{E}$ as a guarded transition system in Section 3.3, but informally it means there is no execution of $\mathcal{E}$ where false is returned.

When a program analysis is applied to the procedures generated by $\mathcal{E}$, it is implementing what is needed to prove branching-time behaviors of the original transition system (*e.g.* backtracking, eventuality checking, tree counterexamples, abstraction, abstraction-refinement, etc).

Notice that, in contrast to bottom-up techniques for proving state-based properties, this depth-first traversal boasts several improvements. First, only reachable states are considered. A safety analysis may over-approximate this set of states, but it will never need to find approximations for unreachable states. By contrast, bottom-up algorithms require that concrete unreachable states be considered. Second, as discussed in Section 1.1, only relevant state/subformula pairs are considered. Our technique is more amenable to infinite-state transition systems because often solutions can be found where bottom-up techniques would diverge.

2.2 Proof search

What remains is to understand how a given $\mathtt{enc}^{\kappa}_{\psi} \in \mathcal{E}(M, \mathcal{M}, \varphi)$ determines whether a subformula (ψ, κ) holds of a state s. By passing the state on the stack, we can consider multiple branching scenarios. When a particular ψ is a $\wedge$ or AW subformula, then $\mathtt{enc}^{\kappa}_{\psi}(s)$ ensures that all possibilities are considered by establishing feasible execution paths of $\mathtt{enc}^{\kappa}_{\psi}(s)$ to all of them. When a particular ψ is a $\vee$ or AF subformula, $\mathtt{enc}^{\kappa}_{\psi}(s)$ enables executions to consider all of the possible cases that might cause ψ to hold of s. If one is found, true is returned. Otherwise, false will be returned if none are found. This is the intuition behind the first invariant maintained by $\mathcal{E}$:

$$INV_1 : \forall s, \psi, \kappa.\ R, s \not\models \psi \text{ implies } \mathtt{enc}^{\kappa}_{\psi}(s) \text{ can return } \mathsf{false}$$

Consider the $\mathtt{enc}^{\kappa}_{\psi \vee \psi'}$ case from the definition of $\mathcal{E}$. Imagine that $\psi \equiv \mathsf{x} \neq 1$, and $\psi' \equiv \mathsf{AG}(\mathsf{x} = 0)$. In this case we want to know that one of the subformulae (*i.e.* $\mathsf{x} \neq 1$ or $\mathsf{AG}(\mathsf{x} = 0)$) holds. A procedure call $\mathtt{enc}^{\mathsf{L}\,\kappa}_{x \neq 1}(s)$ is made to explore whether $\mathsf{x} \neq 1$ as well as a

$\mathcal{E}(M, \mathcal{M}, \varphi) \equiv \mathcal{E}(M, \mathcal{M}, \varphi, \bot)$ where

$$
\begin{aligned}
\mathcal{E}(M, \mathcal{M}, \alpha, \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\alpha}\} \\
\mathcal{E}(M, \mathcal{M}, \psi \wedge \psi', \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\mathsf{A}\psi\wedge\psi'}\} \cup \mathcal{E}(M, \mathcal{M}, \psi, \mathsf{L}\ \kappa) \cup \mathcal{E}(M, \mathcal{M}, \psi', \mathsf{R}\ \kappa) \\
\mathcal{E}(M, \mathcal{M}, \psi \vee \psi', \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\psi\vee\psi'}\} \cup \mathcal{E}(M, \mathcal{M}, \psi, \mathsf{L}\ \kappa) \cup \mathcal{E}(M, \mathcal{M}, \psi', \mathsf{R}\ \kappa) \\
\mathcal{E}(M, \mathcal{M}, \mathsf{AF}\psi, \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\mathsf{AF}\psi}\} \cup \mathcal{E}(M, \mathcal{M}, \psi, \mathsf{L}\ \kappa) \\
\mathcal{E}(M, \mathcal{M}, \mathsf{AG}\psi, \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\mathsf{AG}\psi}\} \cup \mathcal{E}(M, \mathcal{M}, \psi, \mathsf{L}\ \kappa) \\
\mathcal{E}(M, \mathcal{M}, \mathsf{A}[\psi \mathsf{W} \psi'], \kappa) &\equiv \{\mathtt{enc}^{\kappa}_{\mathsf{A}[\psi\mathsf{W}\psi']}\} \cup \mathcal{E}(M, \mathcal{M}, \psi, \mathsf{L}\ \kappa) \cup \mathcal{E}(M, \mathcal{M}, \psi', \mathsf{R}\ \kappa)
\end{aligned}
$$

where $M = (S, R, I)$, and $\forall \psi, \kappa.\ \mathtt{enc}^{\kappa}_{\psi}$ is defined as follows:

```
bool enc^κ_α(state s) { return α(s); }

bool enc^κ_{ψ∧ψ'}(state s) {
  if (*) return enc^{L κ}_ψ(s);
  else return enc^{R κ}_{ψ'}(s);
}

bool enc^κ_{ψ∨ψ'}(state s) {
  if (enc^{L κ}_ψ(s)) return true;
  else return enc^{R κ}_{ψ'}(s);
}

bool enc^κ_{A[ψWψ']}(state s) {
  while (true) {
    if (*) return true;
    if (¬ enc^{L κ}_ψ(s)) return enc^{R κ}_{ψ'}(s);
    s := choose({s' | R(s,s')});
  }
}
```

```
bool enc^κ_{AFψ}(state s) {
  bool dup = false; state 's ;
  while (true) {
    if (*) return true;
    if (enc^{L κ}_ψ(s)) return true;
    if (dup && ¬(∃f ∈ M. f(s) ≺ f('s)))
        return false;
    if (¬ dup && *)
       { dup := true; 's := s; }
    s := choose({s' | R(s,s')});
  }
}
// Included for illustrative purposes.
// AGφ can be treated as A[φW false].
bool enc^κ_{AGψ}(state s) {
  while (true) {
    if (*) return true;
    if (¬ enc^{L κ}_ψ(s)) return false;
    s := choose({s' | R(s,s')});
  }
}
```

Figure 2.1: The encoding $\mathcal{E}$ is a function which takes a machine $M = (S, R, I)$, a finite set of ranking functions $\mathcal{M}$, and an $\forall$CTL property φ, and returns a set of procedures. If a sufficient $\mathcal{M}$ is found such that `assert(enc`$^{\bot}_{\varphi}$`(s))` can be proved safe for all $s \in I$, then φ holds of M (*i.e.* $M \vDash \varphi$). `choose()` nondeterministically selects an element from the set given by its argument. $* \equiv$ `choose({true,false})`

separate procedure call $\mathtt{enc}^{\mathsf{R}\,\kappa}_{\mathsf{AG}(\mathsf{x}=0)}(s)$ with the same current state s to explore $\mathsf{AG}(\mathsf{x}=0)$. 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 abide INV_1), then there will be an execution in which $\mathtt{enc}^{\kappa}_{\psi \vee \psi'}(s)$ can return false (also abiding INV_1). A standard program analysis tool (*e.g.* SLAM [2] or BLAST [25]) will find this case. By maintaining this invariant in each procedure, a proof that the outermost procedure $\mathtt{enc}^{\perp}_{\varphi}$ cannot return false implies that the property φ holds of the machine M.

Because we want to consider every state that is reachable from a finite prefix of an infinite path, it must be possible for the procedure calls to return from every state. If it were possible for the checking of a subformula like $\mathsf{AG}(\mathsf{x}=0)$ to diverge (thus never returning false) then the above code fragment would never return false, and thus the top-level procedure $\mathtt{enc}^{\perp}_{\varphi}$ would never return false. To this end, $\mathcal{E}$ maintains a second invariant:

$$INV_2 : \forall s, \psi, \kappa.\ \mathtt{enc}^{\kappa}_{\psi}(s) \text{ can return } \mathsf{true}$$

It is this requirement that necessitates the additional nondeterministic "`if (*) return true`" commands found within the loops in $\mathtt{enc}^{\kappa}_{\mathsf{A}[\psi \mathsf{W} \psi']}$, $\mathtt{enc}^{\kappa}_{\mathsf{AF}\psi}$, and $\mathtt{enc}^{\kappa}_{\mathsf{AG}\psi}$. One can think of "`if (*) return true`" as a form of backtracking. In our encoding, a nondeterministic return of true is not declaring that the property holds (we must *always* return true to do that). Instead, a nondeterministic return of true in the encoding means that a program analysis can freely backtrack and switch to other possible scenarios during its search for a proof.

In the AF case, 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 this particular subset of the transitive closure of the transition system (we will precisely say which subset in Chapter 3)[1].

When M is a program. In practice, if the input transition system is implemented as a program, then we can perform a number of additional static optimizations from abstract interpretation that facilitates the application of current program analysis tools. These optimizations are implemented in PEVAL : $(s \to \mathbf{B}) \to (s \to \mathbf{B})$. We describe some of these optimizations below and the remainder can be found in Section 4.1.

First, consider the naïve implementation of AG given in Fig. 2.1 which, in essence, is interpreting the cross product of R together with the following program:

```
while true do
   if (*) return true;
   if (¬ enc^κ_ψ(s)) return false;
done
```

[1]This is an adaptation of a known technique [18]. 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.

Since we are considering programs as our input systems, we can build an encoding where the following fragment is instrumented in each line of a procedure based on the original input program:

```
if (¬ enc^κ_ψ(s)) return false;
if (*) return true;
```

Second, because the program state is passed on the stack, a procedure call $\texttt{enc}^{\kappa}_{\psi}$ for a subformula ψ 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. Some additional details about PEval are discussed in Section 4.1.

```
void main {
   bool x; nat n;
   { x := 0; n := *; }
   assert( enc^⊥_{AG((x≠1)∨AF(x=0))}_0(x,n) );
}

bool enc^⊥_{AG((x≠1)∨AF(x=0))}_0(bool x, nat n) {
  while(*) {
     x := 1;
     if (¬enc^{L⊥}_{(x≠1)∨AF(x=0)}_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^{L⊥}_{(x≠1)∨AF(x=0)}_3(bool x, nat n) {
  if (x ≠ 1) return true;
  return enc^{RL⊥}_{AF(x=0)}_3(x,n);
}
```

```
bool enc^{RL⊥}_{AF(x=0)}_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 2.2: The procedures given by $\mathcal{E}(M, \mathcal{M}, \varphi)$ where $\varphi = \mathsf{AG}[(x = 1) \Rightarrow \mathsf{AF}(x = 0)]$ and M is the program from Example 2.1. PEval has been applied to these procedures, and irrelevent procedures have been omitted.

Example. We now return to Example 2.1. The output of $\mathcal{E}(M, \mathcal{M}, \varphi)$, after performing several optimizations (discussed above and in Section 4.1) is given in Figure 2.2. 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 $\mathbf{enc}^{R\,L\,\bot}_{AF(x=0)}\_3$ *where* 3 *indicates that execution will begin where* $pc = 3$*). For every $(\psi, \kappa) \in sub(\varphi)$ and pc valuation, there is a corresponding method* $\mathbf{enc}^{\kappa}_{\psi}\_pc$*. 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* $\mathbf{enc}^{\bot}_{AG((x\neq 1)\vee AF(x=0))}\_0$ *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 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 $\mathbf{enc}^{\bot}_{AG((x\neq 1)\vee AF(x=0))}$ *corresponds to the property* $AG((x \neq 1) \vee AF(x = 0))$ *and returns* false *if there is a reachable state where* $((x \neq 1) \vee AF(x = 0))$ *does not hold. It accomplishes this by calling* $\mathbf{enc}^{L\,\bot}_{((x\neq 1)\vee AF(x=0))}$ *on each line and passing the current state.*

If $((x \neq 1) \vee AF(x = 0))$ *does not hold from the current state, then there will be a way for* $\mathbf{enc}^{L\,\kappa}_{((x\neq 1)\vee AF(x=0))}$ *to return* false*, in which case* $\mathbf{enc}^{\bot}_{AG((x\neq 1)\vee AF(x=0))}$ *immediately returns* false *(leading to an assertion failure in* `main`*). The procedures for disjunction (*$\mathbf{enc}^{L\,\bot}_{((x\neq 1)\vee AF(x=0))}$*) and atomic propositions (*$\mathbf{enc}^{L\,L\,\bot}_{x\neq 1}$ *and* $\mathbf{enc}^{L\,R\,L\,\bot}_{x=0}$*) are straight-forward following Fig. 2.1, and also maintain INV_1. We have inlined atomic propositions.*

The procedure $\mathbf{enc}^{R\,L\,\bot}_{AF(x=0)}$ *is, in some sense, the complement of AG. It is designed to return* true *whenever there is a path to a state where $x = 0$ holds, and will return* 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* $\mathbf{enc}^{L\,R\,L\,\bot}_{x=0}$ *(which as been inlined) returns true, and returning* 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 2.2, as we have not found a sufficient finite set of ranking functions $\mathcal{M}$. We now describe a method for discovering such an $\mathcal{M}$.

2.3 Looking for $\mathcal{M}$

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 $\mathtt{enc}^{\bot}_{\varphi}(s)$ does not return false. Our top-level algorithm adapts a known method [18] in order to iteratively find a sufficient $\mathcal{M}$:

Algorithm 2.2 (Main algorithm).

```
let prove(M, φ) =
    M = ∅;
    (enc^⊥_φ,...) = map PEVAL E(M, M, φ);
    while (∃s ∈ I. enc^⊥_φ(s) can return false) do
        let χ be a counterexample in
        if ∃ lasso path fragment χ' from χ then
            if ∃ witness f showing χ' w.f. then
                M := M ∪ {f};
                (enc^⊥_φ,...) := map PEVAL E(M, M, φ);
            else return χ
        else return χ
    done
    return Success
```

This algorithm begins with the empty set for $\mathcal{M}$, and constructs the set of procedures, partially evaluating them with PEval (see Section 4.1). Then, in our implementation, new ranking functions are automatically synthesized by examining counterexamples. A counterexample in ∀CTL is tree-like as follows:

$$\begin{aligned}\chi \;::=\;& \mathsf{CEX}^{\kappa}_{\alpha} \text{ of } s \mid \mathsf{CEX}^{\kappa}_{\wedge} \text{ of } \chi \mid \mathsf{CEX}^{\kappa}_{\vee} \text{ of } \chi \times \chi \\ \mid\;& \mathsf{CEX}^{\kappa}_{\mathsf{AG}} \text{ of } \pi \times \chi \mid \mathsf{CEX}^{\kappa}_{\mathsf{AF}} \text{ of } \pi \times \pi \times \chi \mid \mathsf{CEX}^{\kappa}_{\mathsf{W}} \text{ of } \pi \times \chi \times \chi\end{aligned}$$

where the parameter κ denotes the path through the formula, and π is a trace through the encoding $\mathcal{E}(M, \mathcal{M}, \varphi)$. 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 $\mathsf{CEX}^{\kappa}_{\alpha}$ is simply a state in which α 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 $\mathcal{E}(M, \mathcal{M}, \varphi)$, including points where dup is set to true. The commands from the input program can be used to populate an instance of χ.

When a counterexample is reported that contains an instance of $\mathsf{CEX}^{\kappa}_{\mathsf{AF}}$ (*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 $\mathcal{M}$. One source of incompleteness of our implementation comes from our reliance on

lassos: 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 [18], but in industrial examples these programs are rare.

Example. We return again to Example 2.1, and apply Algorithm 2.2. Initially we let $\mathcal{M} \equiv \varnothing$. *Running a refinement-based safety prover will yield a counterexample pertaining to line* `lab_5` *of* $\mathbf{enc}^{RL\,\bot}_{AF(x=0)}$, *where we denote a state as* $\begin{bmatrix} x \\ n \\ pc \end{bmatrix}$ *and we denote transition relations as* $\left[\!\!\left[\begin{smallmatrix} \text{'}x=x \\ \text{'}n=n \\ \text{'}pc=pc. \end{smallmatrix} \right]\!\!\right]$:

$$
\begin{array}{l}
(\mathsf{CEX}^{\kappa}_{\mathsf{AG}} \;\; \left(\begin{bmatrix} 0 \\ n \\ 1 \end{bmatrix} :: \begin{bmatrix} 1 \\ n \\ 2 \end{bmatrix} :: \begin{bmatrix} 1 \\ n \\ 3 \end{bmatrix} :: \begin{bmatrix} 1 \\ n \\ 4 \end{bmatrix} :: \begin{bmatrix} 1 \\ n \\ 5 \end{bmatrix}\right), \\
\quad (\mathsf{CEX}^{\kappa}_{\vee} \;\; (\mathsf{CEX}^{\kappa}_{\alpha} \begin{bmatrix} 1 \\ n \\ 5 \end{bmatrix}) \\
\qquad\qquad (\mathsf{CEX}^{\kappa}_{\mathsf{AF}} \;\; \begin{bmatrix} 1 \\ n \\ 5 \end{bmatrix}, \;\; \left[\!\!\left[\begin{smallmatrix} x_5=x \\ n_5=n+1 \\ pc_5=pc \end{smallmatrix} \right]\!\!\right], \;\; (\mathsf{CEX}^{\kappa}_{\alpha} \begin{bmatrix} 1 \\ n \\ 5 \end{bmatrix}))))
\end{array}
$$

This counterexample appears because we have not found a finite $\mathcal{M}$ *such that in* $\mathbf{enc}^{RL\,\bot}_{AF(x=0)}$ *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. *[18]), 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,* $\mathbf{enc}^{\bot}_{AG((x\neq 1)\vee AF(x=0))}$ always *returns* `true`, *and consequently, by Theorem 2.1,* φ *holds of the original program.*

Chapter 3

Correctness

In this section we will formalize our result, Theorem 2.1, which states that for a machine $M = (S, R, I)$ and $\forall$CTL property φ,

$$\exists \text{ finite } \mathcal{M}.\ \mathcal{E}(M, \mathcal{M}, \varphi) \text{ cannot return } \mathsf{false} \quad \Rightarrow \quad M \vDash \varphi$$

For convenience, we introduce an alternative relational formulation of $\forall$CTL, $\vdash$. This formulation more closely matches our definition of $\mathcal{E}$ in that it is given over sets of states, AW is defined in terms of reachability, and AF is defined in terms of well-foundedness. In effect the encoding $\mathcal{E}$ is characterizing these sets as a symbolic characteristic function (from states to $\{\mathsf{true}, \mathsf{false}\}$). Our proof starts by showing that $\vdash$ is equivalent to $\vDash$. We then formally define $\mathcal{E}$ as a function from a program and property to a guarded transition system $\mathcal{G}^{\mathcal{M}}$ parameterized by $\mathcal{M}$, for which a notion of "cannot return false" can be given. We show that for this guarded transition system, $\exists \mathcal{M}.\ \mathcal{G}^{\mathcal{M}}$ cannot return $\mathsf{false} \Rightarrow \langle R, I \rangle \vdash \varphi$, Finally, Theorem 2.1 directly follows, given the equivalence between $\vdash$ and $\vDash$.

3.1 Relational formulation of $\forall$CTL semantics

Our relational formulation of $\forall$CTL is displayed in Fig. 3.1. Unlike the standard formulation given in Figure 1.2, ours is more amenable to reasoning about infinite-state systems. In our formulation proof trees are based on *partitioning* the state space rather than *enumerating* the state space. We use the notation $\langle R, I \rangle \vdash \varphi$ to denote that a property φ is valid for a transition system $M = (S, R, I)$. This entailment relation is then defined inductively.

An atomic proposition α involves a simple check to see if I is contained within the set of states in which α holds. The conjunction rule requires that both φ_1 and φ_2 hold of all states in I and the disjunction rule partitions the states into two sets, one in which φ_1 holds and one in which φ_2 holds.

Frontiers. The property $\mathsf{AF}\varphi$ depends on the existence of a set of states which we will call a *frontier* $\mathcal{F}$. Intuitively, the frontier $\mathcal{F}$ of a set of initial states I, is a set of states through which every trace originating at a state in I must pass.

We use frontiers in our formulation of $\mathsf{AF}\varphi$ to characterize the places where φ holds, requiring that all paths from I eventually reach a frontier. The inductive relation $\mathsf{walk}_I^{\mathcal{F}}$,

$$\frac{I \subseteq \{s \mid \alpha(s)\}}{\langle R, I\rangle \vdash \alpha} \qquad \frac{\langle R, I\rangle \vdash \varphi_1 \quad \langle R, I\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \wedge \varphi_2}$$

$$\frac{\exists I_1, I_2.\ I = I_1 \cup I_2 \wedge \langle R, I_1\rangle \vdash \varphi_1 \wedge \langle R, I_2\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \varphi_1 \vee \varphi_2}$$

$$\frac{\exists \mathcal{F}. \mathsf{walk}_I^{\mathcal{F}} \text{ is w.f.} \ \wedge \langle R, \mathcal{F}\rangle \vdash \varphi}{\langle R, I\rangle \vdash \mathsf{AF}\varphi}$$

$$\frac{\exists \mathcal{F}. \forall (s, s') \in \mathsf{walk}_I^{\mathcal{F}}. \langle R, \{s\}\rangle \vdash \varphi_1 \wedge \langle R, \mathcal{F}\rangle \vdash \varphi_2}{\langle R, I\rangle \vdash \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]}$$

$$\frac{R(s, s') \wedge s \notin \mathcal{F} \wedge s \in I}{\mathsf{walk}_I^{\mathcal{F}}\ (s, s')}$$

$$\frac{R(s', s'') \wedge s' \notin \mathcal{F} \wedge \mathsf{walk}_I^{\mathcal{F}}\ (s, s')}{\mathsf{walk}_I^{\mathcal{F}}\ (s', s'')}$$

Figure 3.1: Relational formulation of ∀CTL: $\vdash$

given on the right in Fig. 3.1, is a subset of R and allows us to characterize the region that includes every possible transition along every trace from I up to, but not including, $\mathcal{F}$. When $\mathcal{F} = \varnothing$, $\mathsf{walk}_I^{\mathcal{F}}$ is equivalent to the portion of the transition relation accessible from I. In our characterization of AF we require that $\mathsf{walk}_I^{\mathcal{F}}$ be well-founded. In this way, 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 $\mathsf{walk}_I^{\mathcal{F}}$ rather than searching for per-trace ranking functions. The final rule in the left of Fig. 3.1 is for the AW operator, which also uses a frontier and the relation $\mathsf{walk}_I^{\mathcal{F}}$ representing the arcs along the way to the frontier $\mathcal{F}$. To prove $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$, all states along the path to the frontier must satisfy φ_1 and states at the frontier—*should one ever get there*—all must satisfy φ_2. Notice that no rule is needed for AG (used in the previous section) since $\mathsf{AG}\varphi = \mathsf{A}[\varphi\ \mathsf{W}\ \mathsf{false}]$.

3.2 Equivalence between $\vdash$ and $\vDash$

We now describe the equivalence result between our relational formulation of ∀CTL given in Figure 3.1 and the standard semantics of ∀CTL given in Figure 1.2.

3.2.1 The ⇒ direction

Lemma 3.1 (Avoiding $\mathcal{F}$)**.** *For all $R, I, \mathcal{F}, \pi \in \mathit{traces}(I, R)$,*

$$\forall n \geq 0.\ \pi_0^n \notin \mathcal{F} \quad \Rightarrow \quad \forall n \geq 0.\ \mathit{walk}_I^{\mathcal{F}}\ (\pi_0^n, \pi_0^{n+1})$$

Lemma 3.2 (Reach $\mathcal{F}$ choice)**.** *For all $R, I, \mathcal{F}, \pi \in \mathit{traces}(I, R)$,*

$$\{\forall n.\ \pi_0^n \notin \mathcal{F}\} + \{\exists n.\ \pi_0^n \in \mathcal{F} \wedge \forall m < n.\ \pi_0^m \notin \mathcal{F}\}$$

Proof. Coinduction and the axiom of choice. □

Theorem 3.3 (⇒ direction)**.** *For all φ, I, R,*

$$\langle R, I\rangle \vdash \varphi \quad \Rightarrow \quad M \vDash \varphi$$

Proof. By structural induction on φ. The $\alpha, \wedge, \vee$, cases are straight-forward. The AF case requires that Theorem 1.4 be applied to the relation $\mathsf{walk}_I^{\mathcal{F}}$, and then proceeds by contradiction. In the $\mathsf{A}[\varphi_1 \mathsf{W} \varphi_2]$ case, for every trace π, Lemma 3.2 says that there are two possibilities. The possibilities then align with those in the semantics of $\vDash$. □

3.2.2 The $\Leftarrow$ direction

We now show the opposite direction, which says that if a property holds in the standard semantics of ∀CTL, then it also holds in our relational semantics. We begin with some useful lemmas.

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 fixed point $\mathsf{lfp}(X \mapsto FO(X))$ exists.

Lemma 3.4 (Reachable set)**.** *For all $I, \mathcal{F}$, there exists a set S such that $s' \in S$ if and only if $\exists s \in I$ such that $(s, s') \in (\mathsf{walk}_I^{\mathcal{F}})^*$.*

Proof. Let $S \equiv \mathsf{lfp}(X \mapsto I \cup \{s' \mid s \in X \wedge (s, s') \in \mathsf{walk}_I^{\mathcal{F}}\})$. This fixedpoint exists because the expression is monotone. □

Lemma 3.5 (Traces escape walk)**.** *For all $\pi, I, \mathcal{F}, n$, $\quad \pi_0^n \in \mathcal{F} \ \Rightarrow\ (\pi_0^n, \pi_0^{n+1}) \notin \mathsf{walk}_I^{\mathcal{F}}$.*
Proof. By induction. □

Lemma 3.6. *For every $I, \mathcal{F}$ $\quad \mathsf{traces}(I, \mathsf{walk}_I^{\mathcal{F}}) \subseteq \mathsf{traces}(I, R)$.*
Proof. Follows from the fact that $\mathsf{walk}_I^{\mathcal{F}} \subseteq R$. □

Lemma 3.7 (Traces reach frontier)**.** *For a frontier $\mathcal{F}$ and state s,*

$$\forall \pi \in \mathsf{traces}(\{s\}, R).\ \exists i.\ \pi_0^i \in \mathcal{F} \ \Rightarrow\ \mathsf{walk}_{\{s\}}^{\mathcal{F}} \text{ is well-founded}$$

Proof. Using Theorem 1.4 letting $I = \{s\}$. Then using the law of excluded middle and Lemmas 3.5 and 3.6. □

Definition 3.1 (Frontier of S)**.** *If $S \vdash \mathsf{AF}\varphi$, then we denote by $\mathsf{front}(S)$ the corresponding frontier needed to satisfy $S \vdash \mathsf{AF}\varphi$. That is, the greatest fixedpoint: $\quad \nu s.\ \langle R, \{s\}\rangle \vdash \varphi$.*

The following two Lemmas are used extensively in Theorem 3.10, allowing us to decompose I into individual states s, and compose individual states s into a set I.

Lemma 3.8 (Decomposability of $\vdash$)**.** *For all $\varphi, M, s \in I$, $\langle R, I\rangle \vdash \varphi \ \Rightarrow\ \langle R, \{s\}\rangle \vdash \varphi$.*

Proof. By induction on φ. 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. □

Lemma 3.9 (Composability of $\vdash$)**.** *For all φ, I, $(\forall s \in I.\ \langle R, \{s\}\rangle \vdash \varphi) \ \Rightarrow\ \langle R, I\rangle \vdash \varphi$.*

Proof. By induction on φ.

Cases $\alpha, \wedge$**:** Trivial.

Case $\varphi' \vee \varphi''$**:** 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 $\langle R, \{s\}\rangle \vdash \varphi'$ that $s \in I'$ and otherwise ($\langle R, \{s\}\rangle \vdash \varphi''$) $s \in I''$. Reasoning is then straight-forward, using Lemma 3.8 and ind. hyp.

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

$$\mathcal{F} \equiv \{t \mid \exists s.\ s \in I \ \wedge\ t \in \mathsf{front}(\{s\})\}$$

Now $\mathsf{walk}_I^{\mathcal{F}}$ is well-founded (by Theorem 1.4) and $\mathcal{F} \vdash \varphi'$ (Lemma 3.8).

Case $\mathbf{A}(\varphi'\mathbf{W}\varphi'')$: Once again, we use the axiom of choice to define a combined frontier:

$$\mathcal{F} \equiv \{s \mid \exists s_0.\ s_0 \in I . s \in \mathsf{front}(\{s_0\}) \ \wedge\ \forall t, t'.\ \mathsf{walk}_{\{s_0\}}^{\mathsf{front}(\{s_0\})}(t, t') \Rightarrow \langle R, \{t\}\rangle \vdash \varphi')\}$$

Now, following the AW semantics, we must show two things:

- $\forall t, t'.\ \mathsf{walk}_I^{\mathcal{F}}(t, t') \Rightarrow \langle R, \{t\}\rangle \vdash \varphi'$. By induction we show $\exists s \in I.\ \mathsf{walk}_{\{s\}}^{\mathcal{F}'}(t, t')$. We know that $\forall s', s''.\ \mathsf{walk}_{\{s\}}^{\mathcal{F}'}(s', s'') \Rightarrow \langle R, \{s'\}\rangle \vdash \varphi'$, so $\langle R, \{t\}\rangle \vdash \varphi'$.
- $\langle R, \mathcal{F}\rangle \vdash \varphi''$. Follows from the definition of $\mathcal{F}$ above, and Lemma 3.8.

□

Theorem 3.10 ($\Leftarrow$ direction)**.** *For all φ, I, R, for all $s \in I$*

$$\langle R, \{s\}\rangle \vdash \varphi \quad \Leftarrow \quad s \vDash \varphi$$

Proof. By structural induction on φ. The $\wedge, \vee, \alpha$ case splits are trivial. To prove the AF case, using classical reasoning, we define a frontier $\mathcal{F}$ to be the set of all states such that the subformula φ' holds. We then have two obligations:

1. Show $\mathsf{walk}_{\{s\}}^{\mathcal{F}}$ is well-founded. This holds by using Lemma 3.7: since every π in $\mathsf{walk}_{\{s\}}^{\mathcal{F}}$ starting from s has an index n where $\pi_0^n \in \mathcal{F}$, $\mathsf{walk}_{\{s\}}^{\mathcal{F}}$ is well-founded.
2. Show that $\langle R, \mathcal{F}\rangle \vdash \varphi'$. We use Lemma 3.9 to show that

$$\forall s \in \mathcal{F}.\ \langle R, \{s\}\rangle \vdash \varphi' \quad \Rightarrow \quad \langle R, \mathcal{F}\rangle \vdash \varphi'$$

For the $\mathsf{A}[\varphi'\ \mathsf{W}\ \varphi'']$ case, we first define a frontier by the axiom of choice:

$$\mathcal{F} \equiv \{t \mid \exists \pi.\ \exists n. \pi_0^n = t \wedge \langle R, \{t\}\rangle \vdash \varphi''\}$$

Now, the semantics of AW gives us two cases:

- $\forall s_1, s_2.\ \mathsf{walk}_{\{s\}}^{\mathcal{F}}(s_1, s_2) \Rightarrow \langle R, \{s_1\}\rangle \vdash \varphi'$
  We use an equivalent definition of $\mathsf{walk}_{\{s\}}^{\mathcal{F}}$ which is parameterized by m, the number of steps taken from s. We can then show inductively (over m) that there exists a trace π such that $\pi_0^m = s_1$ and $\langle R, \{s_1\}\rangle \nvdash \varphi''$ (because if φ'' held, s_1 would be in frontier $\mathcal{F}$ and $\mathsf{walk}_{\{s\}}^{\mathcal{F}}$ would not hold).
  Now, for this π, it must be the case (because $s \vDash \mathsf{A}[\varphi'\ \mathsf{W}\ \varphi'']$) that either:
  1. $\forall i.\ \pi_0^i \vDash \varphi'$.
     With the inductive hypothesis, we can easily conclude that $\langle R, \{s_1\}\rangle \vdash \varphi'$.
  2. $\exists j.\ \pi_0^j \vDash \varphi'' \ \wedge\ \forall i < j.\ \pi_0^i \vDash \varphi'$.
     It must be the case that $j > m$ because recall that φ'' does not hold up to m.
- $\langle R, \mathcal{F}\rangle \vdash \varphi''$
  Follows from Lemma 3.9, the definition of $\mathcal{F}$ and the inductive hypothesis.

□

3.3 Guarded transition systems

Now that we have shown our relational formulation $\vdash$ to be equivalent to the standard $\forall$CTL semantics $\models$, we show the correctness of our encoding $\mathcal{E}$ using $\vdash$. In previous sections $\mathcal{E}$ was given informally as procedures in an imperative language. We now formalize $\mathcal{E}$ as a function $\mathcal{E} : M \times \mathcal{M} \times \varphi \to \mathcal{G}^M_{\mathcal{M},\varphi}$ from the original transition system M, set of rank functions $\mathcal{M}$ and $\forall$CTL property φ to a guarded transition system $\mathcal{G}$ defined below. In $\mathcal{G}$ we track procedure call arguments and return values as part of the configuration.

Definition 3.2. *A* guarded transition system $\mathcal{G} = (N, V, C^0, \Theta)$ *is a finite set of control points N, a finite set of typed variables V, an initial configuration predicate C^0 and a transition predicate Θ over unprimed (V) and primed ($\hat{V} \equiv \{\hat{x} \mid x \in V\}$) variables. The set of variables includes one special variable $\mathsf{nd} : N$ denoting the current control point. A configuration $c \in C$ is a valuation (i.e. a mapping) for the variables.*

A primed configuration $\hat{c}$ is defined similar to primed variables. For configurations c_1, c_2 we say $c_1 \rightsquigarrow c_2$ if and only if $\Theta_\varphi(c_1)(\hat{c_2})$. Propositions in Θ over unprimed variables can be thought of as "guards" and propositions over primed variables can be thought of as "actions." Accordingly, we use the notation $n_1\{g\} \xrightarrow{a} n_2$ to mean

$$\lambda c_1.\ \lambda c_2.\ \ c_1(\mathsf{nd}) = n_1 \wedge g(c_1) \wedge a(c_1 \cup \hat{c_2}) \wedge c_2(\mathsf{nd}) = n_2$$

3.3.1 Encoding

We now formally define the encoding of the task of verifying whether an $\forall$CTL property φ holds of a machine $M = (S, R, I)$ as a particular property ("returning false"—defined later) of a guarded transition system. For φ and M, the encoding as a guarded transition system $\mathcal{G}^M_{\mathcal{M},\varphi} = (N_\varphi, V_\varphi, C^0_\varphi, \Theta_\varphi)$, is parameterized by a finite set of measures $\mathcal{M}$, and defined as follows:

$$\begin{aligned}
N_\varphi &\equiv \{\mathsf{en}, \mathsf{ex}\} \times \kappa \times \mathsf{sub}(\varphi) \\
V_\varphi &\equiv \{\mathsf{nd} : N_\varphi\} \cup \bigcup_{(_,\kappa,\psi)\in N_\varphi} \{\mathsf{rv}^\kappa_\psi : \mathsf{bool},\ \ \sigma^\kappa_\psi : S,\ \ \mathsf{dup}^\kappa_\psi : \mathsf{bool},\ \ `\sigma^\kappa_\psi : S\} \\
C^0_\varphi &\equiv \mathsf{nd} = (\mathsf{en}, \bot, \varphi) \wedge \sigma^\bot_\varphi \in I \wedge \bigwedge_{(_,\kappa,\psi)\in N_\varphi} \mathsf{dup}^\kappa_\psi = \mathsf{false} \\
\Theta_\varphi &\equiv \text{(see Figure 3.2)}
\end{aligned}$$

(Recall the discussion of subformula contexts in Section 1.3.)
We use en and ex to distinguish the entry point of a procedure call from the exit point of a procedure call.

Lemma 3.11 (n decidability)**.** *For every $n_1, n_2 \in N_\varphi$, $n_1 = n_2$ is decidable.*

Proof. Follows from the fact that both $\kappa_1 = \kappa_2$ and $\varphi_1 = \varphi_2$ are decidable, and we assume that atomic proposition equality is decidable. □

The transition predicate Θ_φ (Figure 3.2) is parameterized by $\mathcal{M}$ (but we will write Θ_φ instead of $\Theta_\varphi(\mathcal{M})$ for notational convenience), defined inductively over the structure of φ and κ, and given over an alphabet of unprimed variables V_φ and primed variables $\hat{V}_\varphi$. We

$$
\begin{array}{rcll}
\Theta^{\kappa}_{\alpha} & \equiv & & (\mathsf{en},\kappa,\alpha)\{\alpha(\sigma^{\kappa}_{\alpha})\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\alpha}=\mathsf{true}} (\mathsf{ex},\kappa,\alpha) \\
 & & \vee & (\mathsf{en},\kappa,\alpha)\{\neg\alpha(\sigma^{\kappa}_{\alpha})\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\alpha}=\mathsf{false}} (\mathsf{ex},\kappa,\alpha) \\
\\
\Theta^{\kappa}_{\psi_1\wedge\psi_2} & \equiv & \Theta^{\mathsf{L}\,\kappa}_{\psi_1} \vee \Theta^{\mathsf{R}\,\kappa}_{\psi_2} \quad \vee & (\mathsf{en},\kappa,\psi_1\wedge\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\sigma}^{\mathsf{L}\,\kappa}_{\psi_1}=\sigma^{\kappa}_{\psi_1\wedge\psi_2}} (\mathsf{en},\mathsf{L}\,\kappa,\psi_1) \\
 & & \vee & (\mathsf{en},\kappa,\psi_1\wedge\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\sigma}^{\mathsf{R}\,\kappa}_{\psi_2}=\sigma^{\kappa}_{\psi_1\wedge\psi_2}} (\mathsf{en},\mathsf{R}\,\kappa,\psi_2) \\
 & & \vee & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\psi_1\wedge\psi_2}=\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}} (\mathsf{ex},\kappa,\psi_1\wedge\psi_2) \\
 & & \vee & (\mathsf{ex},\mathsf{R}\,\kappa,\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\psi_1\wedge\psi_2}=\mathsf{rv}^{\mathsf{R}\,\kappa}_{\psi_2}} (\mathsf{ex},\kappa,\psi_1\wedge\psi_2) \\
\\
\Theta^{\kappa}_{\psi_1\vee\psi_2} & \equiv & \Theta^{\mathsf{L}\,\kappa}_{\psi_1} \vee \Theta^{\mathsf{R}\,\kappa}_{\psi_2} \quad \vee & (\mathsf{en},\kappa,\psi_1\vee\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\sigma}^{\mathsf{L}\,\kappa}_{\psi_1}=\sigma^{\kappa}_{\psi_1\vee\psi_2}} (\mathsf{en},\mathsf{L}\,\kappa,\psi_1) \\
 & & \vee & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\psi_1\vee\psi_2}=\mathsf{true}} (\mathsf{ex},\kappa,\psi_1\vee\psi_2) \\
 & & \vee & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\neg\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}\} \xrightarrow{\hat{\sigma}^{\mathsf{L}\,\kappa}_{\psi_2}=\sigma^{\kappa}_{\psi_1\vee\psi_2}} (\mathsf{en},\mathsf{R}\,\kappa,\psi_2) \\
 & & \vee & (\mathsf{ex},\mathsf{R}\,\kappa,\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\psi_1\vee\psi_2}=\mathsf{rv}^{\mathsf{R}\,\kappa}_{\psi_2}} (\mathsf{ex},\kappa,\psi_1\vee\psi_2) \\
\\
\Theta^{\kappa}_{\mathsf{AF}\psi_1} & \equiv & \Theta^{\mathsf{L}\,\kappa}_{\psi_1} \quad \vee & (\mathsf{en},\kappa,\mathsf{AF}\psi_1)\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\mathsf{AF}\psi_1}=\mathsf{true}} (\mathsf{ex},\kappa,\mathsf{AF}\psi_1) \\
 & & \vee & (\mathsf{en},\kappa,\mathsf{AF}\psi_1)\{\mathsf{true}\} \xrightarrow{\hat{\sigma}^{\mathsf{L}\,\kappa}_{\psi_1}=\sigma^{\kappa}_{\mathsf{AF}\psi_1}} (\mathsf{en},\mathsf{L}\,\kappa,\psi_1) \\
 & & \vee & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\mathsf{AF}\psi_1}=\mathsf{true}} (\mathsf{ex},\kappa,\mathsf{AF}\psi_1) \\
 & & \bigvee_{s,s'\in R} & (\mathsf{en},\kappa,\mathsf{AF}\psi_1)\{\neg\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1} \wedge \sigma^{\kappa}_{\mathsf{AF}\psi_1}=s\} \xrightarrow{\hat{\sigma}^{\kappa}_{\mathsf{AF}\psi_1}=s'} (\mathsf{en},\kappa,\mathsf{AF}\psi_1) \\
 & & \bigvee_{s,s'\in R} & (\mathsf{en},\kappa,\mathsf{AF}\psi_1)\{\neg\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1} \wedge \neg\mathsf{dup}^{\kappa}_{\mathsf{AF}\psi_1} \wedge \sigma^{\kappa}_{\mathsf{AF}\psi_1}=s\} \\
 & & & \xrightarrow{\hat{\mathsf{dup}}^{\kappa}_{\mathsf{AF}\psi_1}=\mathsf{true}\wedge\hat{\ }\sigma^{\kappa}_{\mathsf{AF}\psi_1}=\sigma^{\kappa}_{\mathsf{AF}\psi_1}\wedge\hat{\sigma}^{\kappa}_{\mathsf{AF}\psi_1}=s'} (\mathsf{en},\kappa,\mathsf{AF}\psi_1) \\
 & & \vee & (\mathsf{en},\kappa,\mathsf{AF}\psi_1)\{\neg\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1} \wedge \mathsf{dup}^{\kappa}_{\mathsf{AF}\psi_1} \wedge \nexists f\in\mathcal{M}.\, f(\sigma^{\kappa}_{\mathsf{AF}\psi_1}) \prec f(\hat{\ }\sigma^{\kappa}_{\mathsf{AF}\psi_1})\} \\
 & & & \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\mathsf{AF}\psi_1}=\mathsf{false}} (\mathsf{ex},\kappa,\mathsf{AF}\psi_1) \\
\\
\Theta^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]} & \equiv & \Theta^{\mathsf{L}\,\kappa}_{\psi_1} \vee \Theta^{\mathsf{R}\,\kappa}_{\psi_2} \quad \vee & (\mathsf{en},\kappa,\mathsf{A}[\psi_1\mathsf{W}\psi_2])\{\mathsf{true}\} \xrightarrow{\hat{\sigma}^{\mathsf{L}\,\kappa}_{\psi_1}=\sigma^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}} (\mathsf{en},\mathsf{L}\,\kappa,\psi_1) \\
 & & \bigvee_{s,s'\in R} & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1} \wedge \sigma^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}=s\} \xrightarrow{\hat{\sigma}^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}=s'} (\mathsf{en},\kappa,\mathsf{A}[\psi_1\mathsf{W}\psi_2]) \\
 & & \vee & (\mathsf{ex},\mathsf{L}\,\kappa,\psi_1)\{\neg\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}\} \xrightarrow{\hat{\sigma}^{\mathsf{R}\,\kappa}_{\psi_2}=\sigma^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}} (\mathsf{en},\mathsf{R}\,\kappa,\psi_2) \\
 & & \vee & (\mathsf{ex},\mathsf{R}\,\kappa,\psi_2)\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}=\mathsf{rv}^{\mathsf{R}\,\kappa}_{\psi_2}} (\mathsf{ex},\kappa,\mathsf{A}[\psi_1\mathsf{W}\psi_2]) \\
 & & \vee & (\mathsf{en},\kappa,\mathsf{A}[\psi_1\mathsf{W}\psi_2])\{\mathsf{true}\} \xrightarrow{\hat{\mathsf{rv}}^{\kappa}_{\mathsf{A}[\psi_1\mathsf{W}\psi_2]}=\mathsf{true}} (\mathsf{ex},\kappa,\mathsf{A}[\psi_1\mathsf{W}\psi_2])
\end{array}
$$

Figure 3.2: The transition predicate Θ_φ for a guarded transition system $\mathcal{G}^{M}_{\mathcal{M},\varphi}$ encoding for an ∀CTL property φ.

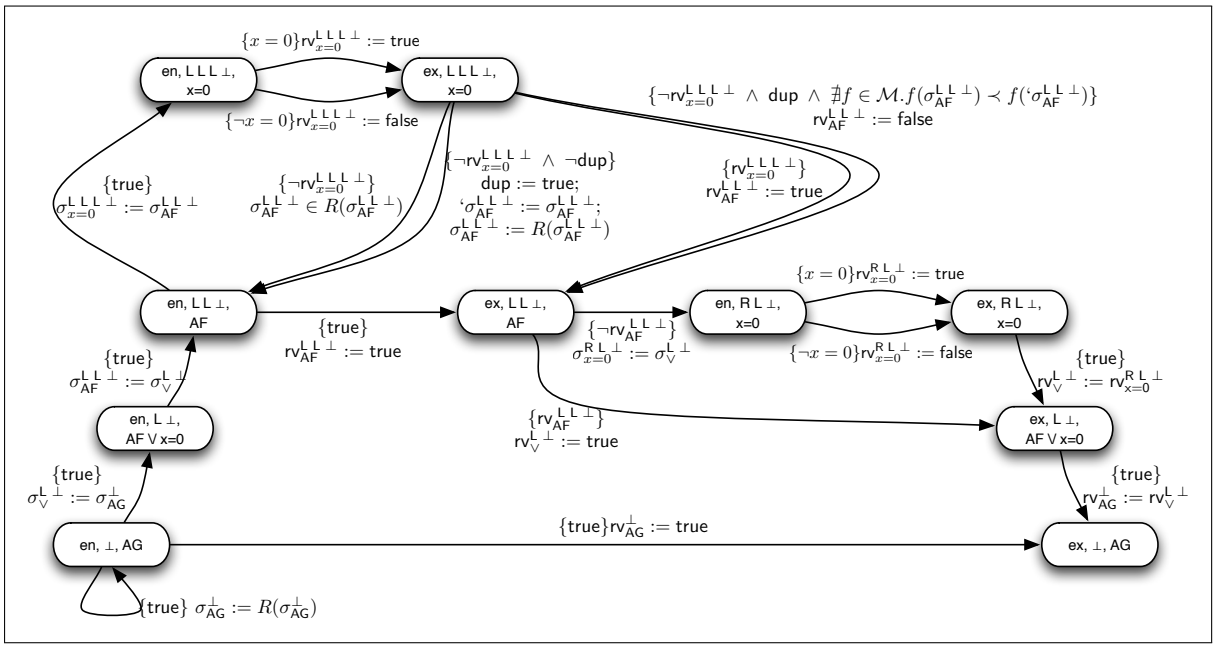

Figure 3.3: Encoding of Example 2.1.

often will refer to the guarded transition system in which $\mathcal{M}$ has not yet been specified. We use the notation $\mathcal{G}^{M}_{(\cdot),\varphi} \equiv \lambda x.\, \mathcal{G}^{M}_{x,\varphi}$.

Example. *For Example 2.1, we replace implication with disjunction in the formula to obtain* $\varphi \equiv \mathsf{AG}[\mathsf{AF}(x = 0) \vee x = 0]$. *Furthermore, we use the following abbreviations:*

$$\begin{aligned} \mathsf{AG} &\equiv \mathsf{AG}[\mathsf{AF}(x = 0) \vee x = 0] \\ \vee &\equiv \mathsf{AF}(x = 0) \vee x = 0 \\ \mathsf{AF} &\equiv \mathsf{AF}(x = 0) \end{aligned}$$

The encoding we obtain is as follows:

$$\begin{aligned} N_\varphi &\equiv \{(\mathsf{en}, \bot, \mathsf{AG}[\mathsf{AF}(x = 0) \vee x = 0]),\ (\mathsf{ex}, \bot, \mathsf{AG}[\mathsf{AF}(x = 0) \vee x = 0]), \\ &\quad (\mathsf{en}, \mathsf{L}\ \bot, \mathsf{AF}(x = 0) \vee x = 0),\ (\mathsf{ex}, \mathsf{L}\ \bot, \mathsf{AF}(x = 0) \vee x = 0), \\ &\quad (\mathsf{en}, \mathsf{L}\ \mathsf{L}\ \bot, \mathsf{AF}(x = 0)),\ (\mathsf{ex}, \mathsf{L}\ \mathsf{L}\ \bot, \mathsf{AF}(x = 0)), \\ &\quad (\mathsf{en}, \mathsf{L}\ \mathsf{L}\ \mathsf{L}\ \bot, x = 0),\ (\mathsf{ex}, \mathsf{L}\ \mathsf{L}\ \mathsf{L}\ \bot, x = 0), \\ &\quad (\mathsf{en}, \mathsf{R}\ \mathsf{L}\ \bot, x = 0),\ (\mathsf{ex}, \mathsf{R}\ \mathsf{L}\ \bot, x = 0)\} \\ C^0_\varphi &\equiv \mathsf{nd} = (\mathsf{en}, \bot, \mathsf{AG}) \wedge \sigma^{\bot}_{\mathsf{AG}} \in I \wedge \mathsf{dup} = \mathsf{false} \\ \Theta_\varphi &\equiv \textit{(see Figure 3.3)} \end{aligned}$$

Since there is only one AF *case, we assume only one* dup *variable. The transition predicate* Θ_φ *is given in Figure 3.3. Recall that* $\mathsf{AG}\varphi = \mathsf{A}[\varphi\ \mathsf{W}\ \mathsf{false}]$. *Consequently, some arcs given by* Θ_{AW} *in Figure 3.2 are unneeded, and ommitted in Figure 3.3. In two cases we use the notation "*$\sigma^{\bot}_{\mathsf{AG}} := R(\sigma^{\bot}_{\mathsf{AG}})$*," which is to say that a successor in* R *is chosen nondeterministically.*

3.3.2 Soundness of the encoding

Definition 3.3 (Execution, complete execution)**.** *An execution $c_0, c_1, c_2, ...$ is such that $c_0 \in C_0$ and $\forall i \in \mathbf{N}.\ c_i \rightsquigarrow c_{i+1}$. For $\mathcal{G}^M_{\mathcal{M},\varphi}$ we say that a finite execution $c_0, ..., c_n$ is a* complete execution *if $c_n(\mathsf{nd}) = (\mathsf{ex}, \bot, \varphi)$, and that the complete execution* returns true *if $c_n(\mathsf{rv}^\bot_\varphi) = \mathit{true}$.*

Definition 3.4 ("Cannot return false")**.** *We say that $\mathcal{G}^M_{\mathcal{M},\varphi}$* cannot return false *if every complete execution* returns true*.*

Before proceeding to the main Lemma 3.15, we first must prove the following lemmas. First, a formal version of INV_2:

Lemma 3.12 (INV_2)**.** *For all $\varphi, \mathcal{M}, R, s$, $\mathcal{G}^{(S,R,\{s\})}_{\mathcal{M},\varphi}$ can return true.*

Proof. By induction on φ. □

The next two lemmas are needed to make our inductive argument in the main Lemma 3.15. For a given guarded transition system $\mathcal{G}$ we will need to be able to argue about the behavior of *sub*systems (specifically, whether they can return false). This requires us to establish a homomorphic mapping between the two, which is not surprising given that $\mathcal{G}$ is defined inductively.

Lemma 3.13 (Homomorphic mapping)**.** *For every $M = (S, R, I)$,$\mathcal{M}$, $\forall$ CTL property φ and $\psi \in \mathit{isub}(\varphi)$ with path κ, there exists a homomorphic mapping H between $\mathcal{G}^M_{\mathcal{M},\psi}$ and $\mathcal{G}^M_{\mathcal{M},\varphi}$ such that for every pair of configurations c, c',*

$$c(\mathsf{nd}), c'(\mathsf{nd}) \in (\{\mathsf{en}, \mathsf{ex}\} \times \kappa \times \mathit{sub}(\psi)) \wedge \Theta_\varphi(c)(\hat{c'}) \ \Rightarrow\ \exists d, d'.\ \Theta^\kappa_\psi(d)(\hat{d'}) \wedge H(c) = d \wedge H(c') = d'$$

H projects out variables in $V_\varphi \setminus V_\psi$ and replaces $\mathsf{L}\,\kappa$ (or $\mathsf{R}\,\kappa$) with κ.

Lemma 3.14 ($\mathcal{G}$ induction)**.** *For every $M = (S, R, I)$, $\mathcal{M}$, $\forall$ CTL property φ and $\psi \in \mathit{isub}(\varphi)$ with path κ, if a configuration c where $c(\mathsf{nd}) = (\mathsf{en}, \kappa, \psi)$ is reachable in $\mathcal{G}^M_{\mathcal{M},\varphi}$, then a complete set of executions for $\mathcal{G}^{(S,R,\{\sigma^\kappa_\psi\})}_{\mathcal{M},\psi}$ can be obtained from the set of executions of $\mathcal{G}^M_{\mathcal{M},\varphi}$.* *Proof. By using the homomorphic mapping in Lemma 3.13 and Lemma 3.12.* □

Lemma 3.15. *For a transition system $M = (S, R, I)$ and $\forall$ CTL property φ, and guarded transition system $\mathcal{G}^M_{(\cdot),\varphi}$*

$$\exists \mathcal{M}.\ \mathcal{G}^M_{\mathcal{M},\varphi} \textit{ cannot return } \mathit{false} \ \Rightarrow\ \langle R, I\rangle \vdash \varphi.$$

Proof. By induction on φ and Θ_φ, using Lemma 3.14 in the induction. In the base case we have that

$$\exists \mathcal{M}.\ \mathcal{G}^M_{\mathcal{M},\alpha} \text{ cannot return } \mathsf{false} \Rightarrow \langle R, I\rangle \vdash \alpha$$

Here $\mathsf{nd} \in (\{\mathsf{en}, \mathsf{ex}\} \times \{\bot\} \times \{\alpha\})$, i.e. there are only two control points. Every execution begins with some $c_0 \in C^0_\alpha$, so $c_0(\mathsf{nd}) = (\mathsf{en}, \bot, \alpha)$. By the definition of $\Theta^\bot_\alpha$, there are only two classes of transitions from $(\mathsf{en}, \bot, \alpha)$, both leading to $(\mathsf{ex}, \bot, \alpha)$. So every complete execution must be of length 2. Consider an execution c_1, c_2. $C^0_\alpha(c_1)$ implies that $c_1(\sigma^\bot_\alpha) \in I$. The LHS above indicates that for every execution, $c_2(\mathsf{rv}^\bot_\alpha) = \mathsf{true}$. This must mean that $\alpha(c_1(\sigma^\bot_\alpha))$. Consequently, α holds of every initial state, and so $\langle R, I\rangle \vdash \alpha$.

The proof proceeds by induction:

$$\forall \psi_i \in \mathsf{isub}(\psi).\ (\exists \mathcal{M}.\ \mathcal{G}^M_{\mathcal{M},\psi_i} \text{ cannot return } \mathsf{false} \Rightarrow \langle R, I\rangle \vdash \psi_i) \Rightarrow \exists \mathcal{M}.\ \mathcal{G}^M_{\mathcal{M},\psi} \text{ cannot return } \mathsf{false} \Rightarrow \langle R, I\rangle \vdash \psi$$

Notice that the hypothesis holds for all initial context path κ. This lets us use a proof of each $\mathcal{G}^M_{\mathcal{M},\psi_i}$ using $\bot$, at the next step in the induction where either $(\mathsf{L}\ \bot)$ or $(\mathsf{R}\ \bot)$ is used. The cases are as follows:

Case $\psi = \psi_1 \wedge \psi_2$**:** 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 ψ_1. By the definition of $\mathcal{G}^M_{\mathcal{M},\psi_1\wedge\psi_2}$, every complete execution $c_0, ..., c_n$ is such that $C^0_{\psi_1\wedge\psi_2}(c_0)$ and that $c_n(\mathsf{rv}^\bot_{\psi_1\wedge\psi_2}) = \mathsf{true}$. Moreover, $c_0(\mathsf{nd}) = (\mathsf{en}, \bot, \psi_1 \wedge \psi_2)$ and $c_n(\mathsf{nd}) = (\mathsf{ex}, \bot, \psi_1 \wedge \psi_2)$.
Consider the subset of all complete executions of $\mathcal{G}^M_{\mathcal{M},\psi_1\wedge\psi_2}$:

$$\begin{aligned} E \equiv \{c_0, c_1, ..., c_{n-1}, c_n \mid\ & c_1(\mathsf{nd}) = (\mathsf{en}, \mathsf{L}\ \kappa, \psi_1) \wedge c_{n-1}(\mathsf{nd}) = (\mathsf{ex}, \mathsf{L}\ \kappa, \psi_1) \wedge \\ & \Theta^\bot_{\psi_1\wedge\psi_2}(c_0, c_1) \wedge \Theta^\bot_{\psi_1\wedge\psi_2}(c_{n-1}, c_n) \wedge \\ & \forall i \in [1, n-2].\ \Theta^{\mathsf{L}\ \bot}_{\psi_1}(c_i, c_{i+1})\} \end{aligned}$$

By Lemma 3.14 there is a homomorphic mapping between the complete set of executions of $\mathcal{G}^M_{\mathcal{M},\psi_1}$ and $\mathcal{G}^M_{\mathcal{M},\psi_1\wedge\psi_2}$. The same holds for $\mathcal{G}^M_{\mathcal{M},\psi_2}$. Since $\mathcal{G}^M_{\mathcal{M},\psi_1\wedge\psi_2}$ cannot return false, and $\mathsf{rv}^\bot_{\psi_1\wedge\psi_2}$ is given by $\mathsf{rv}^{\mathsf{L}\ \bot}_{\psi_1}$ in every execution in E, it must be the case that $\mathsf{rv}^{\mathsf{L}\ \bot}_{\psi_1} = \mathsf{true}$ in each c_{n-1} of an execution in E. So every complete execution of $\mathcal{G}^M_{\mathcal{M},\psi_1}$ cannot return false and thus (ind. hyp.) $\langle R, I\rangle \vdash \psi_1$.

Case $\psi = \psi_1 \vee \psi_2$**:** Consider an initial state $s_0 \in I$. By the definition of $\Theta^\bot_{\psi_1\vee\psi_2}$, we can partition the complete executions based on the initial value of $\sigma^\kappa_{\psi_1\vee\psi_2}$, and then into two classes:

$$\begin{aligned} E^{s_0}_L \equiv\ & \{c_0, c_1, ..., c_{n-1}, c_n \mid c_0(\sigma^\kappa_{\psi_1\vee\psi_2}) = s_0 \wedge \\ & c_1(\mathsf{nd}) = (\mathsf{en}, \psi_1, \mathsf{L}\ \kappa) \wedge c_{n-1}(\mathsf{nd}) = (\mathsf{ex}, \psi_1, \mathsf{L}\ \kappa) \wedge c_n(\mathsf{nd}) = (\mathsf{ex}, \psi_1 \vee \psi_2, \kappa)\} \\ E^{s_0}_R \equiv\ & \{c_0, c_1, ..., c_{n-1}, c_n, ..., c_{n+m-1}, c_{n+m} \mid c_0(\sigma^\kappa_{\psi_1\vee\psi_2}) = s_0 \wedge \\ & c_1(\mathsf{nd}) = (\mathsf{en}, \psi_1, \mathsf{L}\ \kappa) \wedge c_{n-1}(\mathsf{nd}) = (\mathsf{ex}, \psi_1, \mathsf{L}\ \kappa) \wedge c_n(\mathsf{nd}) = (\mathsf{en}, \psi_2, \mathsf{R}\ \kappa) \wedge \\ & c_{n+m-1}(\mathsf{nd}) = (\mathsf{ex}, \psi_2, \mathsf{R}\ \kappa) \wedge c_{n+m}(\mathsf{nd}) = (\mathsf{ex}, \psi_1 \vee \psi_2, \kappa)\} \end{aligned}$$

Claim 1: $\forall \epsilon_L \in E^{s_0}_L.\ c_{n-1}(\mathsf{rv}^{\mathsf{L}\ \kappa}_{\psi_1}) = \mathsf{true} \quad \bigvee \quad \forall \epsilon_R \in E^{s_0}_R.\ c_{n+m-1}(\mathsf{rv}^{\mathsf{R}\ \kappa}_{\psi_2}) = \mathsf{true}$
Pf: Asm not. $\exists \epsilon_L \in E^{s_0}_L.\ c_{n-1}(\mathsf{rv}^{\mathsf{L}\ \kappa}_{\psi_1}) = \mathsf{false} \quad \bigwedge \quad \exists \epsilon_R \in E^{s_0}_R.\ c_{n+m-1}(\mathsf{rv}^{\mathsf{R}\ \kappa}_{\psi_2}) = \mathsf{false}$.
Given the RHS, there exists $\epsilon \in E^{s_0}_R$ such that $c_{n+m}(\mathsf{rv}^\bot_{\psi_1\vee\psi_2}) = \mathsf{false}$ (def. of $\Theta^\bot_{\psi_1\vee\psi_2}$). Contradiction ($\mathcal{G}^M_{\mathcal{M},\psi_1\vee\psi_2}$ cannot return false).

By Claim 1 there are two cases. From the LHS we can show that $\mathcal{G}^{(S,R,\{s_0\})}_{\mathcal{M},\psi_1}$ cannot return false, and from the RHS we can show that $\mathcal{G}^{(S,R,\{s_0\})}_{\mathcal{M},\psi_2}$ cannot return false(using Lemma 3.14 in each case). Hence either $\langle R, \{s_0\}\rangle \vdash \psi_1$ or $\langle R, \{s_0\}\rangle \vdash \psi_2$ (ind. hyp.) so $\langle R, \{s_0\}\rangle \vdash \psi_1 \vee \psi_2$. We can use this reasoning for every $s_0 \in I$ to obtain a partitioning of I to satisfy $\langle R, I\rangle \vdash \psi_1 \vee \psi_2$.

Case $\psi = \mathbf{AF}\psi_1$**:** We must show that there exists a set $\mathcal{F}$ such that $\mathsf{walk}_I^{\mathcal{F}}$ is well-founded and $\langle R, \mathcal{F}\rangle \vdash \psi_1$. First we define the following:

$$\mathcal{F} \equiv \{s \mid \mathcal{G}^{(S,R,\{s\})}_{\mathcal{M},\psi_1} \text{ cannot return } \mathsf{false}\}$$

- *Claim:* $\mathsf{walk}_I^{\mathcal{F}}$ is well-founded.
  Pf. By showing there are no infinite sequences induced by $\mathsf{walk}_I^{\mathcal{F}}$. Assume not. Then there is an infinite sequence $s_0, s_1, ...$ such that $\forall i \geq 0.\ (s_i, s_{i+1}) \in \mathsf{walk}_I^{\mathcal{F}}$. By definition of $\mathcal{F}$ and an inductive argument over $\mathsf{walk}_I^{\mathcal{F}}$, we can show that $\forall i \geq 0.\ \mathcal{G}^{(S,R,\{s_i\})}_{\mathcal{M},\psi_1}$ can possibly return false. Given that $\mathsf{walk}_I^{\mathcal{F}} \subseteq R$, we also know that $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$. So by the definition of Θ_φ, we can show that there is an infinite execution of $\mathcal{G}^M_{\mathcal{M},\mathsf{AF}\psi_1}$,

$$\begin{aligned} c_0, c_0', c_0'', c_1, c_1', c_1'', ... \text{ s.t. } \forall i \geq 0. \qquad & c_i(\sigma) = s_i \\ \wedge\ & c_i(\mathsf{nd}) = (\mathsf{en}, \kappa, \mathsf{AF}\psi_1) \\ \wedge\ & c_i'(\mathsf{nd}) = (\mathsf{en}, \mathsf{L}\ \kappa, \psi_1) \\ \wedge\ & c_i''(\mathsf{nd}) = (\mathsf{ex}, \mathsf{L}\ \kappa, \psi_1) \end{aligned}$$

  Note that for every consecutive pair of states (s_i, s_{i+1}), there is also an execution in which $c_{i+1}(`\sigma) = s_i$ and $c_{i+1}(\sigma) = s_{i+1}$. Since $\mathcal{G}^M_{\mathcal{M},\mathsf{AF}\psi_1}$ cannot return false, it must be the case that $\forall i \geq 0.\ \exists f \in \mathcal{M}.\ f(s_{i+1}) \prec f(s)$. Contradiction.
- *Claim:* $\langle R, \mathcal{F}\rangle \vdash \psi_1$.
  Pf. Trivial, given the definition of $\mathcal{F}$ and the inductive hypothesis.

Case $\psi = \mathbf{A}(\psi_1 \mathbf{W} \psi_2)$**:** We must show that there exists a frontier $\mathcal{F}$ that satisfies the conditions in the AW case in Figure 3.1. First, a Lemma:

Lemma 3.16. *For every $(s_n, t) \in \mathsf{walk}_I^{\mathcal{F}}\ (s_n, t)$ 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 $\mathcal{F} \equiv \{s \mid \mathcal{G}^{(S,R,\{s\})}_{\mathcal{M},\psi_2} \text{ cannot return } \mathsf{false}\}$. What remains is to show that $\mathcal{F}$ satisfies the conditions in Figure 3.1:

- *Claim 1:* $\forall (s_n, t) \in \mathsf{walk}_I^{\mathcal{F}}.\ \langle R, \{s_n\}\rangle \vdash \psi_1$.
  Pf. Pick some $(s_n, t) \in \mathsf{walk}_I^{\mathcal{F}}$. By Lemma 3.16, 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: $\forall i \in [0, n].\mathcal{G}^{(S,R,\{s_i\})}_{\mathcal{M},\psi_1}$ cannot return false.
  Pf: By induction on the list $s_0, ..., s_n$, also maintaining the invariant that $(s_i, s_{i+1}) \in R$ and so

$$\Theta_\varphi = ... \vee (\mathsf{ex}, \psi_1, \mathsf{L}\ \kappa)\{\mathsf{rv}^{\mathsf{L}\ \kappa}_{\psi_1} \wedge \sigma^\kappa_{\mathsf{A}[\psi_1 \mathsf{W} \psi_2]} = s_i\} \xrightarrow{\hat{\sigma}^\kappa_{\mathsf{A}[\psi_1 \mathsf{W} \psi_2]} = s_{i+1}} (\mathsf{en}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa)$$

Base case: $\mathcal{G}^{(S,R,\{s_0\})}_{\mathcal{M},\psi_1}$ c.r. false. Define

$$E^{s_0} \equiv \{c_0, ..., c_{n-2}, c_{n-1}, c_n \mid c_0(\sigma) = c_{n-2}(\sigma) = s_0 \wedge c_{n-2}(\mathsf{nd}) = (\mathsf{ex}, \psi_1, \mathsf{L}\,\kappa) \wedge c_{n-1}(\mathsf{nd}) = (\mathsf{en}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa) \wedge c_n(\mathsf{nd}) = (\mathsf{ex}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa)\}$$

By definition of $\mathcal{F}$ and $\mathsf{walk}^{\mathcal{F}}_I$, there cannot be an execution in which $c_{n-2}(\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}) = \mathsf{false}$ So from E^{s_0}, we can obtain a complete set of traces for $\mathcal{G}^{(S,R,\{s_0\})}_{\mathcal{M},\psi_1}$ (Lemma 3.14), none of which return false.

Induction: For $s_0, ..., s_i$, $\mathcal{G}^{(S,R,\{s_i\})}_{\mathcal{M},\psi_1}$ c.r.f $\Rightarrow$ $\mathcal{G}^{(S,R,\{s_{i+1}\})}_{\mathcal{M},\psi_1}$ c.r.f. Consider the following set of executions:

$$E^{s_{i+1}} \equiv \{c_0, ..., c_1, c_2, c_3, ..., c_4, c_5 \mid c_0(\sigma) = s_0 \wedge c_1(\mathsf{nd}) = (\mathsf{ex}, \psi_1, \mathsf{L}\,\kappa) \wedge c_1(\sigma) = s_i \wedge c_2(\mathsf{nd}) = (\mathsf{en}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa) \wedge c_2(\sigma) = s_{i+1} \wedge c_3(\mathsf{nd}) = (\mathsf{en}, \psi_1, \mathsf{L}\,\kappa) \wedge c_3(\sigma) = s_{i+1} \wedge c_4(\mathsf{nd}) = (\mathsf{ex}, \psi_1, \mathsf{L}\,\kappa) \wedge c_4(\sigma) = s_{i+1}\}$$

This set is nonempty. c_3 is reachable because the ind. hyp. says that each $\mathcal{G}^{(S,R,\{s_i\})}_{\mathcal{M},\psi_1}$ c.r.f. and there is a transition in Θ_φ for each (s_i, s_{i+1}). By definition of $\mathcal{F}$ and $\mathsf{walk}^{\mathcal{F}}_I$, there cannot be an execution in which $c_3(\mathsf{rv}^{\mathsf{L}\,\kappa}_{\psi_1}) = \mathsf{false}$. So from $E^{s_{i+1}}$, we can obtain a complete set of traces for $\mathcal{G}^{(S,R,\{s_{i+1}\})}_{\mathcal{M},\psi_1}$ (Lemma 3.14), none of which return false.

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

$$E \equiv \{c_0, ..., c_{n-1}, c_n \mid c_0(\mathsf{nd}) = (\mathsf{en}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa) \wedge c_{n-1}(\mathsf{nd}) = (\mathsf{ex}, \psi_2, \mathsf{R}\,\kappa) \wedge c_n(\mathsf{nd}) = (\mathsf{ex}, \mathsf{A}[\psi_1 \mathsf{W} \psi_2], \kappa)\}$$

The executions in E cannot return false because $\mathcal{G}^{M}_{\mathcal{M},\mathsf{A}[\psi_1 \mathsf{W} \psi_2]}$ c.r. false. For every $s \in \mathcal{F}$, we can obtain a complete set of traces for $\mathcal{G}^{(S,R,\{s\})}_{\mathcal{M},\psi_2}$ (Lemma 3.14) from E. So by the ind. hyp. and Lemma 3.9, we find that $\langle R, \mathcal{F}\rangle \vdash \psi_2$.

□

From these lemmas we can prove Theorem 2.1.

Chapter 4

Evaluation

4.1 Implementation

In this section, we describe several optimizations implemented in the partial evaluation procedure PEval mentioned in Chapter 2. We will demonstrate each optimization as it pertains to Example 2.1, transforming the encoding in Figure 4.1 into the optimized encoding in Figure 2.2.

pc-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 $\mathcal{L}$, we can specialize with respect to the program counter similar to the φ-specialization discussed in Chapter 2. Consider the following example:

```
bool enc^κ_φ(pc, s) {
    switch(pc) {
       case 0:  goto lab_enc^κ_φ_0;
       case 1:  goto lab_enc^κ_φ_1;
       ...
       case n:  goto lab_enc^κ_φ_n;
    }
    pc = 0;
  lab_enc^κ_φ_0:  ...;
    pc = 1;
  lab_enc^κ_φ_1:  ...;
    ...
    pc = n;
  lab_enc^κ_φ_n:  ...;
}
```

```
bool enc^κ_φ_0(s) {
    goto lab_enc^κ_φ_0;
  lab_enc^κ_φ_0:  ...;
}
bool enc^κ_φ_1(s) {
    goto lab_enc^κ_φ_1;
    ...
  lab_enc^κ_φ_1:  ...;
}
...
bool enc^κ_φ_n(s) {
    goto lab_enc^κ_φ_n;
    ...
  lab_enc^κ_φ_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 onus from elaborate procedure summaries for a few procedures to compact per-procedure summaries for many more procedures. In practice

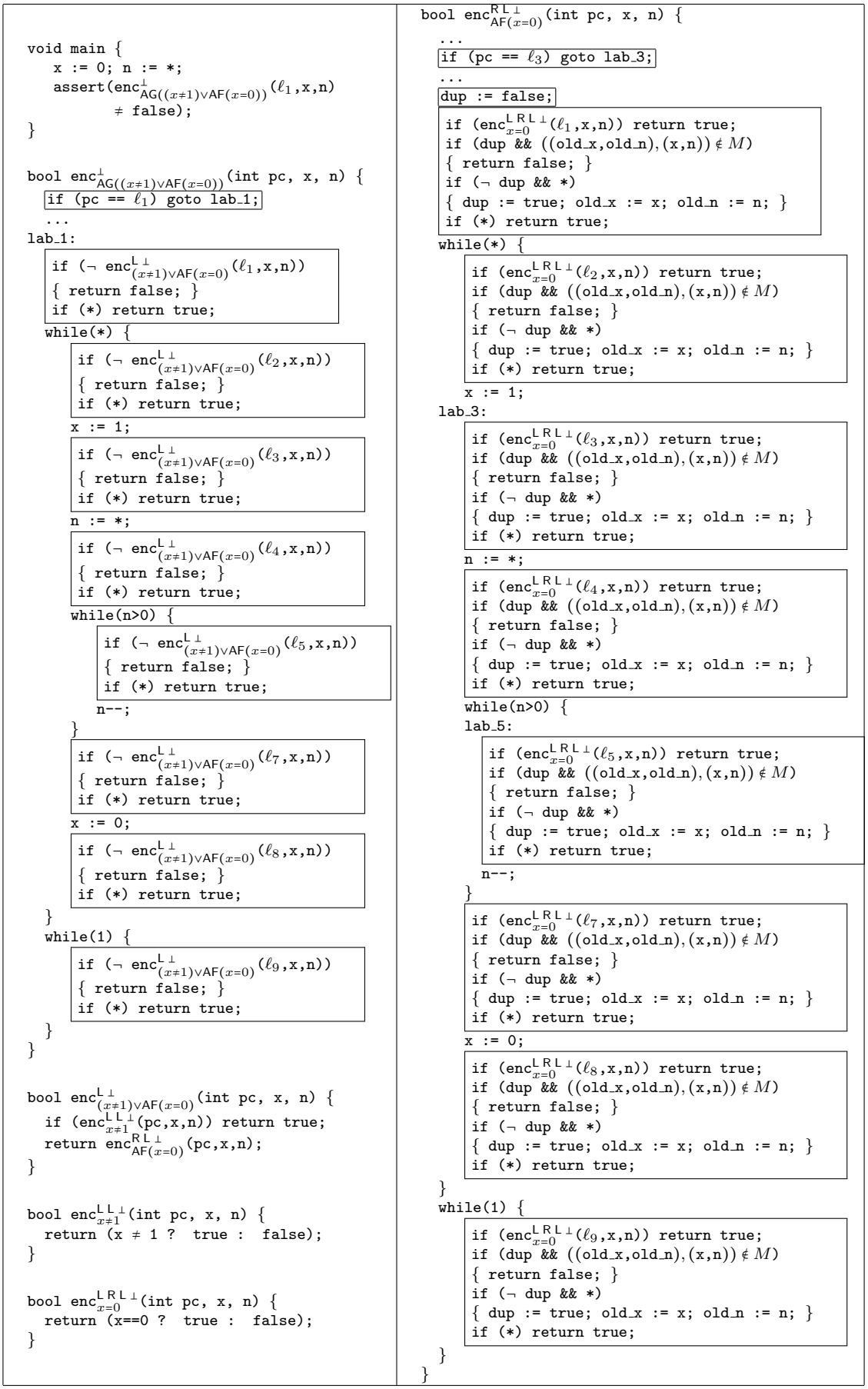

Figure 4.1: The encoding $\mathcal{E}$ of Example 2.1 (before the partial evaluation has been applied to obtain the output in Fig. 2.2). Recall that the property of interest was $\mathsf{AG}[(x = 1) \Rightarrow \mathsf{AF}(x = 0)]$.

(as discussed below) this is far preferable. Note that as future work we hope to implement a more lazy expansion and partial evaluation *á la* IMPACT [29]. This specialization can be seen when comparing, for example, $\texttt{enc}^{\mathsf{R}\,\mathsf{L}\,\perp}_{\mathsf{AF}(x=0)}\texttt{(pc,x,n)}$ in Figure 4.1 with $\texttt{enc}^{\mathsf{R}\,\mathsf{L}\,\perp}_{\mathsf{AF}(x=0)}\texttt{-3(x,n)}$ in Figure 2.2.

Inlining. Sub-procedures of $\mathcal{E}$ for non-temporal formulae can be inlined. For example, rather than the following procedures for the property $\mathsf{AG}(x = 1 \vee (y > 0 \wedge z < 0))$:

```
bool enc^{L R L κ}_{(y>0)}(int a, x, y, z) { return (y > 0); }
bool enc^{R R L κ}_{(z<0)}(int a, x, y, z) { return (z < 0); }
bool enc^{L L κ}_{(x=1)}(int a, x, y, z) { return (x==1); }
bool enc^{R L κ}_{(y>0∧z<0)}(int a, x, y, z) { ... }
bool enc^{L κ}_{(x=1∨(y>0∧z<0))}(int a, x, y, z) { ... }
bool enc^{κ}_{AG(x=1∨(y>0∧z<0))}(int a, x, y, z) { ... }
```

We can instead inline the non-temporal procedures and obtain:

```
bool enc^{κ}_{AG(x=1∨(y>0∧z<0))}(int a, x, y, z) {
  ...
  if (x == 1) return true;
  if (¬ y<0) return false;
  return (z<0 ?  true :  false);
  ...
}
```

For example, in Figure 2.2 we have inlined $\texttt{enc}^{\mathsf{L}\,\mathsf{R}\,\mathsf{L}\,\perp}_{x=0}$ within the body of $\texttt{enc}^{\mathsf{R}\,\mathsf{L}\,\perp}_{\mathsf{AF}(x=0)}$.

Ordering disjunction. For a disjunctive property $\varphi \vee \psi$, 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 $(\mathsf{AGAF}\ y = 1) \vee (x = 1)$. Clearly in most cases it is easier to show that the sub-procedure corresponding to the atomic proposition $(x = 1)$ cannot return false rather than showing that the $(\mathsf{AGAF}\ y = 1)$ cannot return 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 $\texttt{enc}^{\mathsf{L}\,\perp}_{(x\neq 1)\vee\mathsf{AF}(x=0)}$ in Figure 4.1.

Intra-procedural analysis. In our treatment of AF and 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 $\mathsf{AG}(x = 1 \vee (y > 0 \wedge z < 0))$ and the following fragment of $\texttt{enc}^{\kappa}_{\mathsf{AG}(x=1\vee(y>0\wedge z<0))}$:

```
1 bool enc^{κ}_{AG(x=1∨(y>0∧z<0))}(int a, int x, int y, int z):
2   ...
3   if (¬ enc^{L κ}_{(x=1∨(y>0∧z<0))}(a,x,y,z)) return false;
4   a := 56;
5   if (¬ enc^{L κ}_{(x=1∨(y>0∧z<0))}(a,x,y,z)) return false;
6   ...
7 }
```

Clearly, the assignment `a:=56` does not impact the truth value of $(x = 1 \vee (y > 0 \wedge z < 0))$, so if false can be returned on Line 3, then false can also be returned on Line 5. Also, if

false cannot be returned on Line 3, then false cannot be returned on Line 5. We apply a simple *intra*procedural analysis to remove superfluous calls such as the one on Line 5. This optimization can be seen in $\mathtt{enc}^{\perp}_{\mathsf{AG}[(x\neq 1)\vee\mathsf{AF}(x=0)]}$ and in $\mathtt{enc}^{\mathsf{R}\,\mathsf{L}\,\perp}_{\mathsf{AF}(x=0)}$ in Figure 2.2.

Program	LOC	Property	Prev. tool [14] Time	Prev. tool [14] Result	Our tool (Chap. 2) Time	Our tool (Chap. 2) Result
Acq/rel	14	$\mathsf{AG}(a \Rightarrow \mathsf{AF}b)$	103.48	✓	14.18	✓
Ex from Fig. 8 of [14]	34	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	209.64	✓	27.94	✓
Toy linear arith. 1	13	$p \Rightarrow \mathsf{AF}q$	126.86	✓	34.51	✓
Toy linear arith. 2	13	$p \Rightarrow \mathsf{AF}q$	>14400.00	**???**	6.74	✓
PostgreSQL smsrv	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	>14400.00	**???**	9.56	✓
PostgreSQL smsrv+bug	259	$\mathsf{AG}(p \Rightarrow \mathsf{AFAG}q)$	87.31	χ	47.16	χ
PostgreSQL pgarch	61	$\mathsf{AFAG}p$	31.50	✓	15.20	✓
Apache progress	314	$\mathsf{AG}(p \Rightarrow (\mathsf{AF} \vee \mathsf{AF}))$	685.34	✓	684.24	✓
Windows OS 1	180	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	901.81	✓	539.00	✓
Windows OS 4	327	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	**???**	1,114.18	✓
Windows OS 4	327	$(\mathsf{AF}a) \vee (\mathsf{AF}b)$	1,223.96	✓	100.68	✓
Windows OS 5	648	$\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$	>14400.00	**???**	>14400.00	**???**
Windows OS 7	13	$\mathsf{AGAF}p$	>14400.00	**???**	55.77	✓

Figure 4.2: Comparison between our tool and Cook *et al.* [14] 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 [14].

Program	LOC	Property	Prev. tool [14] Time	Prev. tool [14] Result	Our tool (Chap. 2) Time	Our tool (Chap. 2) #	Our tool (Chap. 2) Result
Ex. from [15]	5	$\mathsf{FG}p$	2.32	✓	1.98	2	✓
PostgreSQL dropbuf	152	$\mathsf{G}(p \Rightarrow \mathsf{F}q)$	53.99	✓	27.54	3	✓
Apache `accept` liveness	314	$\mathsf{G}p \Rightarrow \mathsf{GF}q$	>14400.00	**???**	197.41	3	✓
Windows OS 2	158	$\mathsf{FG}p$	16.47	✓	52.10	4	✓
Windows OS 2+bug	158	$\mathsf{FG}p$	26.15	χ	30.37	1	χ
Windows OS 3	14	$\mathsf{FG}p$	4.21	✓	15.75	2	✓
Windows OS 6	13	$\mathsf{FG}p$	149.41	✓	59.56	1	✓
Windows OS 6+bug	13	$\mathsf{FG}p$	6.06	χ	22.12	1	χ
Windows OS 8	181	$\mathsf{FG}p$	>14400.00	**???**	5.24	1	✓

Figure 4.3: Comparison between our tool and Cook *et al.* [14] on LTL benchmarks. For our tool, we use a recently described iterative symbolic determinization strategy [15] to prove LTL properties by using Alg. 2.2 as the underlying ∀CTL proof technique. The number of iterations is reported in the # column.

4.2 Experiments

In this section we report on experiments with a prototype tool that implements $\mathcal{E}$ from Fig. 2.1 as well as the refinement procedure from Algorithm 2.2. In our tool we have implemented $\mathcal{E}$ as a source-to-source translation using the CIL compiler infrastructure. We use SLAM [2] as our implementation of the safety prover, and RankFinder [31] as the rank function synthesis tool.

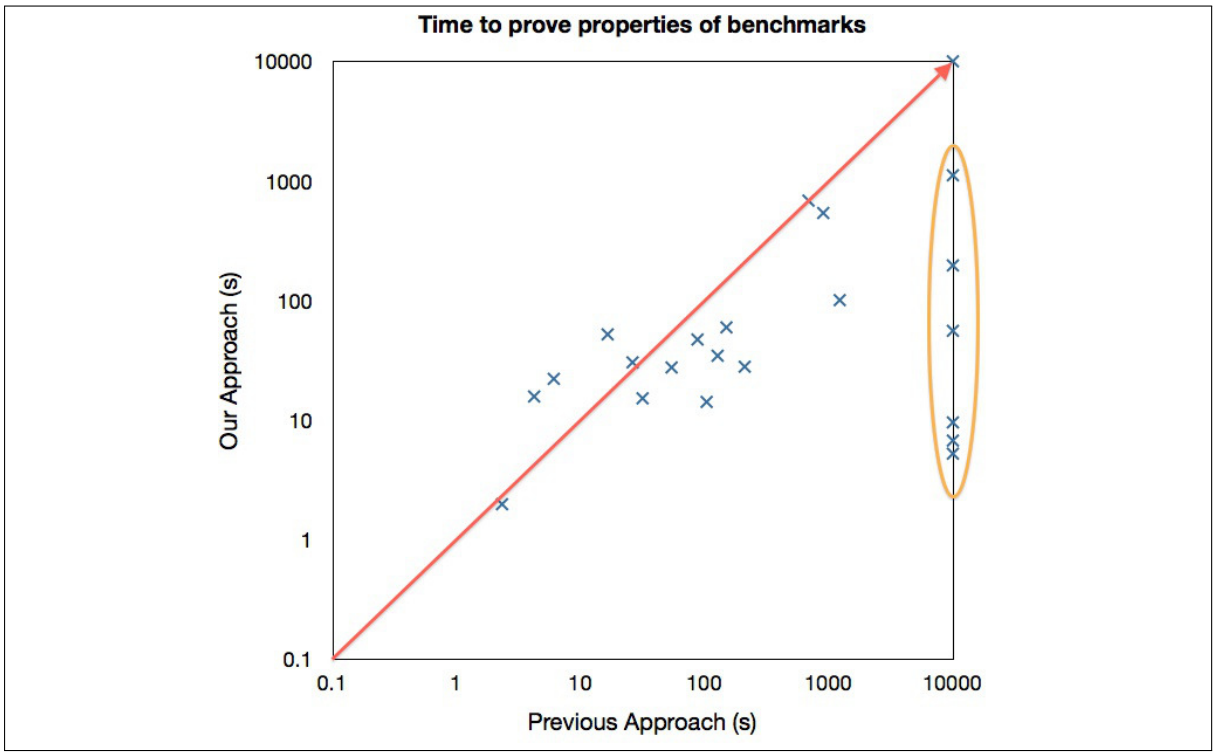

Figure 4.4: Comparison of the times (logarithmic scale) required to prove the property of each benchmark with the previous technique [14] versus our technique. The diagonal line indicates where the tools would have the same performance. The data points in the circle on the right are cases where the previous technique timed out after 4 hours.

We have drawn out a set of both ∀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 Fig. 8 in [14]. Sources of examples can be found in our technical report [16]. Heap commands from the original sources have been abstracted away using the approach due to Magill *et al.* [27]. 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 [30]. Support for variables that range over the natural numbers is crucial for this abstraction.

As previous mentioned in Section 1.1, 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 [24] 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 [23], as it is primarily designed to work for unnested existential properties (*e.g.* $\mathsf{EG}p$ or $\mathsf{EF}p$). We have also implemented the approach due to Chaki *et al.* [9]. 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 [33] 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 [1] or NuSMV [10] 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.* [14] can be used to prove LTL properties if used in combination with an LTL to Büchi automata conversion tool (*e.g.* [22]). To compare our approach to this tool we have used two sets of experiments: Fig. 4.2 displays the results on challenge problems in ∀CTL verification; Fig. 4.3 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 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 (Fig. 4.2) we have chosen properties that are equivalent in ∀CTL and LTL and then directly compared our procedure (Algorithm 2.2) to the tool in Cook *et al.* [14]. When comparing approaches on LTL verification problems (Fig. 4.3) we have used an iterative symbolic determinization strategy [15] which calls our Algorithm 2.2 on successively refined ∀CTL verification problems. The number of such iterations is given as column "#." in Fig. 4.3. For example, in the case of benchmark Windows OS 3, our procedure was called twice while attempting to prove a property of the form FGp.

A visual comparison is given in Figure 4.4. Using a logarithmic scale, we compare the times required to prove the property of each benchmark with the previous technique [14] (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.

Chapter 5

Conclusions

We have introduced a novel temporal reasoning technique for (potentially infinite-state) transition systems, with an implementation designed for systems described as programs. Our approach shifts the task of temporal reasoning to a program analysis problem. When an analysis is performed on the output of our encoding, it is effectively reasoning about the temporal and possibly branching behaviors of the original system. Consequently, we can use the wide variety of efficient program analysis tools to prove properties of programs. We have demonstrated the practical viability of the approach using industrial code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Acknowledgments. We would like to thank Josh Berdine, Matko Botinčan, Michael Greenberg, Daniel Kroening, Axel Legay, Rupak Majumdar, Peter O'Hearn, Joel Ouaknine, Matthew Parkinson, Nir Piterman, Andreas Podelski, Noam Rinetzky, and Hongseok Yang for valuable discussions regarding this work. We also thank the Gates Cambridge Trust for funding Eric Koskinen's Ph.D. degree program.

Bibliography

[1] Cadence SMV. `http://www.kenmcmil.com/smv.html`.

[2] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *EuroSys* (2006), pp. 73–85.

[3] Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P. W. Variance analyses from invariance analyses. In *POPL* (2007), pp. 211–224.

[4] Bernholtz, O., Vardi, M. Y., and Wolper, P. An automata-theoretic approach to branching-time model checking (extended abstract). In *CAV* (1994), pp. 142–155.

[5] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *PLDI* (2003), pp. 196–207.

[6] Bradley, A., Manna, Z., and Sipma, H. The polyranking principle. *Automata, Languages and Programming* (2005), 1349–1361.

[7] Burch, J., Clarke, E., et al. Symbolic model checking: 10^{20} states and beyond. *Information and computation 98*, 2 (1992), 142–170.

[8] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Compositional shape analysis by means of bi-abduction. In *POPL* (2009), pp. 289–300.

[9] Chaki, S., Clarke, E. M., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., and Veith, H. State/event software verification for branching-time specifications. In *IFM* (2005), pp. 53–69.

[10] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *CAV* (2002), pp. 241–268.

[11] Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS 8*, 2 (1986), 263.

[12] Clarke, E., Grumberg, O., and Peled, D. *Model checking*. 1999.

[13] Clarke, E., Jha, S., Lu, Y., and Veith, H. Tree-like counterexamples in model checking. In *LICS* (2002), pp. 19–29.

[14] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y. Proving that programs eventually do something good. In *POPL* (2007), pp. 265–276.

[15] Cook, B., and Koskinen, E. Making prophecies with decision predicates. In *POPL* (2011), pp. 399–410.

[16] Cook, B., Koskinen, E., and Vardi, M. Branching-time reasoning for programs. Tech. Rep. UCAM-CL-TR-788, University of Cambridge, Computer Laboratory, Jan. 2011.

[17] Cook, B., Koskinen, E., and Vardi, M. Y. Temporal property verification as a program analysis task. In *CAV* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 333–348.

[18] Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI* (2006), pp. 415–426.

[19] Delzanno, G., and Podelski, A. Model checking in CLP. *TACAS* (1999), 223–239.

[20] Emerson, E., and Namjoshi, K. Automatic verification of parameterized synchronous systems. In *CAV* (1996), pp. 87–98.

[21] Fioravanti, F., Pettorossi, A., Proietti, M., and Senni, V. Program specialization for verifying infinite state systems: An experimental evaluation. In *LOPSTR'10* (2010).

[22] Gastin, P., and Oddoux, D. Fast LTL to Büchi automata translation. In *CAV* (July 2001).

[23] Gurfinkel, A. Personal communication. 2010.

[24] Gurfinkel, A., Wei, O., and Chechik, M. Yasm: A software model-checker for verification and refutation. In *CAV* (2006), pp. 170–174.

[25] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV* (2002), pp. 526–538.

[26] Kupferman, O., Vardi, M., and Wolper, P. An automata-theoretic approach to branching-time model checking. *J. ACM 47*, 2 (2000), 312–360.

[27] Magill, S., Berdine, J., Clarke, E., and Cook, B. Arithmetic strengthening for shape analysis. In *SAS* (2007), vol. 4634, p. 419.

[28] Manna, Z., and Pnueli, A. *Temporal verification of reactive systems: safety*, vol. 2. Springer Verlag, 1995.

[29] McMillan, K. Lazy abstraction with interpolants. In *CAV* (2006), pp. 123–136.

[30] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *Computer Science Logic* (2001), pp. 1–19.

[31] Podelski, A., and Rybalchenko, A. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI* (2004), pp. 239–251.

[32] Podelski, A., and Rybalchenko, A. Transition invariants. In *LICS* (2004), pp. 32–41.

[33] Podelski, A., and Rybalchenko, A. Transition predicate abstraction and fair termination. In *POPL* (2005).

[34] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL* (1995), pp. 49–61.

[35] Schmidt, D., and Steffen, B. Program analysis as model checking of abstract interpretations. *Static Analysis* (1998), 351–380.

[36] Stirling, C. Games and modal mu-calculus. In *TACAS* (1996), pp. 298–312.

[37] Vardi, M. Y. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop* (1995), pp. 238–266.

[38] Walukiewicz, I. Pushdown processes: Games and model checking. In *CAV* (1996), pp. 62–74.

[39] Walukiewicz, I. Model checking CTL properties of pushdown systems. *FST TCS* (2000), 127–138.