Temporal Logic Model Checking as Automated Theorem Proving

by

Amirhossein Vakili

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Model checking is an automatic technique for the verification of temporal properties of a system. In this technique, a system is represented as a labelled graph and the specification as a temporal logic formula. The core of temporal logic model checking is the reachability problem, which is not expressible in first-order logic (FOL); as a result, model checking of finite/infinite state systems without the use of iteration or abstraction is considered beyond the realm of automated FOL theorem provers. In this thesis, we focus on formulating the temporal logic model checking problem as a FOL theorem proving problem and use automated tools, such as SAT/SMT solvers to directly model check a system without the need for a fixed-point calculation or abstraction.

We present *CTL-Live*: a fragment of computational tree logic whose model checking for (infinite) Kripke structures is reducible to FOL validity checking. CTL-Live includes the CTL connectives that are often used to express liveness properties. We also derive decidability results about CTL-Live model checking by examining decidable subsets of FOL. We evaluate our reduction technique for CTL-Live model checking. Our case studies show that state-of-the-art SMT solvers are capable of verifying CTL-Live properties of infinite systems; moreover, the verification of an infinite state model can sometimes complete more quickly than verifying a finite version of the model. We prove the maximality of CTL-Live: we show that CTL-Live is the largest fragment of CTL whose model checking is reducible to FOL validity checking.

The maximality of CTL-Live implies that model checking safety properties requires a logic more expressive than FOL; as a result, we examine FOL plus transitive closure (FOLTC). We can reduce model checking of a more expressive fragment of CTL, which we call CTL$\backslash$**EG**, to validity checking in FOLTC. CTL$\backslash$**EG** is more expressive than CTL-Live and yet less expressive than CTL. By adding a finiteness restriction, we can reduce model checking of all of CTL with fairness constraints (CTLFC) formulas to validity checking in FOLTC. The finiteness restriction requires that the system under-study must have a finite number of states, but it does not require this number to be known. Reduction of CTLFC to FOLTC allows us to use the Alloy Analyzer for model checking. Our case studies show that the Alloy Analyzer can analyze CTLFC formulas up to the same scopes that Alloy models are analyzed.

Acknowledgements

A PhD is an investment, and in my case, it was a looooooooooong-term investment. From a personal perspective, this investment has been a complete success, and without a doubt, Professor Nancy A. Day played the leading role. Nancy taught me how to research, measure real contribution, write, publish, teach, and be professional. Considering my almost empty set of skills at the beginning of the PhD program makes me doubt her ability at choosing good students :)

I would like to thank Professor Ganesh Gopalakrishnan for taking the time to read my thesis. His feedback gave me a fresh perspective on my work. Professors Krzysztof Czarnecki, Richard Trefler, and Peter van Beek are the best committee members one can ask for. They have been always accessible and their constructive feedback has had a significant impact on my work.

Special thanks to the members of the NECSIS project, in particular, Prof. Joanne Atlee, Prof. Krzysztof Czarnecki, Dr. Michal Antkiewicz, and Dr. Shoham Ben-David. Their constant feedback and support has taught me what teamwork is supposed to be. The backbone of the Cheriton School of Computer Science is its staff members. I would have been hopeless without Wendy Rush, Ronaldo Garcia, Margaret Towell, and Helen Jardine. Helen must be thrilled by my graduation: no more bothering her for leaving my keys inside my office and getting locked out.

I would like to thank my friends, Barbara O'Gorman, Reza, Yasaman, Vahed, Hadi, Pouria, Rana, John, Adam, Vajih, and Amirali. Each one is unique in their own way. Vajih is an outstanding software engineer who I learned a lot from. Katia, Ramin, and Doorsa are my extended family in Canada; their kindness, warmth, generosity, and love cannot be described by words.

My parents, Mahboubeh and Mohammad, are my role models. They made all possible sacrifices and supported me during all the stages of my life. My education would have been impossible without them. The support of my lovely siblings, Panteha, Ali, and Amin has always helped me in different stages of my life, including this thesis.

My amazing in-laws, Fereshteh Tavallari and Mohammad Roohparvarzadeh, created an outstanding environment for me to write my thesis. They took care of my wife's cravings during her pregnancy, and made their home into a thesis camp. Their love and support has no parallel. I cannot thank them enough.

Sogol and I got married during my PhD and that prolonged my "investment." I do not blame her for that, but she is responsible for the extra 10 pounds that I'm carrying around since we've been married. She is an excellent cook/mechanical engineer.

dedicated to Sogoli, the most beautiful mama bear

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Model checking is an automatic technique for the verification of temporal properties of a system [7, 25]. In this technique, a system is formalized as a set of states with transitions between them. Each state represents a configuration of the system, transitions represent how a system moves from one state to another. This representation is called a *Kripke structure*. A temporal property is usually represented as a temporal logic formula. Temporal logics, such as computational tree logic (CTL) [26], are used to represent concisely a temporal property. A model checker takes a Kripke structure and temporal logic formula as input, and it determines if the Kripke structure satisfies the temporal logic formula or not: if it does, the model checker's output is `yes`; otherwise, the output is a counterexample that shows how the system under study violates the property that is formalized as a temporal logic formula.

The automatic nature of model checkers has turned them into an effective (and popular) technique to find bugs in both software and hardware systems. Holzmann used model checking and found 112 major bugs within a real world protocol [39]. The errors revealed by model checking have lead to major improvements in the IEEE Futurebus protocol [27]. Applications of model checking at IBM has shown it to be an effective technique in finding bugs in a memory bus adapter [3, 51]. In these case studies, authors reported that 40% of the bugs found by model checking would not have been found by simulation or testing.

Over the past 20 years, there have been significant advancements in the world of SAT and satisfiability modulo theory (SMT) solving [9]. The input languages of such theorem provers are usually at most as expressive as first-order logic (FOL), and (unlike temporal logics) they do not provide any notion of time. The advancements in FOL theorem proving has motivated researchers to design model checking techniques that use these solvers [14,16, 20, 52]. In fact, according to the Hardware Model Checking Competition 2011 [1], leading model checkers are based on SAT solving rather than traditional BDD-based methods [18] or explicit-state search [36].

Model checking techniques based on automated FOL theorem provers, such as SAT and SMT solvers, can be divided into two major categories: 1) bounded model checking (e.g., [14, 52]) and 2) unbounded model checking (e.g., [17, 53]). Bounded methods check whether a property holds for a certain length of execution path by creating a formula consisting of the transition relation expanded to the desired bound. Since the bound is finite, the problem can be expressed in FOL, therefore, an FOL theorem prover can be used to solve the entire bounded (and therefore incomplete) model checking problem at one time. Unbounded methods call an FOL theorem prover multiple times iteratively to traverse the reachable state space may not guarantee termination. This iteration can result in parts of the reasoning being redone multiple times. FOL theorem provers, such as SMT solvers, have not been used to solve an entire unbounded model checking problem in one call because model checking is a question of reachability within a graph (in this case a Kripke structure), and the reachability relation (transitive closure) is not expressible in FOL. Therefore, temporal logic model checking for infinite state systems without the use of iteration or abstraction is usually considered beyond the realm of FOL theorem provers.

In this thesis, we focus on formulating the unbounded model checking problem as a theorem proving problem for logics that have sophisticated automated provers.

1.1 Thesis Overview

First, we show that for a fragment of computational tree logic (CTL), which we call *CTL-Live*, the model checking problem is reducible to FOL validity checking [56]. CTL-Live includes the CTL connectives that are often used to express liveness properties (e.g., **AF**, **AU**, etc.). Our key insight in this reduction is the use of the implicit higher-order universal quantifier in the definition of the FOL validity relation. This universal quantifier allows us to describe the semantics of CTL-Live formulas.

Based on this reduction, we use SMT solvers for model checking some case studies [58]. Our case studies show that SMT solvers, in particular Z3 [29], are effective in verifying CTL-Live properties of infinite systems.

We also show that CTL-Live is the largest fragment of CTL whose model checking is reducible to FOL validity checking [57]. Using this result, we also show that the inductive invariant method for verification of safety properties is incomplete. Another theoretical and

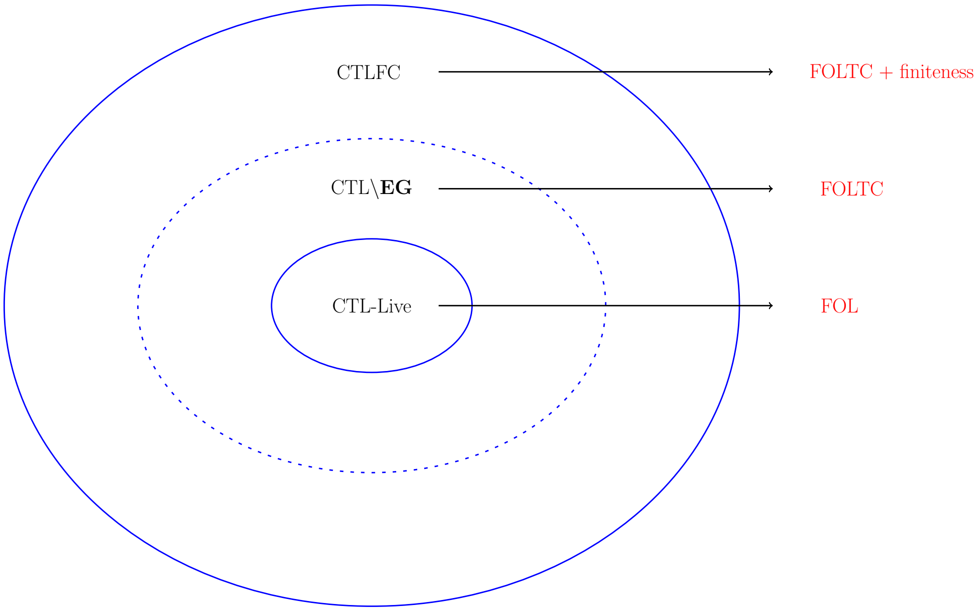


Figure 1.1: Temporal Logics Hierarchy

practical implication of CTL-Live maximality is that in order to reduce model checking to validity checking, one needs to consider more expressive logics than FOL. For this reason, next we focus on FOL plus transitive closure (FOLTC).

We reduce model checking of a more expressive fragment of CTL to validity checking for FOLTC. We call this fragment CTL**EG**. It is more expressive than CTL-Live and yet less expressive than CTL. By adding a finiteness restriction, we can reduce model checking all of CTL with fairness constraints (CTLFC) formulas [25] to FOLTC validity checking [55]. The finiteness restriction requires that the system under study must have a finite number of states, but it does not require this number to be known. There are two major insights in this reduction: 1) transitive closure contains reachability information, 2) given a finite number of states, the only way to have an infinite path is through repetition of states.

Figure 1.1 depicts the hierarchy of temporal logics that we study in this thesis and the logics for expressing their model checking problems. We do not have a maximality result about CTL**EG**. For this reason, dashes are used to represent the boundaries of CTL**EG** in Figure 1.1.

> **Thesis statement:** There is a fragment of CTL whose model checking is reducible to FOL validity checking. Using this reduction, modern SMT solvers

are capable of verifying infinite state systems without the need for abstraction, invariant generation, or iteration. FOLTC, a more expressive logic, can be used to encode more CTL formulas than FOL. Using FOLTC and focusing on finite systems, every CTLFC formula can be encoded in FOLTC. This reduction can be used to express temporal properties in a language like Alloy that is as expressive as FOLTC.

1.2 Contributions

In this thesis we show that despite the fact that reachability is not expressible in FOL, there are reachability queries that can be articulated in FOL by using a very novel approach. We use this approach to automaticly verify some infinite systems. We also show that to articulate more reachability queries, a more expressive logic than FOL is needed. In particular, we focus on FOLTC and present reachability queries that can be formulated in FOLTC.

The following lists the contributions of this thesis:

- Introducing CTL-Live: a fragment of CTL whose model checking problem is reducible to FOL validity checking.
- Using SMT solvers, we have shown the effectiveness of SMT solvers in model checking CTL-Live properties.
- Proving the maximality of CTL-Live: model checking of CTL connectives that are not included in CTL-Live is not reducible to FOL validity checking.
- Showing that the inductive invariant method for verification of safety properties is not complete.
- Deriving decidability results for CTL-Live model checking by examining some decidable fragments of FOL.
- Showing how CTL connectives, except **EG**, can be encoded as FOLTC formula.
- Showing how all of CTLFC formula can be encoded in FOLTC for finite systems.
- Using the Alloy Analyzer, we have shown CTLFC properties can be analyzed up to the same scopes that the Alloy models are analyzed.

Our major philosophical contribution in this thesis is to show that expressibility and verifiability of a property are two distinct things: a property may be verifiable despite being inexpressible.

1.3 Validation

We have validated the practicality of our model checking technique for CTL-Live by using SMT solvers verifying four infinite systems. These results are provided in Chapter 5. Our results show that sophisticated SMT solvers, in particular Z3 [29], are effective in verification of CTL-Live properties [58].

The practicality of our reduction technique of CTLFC to FOLTC has been validated by case studies in Alloy [55]. Alloy is declarative relational language that is expressive as FOLTC. The Alloy Analyzer provides automatic finite scope analysis for Alloy models. The expressive power of Alloy and its automatic analyzer made it a perfect candidate for validation our reduction of CTLFC to FOLTC. Our case studies show that CTLFC formulas can be analyzed by the Alloy Analyzer up to the same scopes that non-temporal properties are analyzed in Alloy.

1.4 Thesis Outline

This thesis is divided into three parts. In Part I, we present the foundations of our work: Chapter 2 presents background material on FOL, temporal logics and model checking. In this chapter, we also discuss transitive closure and its relationship to FOL. Chapter 3 is an overview of how we represent Kripke structures in FOL. We call this representation a *symbolic Kripke structure* (SKS).

In Part II, we examine the use of FOL for model checking. In Chapter 4, we present CTL-Live and how its model checking is reducible to FOL validity checking. This chapter also discusses the maximality of CTL-Live. Chapter 5 presents the practicality of using SMT solvers in solving model checking problems based on the reduction technique presented in Chapter 4.

Part III discusses the use of FOLTC for model checking. Chapter 6 presents our encoding of CTL$\backslash$**EG** and CTLFC in FOLTC. This encoding is evaluated in Chapter 7, by using the Alloy Analyzer as the back-end solver. Chapter 8 concludes this thesis and presents future directions. We discuss related work in each chapter.

Part I

Foundations

Chapter 2

Background

In this chapter, we present the background material and the notation that is used in this thesis. Table 2.1 presents some set and relational operators.

Section 2.1 is an overview on first-order logic (FOL). In Section 2.2, we overview the transitive closure operator and discuss its relationship to FOL. Section 2.3 presents computational tree logic (CTL) and CTL with fairness constraints (CTLFC). In Section 2.3, we overview some classical model checking algorithms as well.

2.1 First-Order Logic

Formulas in FOL are built from logical connectives, functional symbols, and relational symbols [19]. The set of logical connectives and their semantics in FOL is fixed. The following is a standard set of logical connectives for FOL: true ($\top$), false ($\bot$), negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), iff ($\Leftrightarrow$), existential quantifier ($\exists$), and universal quantifier ($\forall$). Since for different problems different sets of functional and relational symbols are used, we have the following definition:

Definition 1. (Base) A base $\mathscr{B}$ is a finite set of **symbols** that can be divided into the three categories of relational symbols, functional symbols and variables:

$$\mathscr{B} = \{R_0, .., R_n, F_0, .., F_m, v_0, .., v_p\}$$

Every relational and function symbol has a corresponding arity representing the number of arguments that is required by that symbol. ■

Table 2.1: Standard set and relational operators, where $\mathcal{S}$, $\mathcal{S}_1$, $\mathcal{S}_2$ are sets and $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$ are binary relations.

Syntax	Name	Meaning
$\mathcal{S}_1 \subseteq \mathcal{S}_2$	Subset	every element of $\mathcal{S}_1$ is an element of $\mathcal{S}_2$
$\mathcal{S}_1 \cap \mathcal{S}_2$	Intersection	$\{x \mid\ x \in \mathcal{S}_1 \text{ and } x \in \mathcal{S}_2\}$
$\mathcal{S}_1 \cup \mathcal{S}_2$	Union	$\{x \mid\ x \in \mathcal{S}_1 \text{ or } x \in \mathcal{S}_2\}$
$\mathcal{S}_1 \setminus \mathcal{S}_2$	Difference	$\{x \mid\ x \in \mathcal{S}_1 \text{ and } x \notin \mathcal{S}_2\}$
$\mathcal{S}_1 \times \mathcal{S}_2$	Cartesian product	$\{(x, y) \mid\ x \in \mathcal{S}_1 \text{ and } y \in \mathcal{S}_2\}$
$\mathcal{R}_1 \cap \mathcal{R}_2$	Intersection	$\{(x, y) \mid\ (x, y) \in \mathcal{R}_1 \text{ and } (x, y) \in \mathcal{R}_2\}$
$\mathcal{R}_1 \cup \mathcal{R}_2$	Union	$\{(x, y) \mid\ (x, y) \in \mathcal{R}_1 \text{ or } (x, y) \in \mathcal{R}_2\}$
$\texttt{id}(\mathcal{S})$	Identity relation	$\{(s, s) \mid\ s \in \mathcal{S}\}$
$\mathcal{S}; \mathcal{R}$	Relational join	$\{y \mid\ \text{for some } x \in \mathcal{S} \bullet\ (x, y) \in \mathcal{R}\}$
$\mathcal{R}; \mathcal{S}$	Relational join	$\{x \mid\ \text{for some } y \in \mathcal{S} \bullet\ (x, y) \in \mathcal{R}\}$
$\mathcal{R}_1; \mathcal{R}_2$	Relational join	$\{(x, z) \mid\ \text{for some } y \bullet\ (x, y) \in \mathcal{R}_1 \text{ and } (y, z) \in \mathcal{R}_2\}$
$\mathcal{S} \lhd \mathcal{R}$	Domain restriction	$\{(x, y) \mid\ x \in \mathcal{S} \text{ and } (x, y) \in \mathcal{R}\}$
$\mathcal{R} \rhd \mathcal{S}$	Range restriction	$\{(x, y) \mid\ y \in \mathcal{S} \text{ and } (x, y) \in \mathcal{R}\}$

A functional (relational) symbol with arity 0 is called a *constant* (*proposition*). A relational symbol X with arity n is denoted by X/n.

Definition 2. (Syntax of FOL) Let $\mathscr{B}$ be a base. The set of formulas over $\mathscr{B}$ is defined as follows:

$$\begin{array}{lll} \Phi & ::= & \top \mid \bot \mid p \qquad\qquad\qquad \text{where } p \in \mathscr{B} \text{ is a proposition,} \\ & ::= & R(t_1, \ldots, t_n) \mid t_1 = t_2 \quad \text{where } R \in \mathscr{B} \text{ is a relational symbol, and } t_i \text{ is a term} \\ & ::= & \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \Rightarrow \Phi_2 \mid \Phi_1 \Leftrightarrow \Phi_2 \\ & ::= & \exists v \bullet \Phi \mid \forall v \bullet \Phi \qquad\quad \text{where } v \in \mathscr{B} \text{ is a variable,} \end{array} \tag{2.1}$$

$$\begin{array}{lll} t & ::= & v \mid c \qquad\qquad \text{where } v \in \mathscr{B} \text{ and } c \in \mathscr{B} \text{ is a variable and a constant respectively,} \\ & ::= & F(t_1, \ldots, t_n) \quad \text{where } F \in \mathscr{B} \text{ is a functional symbol and it is } n\text{-ary.} \end{array} \tag{2.2}$$

■

Equations 2.1 and 2.2 represent the rules for constructing formulas and terms respectively.

The semantics of FOL is defined by using *interpretations*. An interpretation defines the meaning of a base by assigning values to variables, functional and relational symbols. Using these values along with the fixed semantics of FOL logical connectives, a formula is evaluated to *true* or *false*.

Definition 3. (Interpretation) Let $\mathscr{B}$ be a base for FOL. An interpretation of $\mathscr{B}$ is a pair $\mathcal{I} = \langle \mathcal{D}, .^{\mathcal{I}} \rangle$, where $\mathcal{D}$ is a non-empty set, *domain* of $\mathcal{I}$, and $.^{\mathcal{I}}$ is a mapping that assigns:

1. to every variable $v \in \mathscr{B}$ an element in $\mathcal{D}$, $v^{\mathcal{I}} \in \mathcal{D}$,
2. to every 0-ary functional symbol $c \in \mathscr{B}$ an element in $\mathcal{D}$, $c^{\mathcal{I}} \in \mathcal{D}$,
3. to every functional symbol $f \in \mathscr{B}$ of arity $n \geq 1$ a total function from $\mathcal{D}^n$ to $\mathcal{D}$, $f^{\mathcal{I}} : \mathcal{D}^n \to \mathcal{D}$,
4. to every 0-ary relational symbol $p \in \mathscr{B}$ either $true$ or $false$, $p^{\mathcal{I}} = true$ or $p^{\mathcal{I}} = false$,
5. to every relational symbol $r \in \mathscr{B}$ of arity $n \geq 1$ a relation over $\mathcal{D}^n$, $r^{\mathcal{I}} \subseteq \mathcal{D}^n$.

■

Relations come from the world of semantics and relational symbols are from the world of syntax. A relational symbol can be considered as a variable having different values under different interpretations. Interpretations assign relations to relational symbols.

Before defining the semantics of FOL, we need to introduce a notation that is used to define the semantics of quantifiers:

Definition 4. (Substitution for interpretations) Let $\mathscr{B}$ be a base for FOL and $\mathcal{I} = \langle \mathcal{D}, .^{\mathcal{I}} \rangle$ an interpretation for $\mathscr{B}$. For every $d \in \mathcal{D}$ and variable $v \in \mathscr{B}$, $\mathcal{I}^{v:=d}$ is an interpretation over $\mathscr{B}$ that is same as $\mathcal{I}$ except it maps v to d:

$$x^{\mathcal{I}^{v:=d}} = \begin{cases} d & \text{if } x = v, \\ x^{\mathcal{I}} & \text{otherwise.} \end{cases}$$

■

Definition 5. (Satisfiability of an FOL formula) Let $\mathscr{B}$ be a base for FOL and $\mathcal{I} = \langle \mathcal{D}, .^{\mathcal{I}} \rangle$ an interpretation for B. The satisfiability relation over formulas and interpretations, $\Vdash$, is defined by using structural induction on the formula Φ and the term t knowing that $\top$ is always satisfied ($\mathcal{I} \Vdash \top$) and $\bot$ is never satisfied ($\mathcal{I} \Vdash \bot$):

$$
\begin{array}{lll}
\mathcal{I} \Vdash p & \text{iff} & p^{\mathcal{I}} = \mathit{true}, \text{ where } p \in R \text{ with arity } 0, \\
\mathcal{I} \Vdash r(t_1, \dots, t_n) & \text{iff} & r^{\mathcal{I}}(t_1^{\mathcal{I}}, \dots, t_n^{\mathcal{I}}) \text{ holds}, \\
\mathcal{I} \Vdash t_1 = t_2 & \text{iff} & t_1^{\mathcal{I}} \text{ is equal to } t_2^{\mathcal{I}}, \\
\mathcal{I} \Vdash \neg\Phi & \text{iff} & \mathcal{I} \text{ does not satisfy } \Phi, \\
\mathcal{I} \Vdash \Phi_1 \wedge \Phi_2 & \text{iff} & \mathcal{I} \Vdash \Phi_1 \text{ and } \mathcal{I} \Vdash \Phi_2, \\
\mathcal{I} \Vdash \Phi_1 \vee \Phi_2 & \text{iff} & \mathcal{I} \Vdash \Phi_1 \text{ or } \mathcal{I} \Vdash \Phi_2, \\
\mathcal{I} \Vdash \Phi_1 \Rightarrow \Phi_2 & \text{iff} & \mathcal{I} \Vdash \neg\Phi_1 \vee \Phi_2, \\
\mathcal{I} \Vdash \Phi_1 \Leftrightarrow \Phi_2 & \text{iff} & \mathcal{I} \Vdash \Phi_1 \Rightarrow \Phi_2 \text{ and } \mathcal{I} \Vdash \Phi_2 \Rightarrow \Phi_1, \\
\mathcal{I} \Vdash \exists x \bullet \Phi & \text{iff} & \text{there exists a } d \in D \text{ such that } \mathcal{I}^{x:=d} \Vdash \Phi. \\
\mathcal{I} \Vdash \forall x \bullet \Phi & \text{iff} & \text{for all } d \in D,\ \mathcal{I}^{x:=d} \Vdash \Phi. \\
 & & \\
\big(f(t_1, \dots, t_n)\big)^{\mathcal{I}} & := & f^{\mathcal{I}}(t_1^{\mathcal{I}}, \dots, t_n^{\mathcal{I}})
\end{array}
$$

The notation $\mathcal{I} \nVdash \Phi$ denotes that $\mathcal{I}$ does not satisfy Φ. ■

For a set of FOL formula Γ, and an interpretation $\mathcal{I}$, we use the notation $\mathcal{I} \Vdash \Gamma$ to denote that $\mathcal{I}$ satisfies every formula in Γ.

Definition 6. (Validity in FOL) Let Γ be a set of FOL formulas. An FOL formula Φ is valid with respect to Γ iff every interpretation $\mathcal{I}$ that satisfies all the formulas in Γ also satisfies Φ. Validity is denoted by $\models$. ■

Validity checking for FOL is recursively enumerable [23, 37]. This means that validity checking for FOL is not computable, but there is a procedure that given Γ and Φ produces a proof in the case $\Gamma \models \Phi$ holds.

Definition 7. (Consistency) A set of FOL formulas Γ is consistent iff there is an interpretation that satisfies all the formulas in Γ. ■

Next, we present two theorems about satisfiability and validity in FOL. The proofs of these theorems can be found in [19].

Theorem 1. Let Γ be a set of FOL formulas and Φ an FOL formula. We have:

$$\Gamma \models \Phi \quad \text{iff} \quad \Gamma \cup \{\neg\Phi\} \text{ is not consistent.}$$

Theorem 2. (Compactness) A set of FOL formulas Γ is consistent iff every finite subset of Γ is consistent.

2.2 Transitive Closure and FOL

This section is an overview of the *transitive closure* operator and how it is related to FOL.

Definition 8. (Transitive closure) Let $\mathcal{D}$ be a set and $\mathcal{R}$ a binary relation. The transitive closure of $\mathcal{R}$, denoted by $\mathcal{R}^+$, is the smallest binary relation that contains $\mathcal{R}$ and it is transitive:

$$\mathcal{R}^+ \text{ includes } \mathcal{R} \quad : \quad \mathcal{R} \subseteq \mathcal{R}^+ \tag{2.3}$$

$$\mathcal{R}^+ \text{ is transitive} \quad : \quad \mathcal{R}^+ ; \mathcal{R}^+ \subseteq \mathcal{R}^+ \tag{2.4}$$

$$\mathcal{R}^+ \text{ is the smallest} \quad : \quad \forall \mathcal{RT} \bullet \quad \mathcal{R} \subseteq \mathcal{RT} \wedge \mathcal{RT} ; \mathcal{RT} \subseteq \mathcal{RT} \quad \Rightarrow \quad \mathcal{R}^+ \subseteq \mathcal{RT} \tag{2.5}$$

■

Extending FOL with the transitive closure operator results in a new logic called *FOL with transitive closure* (FOLTC). A base $\mathscr{B}$ for FOL is also used as a base for FOLTC. The syntax of FOLTC is same as FOL with one addition: for every binary relational symbol $R/2$ in $\mathscr{B}$, R^+ is also a binary relational symbol. The notation $^+$ is overloaded: it is applied to binary relations (semantics) and relational symbols with arity 2 (syntax). For every interpretation $\mathcal{I}$, base $\mathscr{B}$, and binary relational symbol $R \in \mathscr{B}$, the value of R^+ under $\mathcal{I}$ is defined as follows:

$$(R^+)^{\mathcal{I}} := (R^{\mathcal{I}})^+$$

Satisfiability, validity, and consistency for FOLTC are the same as for FOL.

The major difference between FOL and FOLTC is that FOLTC does not have the compactness property of Theorem 2; as a result, FOLTC is essentially more expressive than FOL. The intuition behind this is that the universal quantifier in Equation 2.5 is higher-order and the syntax of FOL does not allow such quantification.

2.3 Temporal Logics and Model Checking

Temporal logics refers to a family of formal logics that are used to express dynamic properties of a system concisely: properties that describe some behaviour of a system over time. In this section, we present *computational tree logic* (CTL) and CTL with fairness constraints (CTLFC) [25, 26].

The semantics of temporal logics are defined using *Kripke structures*. A Kripke structure is a directed labelled graph. The set of vertices and the set of edges of a Kripke structure are often called the *state space* and the *transition relation* respectively. The labels of each state show the local properties of the state: what holds and what does not hold in a state. In practice, different combinations of the values of the variables that are used to define a system represent the state space.

Definition 9. (Kripke structure) A Kripke structure is a four tuple $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$, where: $\mathcal{S}$ is a set of states; $\mathcal{S}_0$, the set of initial states, is a non-empty subset of $\mathcal{S}$; $\mathcal{N}$, the next-state relation is a binary relation over $\mathcal{S}$; $\mathbb{P}$ is a finite set of unary predicates over states. Predicates represent the local properties of the states, and are called *labelling predicates.* ■

A Kripke structure can be considered as a structure that defines a set of infinite paths:

Definition 10. (Infinite computation path) An infinite computation path, for short computation path, in a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ starting at state $s \in \mathcal{S}$ is a total function δ from natural numbers $\mathbb{N}$ to $\mathcal{S}$ such that $\delta(0) = s$ and it satisfies the following property:

$$\forall i \in \mathbb{N} \bullet \ \mathcal{N}\big(\delta(i), \delta(i+1)\big)$$

■

2.3.1 CTL

CTL contains a set of propositional and temporal connectives. The syntax of CTL is presented in Definition 11.

Definition 11. (Syntax of CTL) The set of CTL formulas over a set of labelling predicates $\mathbb{P}$ is defined by the following grammar:

$$\begin{aligned} \varphi \ &::= \ \top \mid \mathcal{P} \mid \neg\varphi, \text{ where } \mathcal{P} \in \mathbb{P} \\ &::= \ \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \\ &::= \ \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{AG}\varphi \\ &::= \ \varphi_1\mathbf{EU}\varphi_2 \mid \varphi_1\mathbf{AU}\varphi_2 \end{aligned} \tag{2.6}$$

■

CTL is a branching-time temporal logic. A temporal connective of CTL consists of two parts: a path and a state quantifier. The path quantifier is either **E**, there exists a path, or **A**, for all paths. The state quantifiers are **X** (next state), **F** (eventually), **G** (globally), and **U** (strong until). The satisfiability relation for CTL, $\Vdash_c$, is used to give meaning to CTL formulae. The notation $\mathcal{K}, s \Vdash_c \varphi$ denotes that the state s of the Kripke structure $\mathcal{K}$ satisfies the CTL formula φ and $\mathcal{K}, s \nVdash_c \varphi$ is used when $\mathcal{K}, s \Vdash_c \varphi$ does not hold.

Definition 12. Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ be a Kripke structure and φ a CTL formula. The satisfiability relation for CTL, $\Vdash_c$, is defined by structural induction on φ knowing that $\mathcal{K}, s \Vdash_c \top$ always holds:

$\mathcal{K}, s \Vdash_c \mathcal{P}$ iff $s \in \mathcal{P}$ for $\mathcal{P} \in \mathbb{P}$
$\mathcal{K}, s \Vdash_c \neg\varphi$ iff $\mathcal{K}, s \nVdash_c \varphi$
$\mathcal{K}, s \Vdash_c \varphi_1 \wedge \varphi_2$ iff $\mathcal{K}, s \Vdash_c \varphi_1$ and $\mathcal{K}, s \Vdash_c \varphi_2$
$\mathcal{K}, s \Vdash_c \varphi_1 \vee \varphi_2$ iff $\mathcal{K}, s \Vdash_c \varphi_1$ or $\mathcal{K}, s \Vdash_c \varphi_2$
$\mathcal{K}, s \Vdash_c \mathbf{EX}\varphi$ iff $\exists s' \in \mathcal{S} \bullet\ \mathcal{N}(s, s') \wedge \mathcal{K}, s' \Vdash_c \varphi$
$\mathcal{K}, s \Vdash_c \mathbf{AX}\varphi$ iff $\forall s' \in \mathcal{S} \bullet\ \mathcal{N}(s, s') \Rightarrow \mathcal{K}, s' \Vdash_c \varphi$
$\mathcal{K}, s \Vdash_c \mathbf{EF}\varphi$ iff there exists a computation path δ starting at s, and an i such that $\mathcal{K}, \delta(i) \Vdash_c \varphi$.
$\mathcal{K}, s \Vdash_c \mathbf{AF}\varphi$ iff for all computation paths δ's starting at s, there exists an i such that $\mathcal{K}, \delta(i) \Vdash_c \varphi$.
$\mathcal{K}, s \Vdash_c \mathbf{EG}\varphi$ iff there exists a computation path δ starting at s such that for all i's $\mathcal{K}, \delta(i) \Vdash_c \varphi$.
$\mathcal{K}, s \Vdash_c \mathbf{AG}\varphi$ iff for all computation paths δ's starting at s, and i's $\mathcal{K}, \delta(i) \Vdash_c \varphi$.
$\mathcal{K}, s \Vdash_c \varphi_1\mathbf{EU}\varphi_2$ iff there exists a j and a computation path δ starting at s such that $\mathcal{K}, \delta(j) \Vdash_c \varphi_2$ and for all $i < j$ $\mathcal{K}, \delta(i) \Vdash_c \varphi_1$.
$\mathcal{K}, s \Vdash_c \varphi_1\mathbf{AU}\varphi_2$ iff for all computation paths δ's starting at s, there exists a j such that $\mathcal{K}, \delta(j) \Vdash_c \varphi_2$ and for all $i < j$ $\mathcal{K}, \delta(i) \Vdash_c \varphi_1$.

For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ and a CTL formula φ, we define the notation $[\varphi]_\mathcal{K}$ to be the subset of $\mathcal{S}$ that satisfy φ:

$$[\varphi]_\mathcal{K} := \{ s \in \mathcal{S} \mid \ \mathcal{K}, s \Vdash_c \varphi \}$$

Definition 13. (Semantics of CTL) Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ be a Kripke structure and φ a CTL formula. The Kripke structure $\mathcal{K}$ satisfies φ iff all of its initial states satisfy φ:

$$\mathcal{K} \Vdash_c \varphi \quad \text{if and only if} \quad \mathcal{S}_0 \subseteq [\varphi]_\mathcal{K}$$

■

Definition 14. (`rank(`s`)`) Let s be a state in Kripke structure $\mathcal{K}$ such that $\mathcal{K}, s \Vdash_c \varphi_1$ **EU** φ_2. We define `rank(`s`)` to be the least number of steps required to reach a state that satisfies φ_2 from s:

$$\texttt{rank}(s) = \begin{cases} 0 & \text{if } \mathcal{K}, s \Vdash_c \varphi_2 \\ 1 + \mathit{min}\{\texttt{rank}(s') \mid \mathcal{N}(s, s') \text{ and } \mathcal{K}, s' \Vdash_c \varphi_1 \textbf{ EU } \varphi_2\} & \text{otherwise.} \end{cases}$$

`rank(`s`)` is defined similarly when $\mathcal{K}, s \Vdash_c \varphi_1$ **AU** φ_2. ■

2.3.2 CTLFC

CTL with fairness constraints extends CTL with the concept of *fair paths*. Intuitively, a fair path is a computation path that satisfies some properties *infinitely often*.

Definition 15. (Fair path) Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ be a Kripke structure and $C = \{\varphi_1, ..., \varphi_n\}$ a set of CTL formulas over $\mathbb{P}$. A computation path δ of $\mathcal{K}$ is fair iff each $\varphi \in C$ is satisfied an infinite number of times along δ:

$$\forall \varphi \in C \bullet \text{ the set } \{i \mid \mathcal{K}, \delta(i) \Vdash_c \varphi\} \text{ is infinite.}$$

■

The syntax of CTLFC is same as CTL with the addition of a new temporal connective $\mathbf{E_C G}$, where the index $\mathbf{C}$ indicates that we are only interested in fair paths:

Definition 16. (Semantics of $\mathbf{E_C G}$) Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ be a Kripke structure, $C = \{\varphi_1, ..., \varphi_n\}$ a set of fairness constraints, and φ a CTLFC formula. The semantics of $\mathbf{E_C G}$ is defined as follows:

$\mathcal{K}, s \Vdash_c \mathbf{E_C G}\varphi$ iff there exists a *fair* path δ with respect to C that starts from s and for all i's $\mathcal{K}, \delta(i) \Vdash_c \varphi$.

■

CTLFC has other temporal connectives that can be defined in terms of the ones that we have presented so far. Figure 2.3.2 on the facing page presents all CTLFC connectives and their relationship to each other and to CTL connectives.

Given a Kripke structure $\mathcal{K}$ where its set of states is finite, verifying if $\mathcal{K}$ satisfies a CTL (CTLFC) formula φ ($\mathcal{K} \Vdash_c \varphi$) is decidable [26]. The decision procedure for verifying

$$\begin{array}{lcl}
\mathbf{E_C X}\varphi & := & \mathbf{EX}\big(\varphi \wedge (\mathbf{E_C G}\top)\big) \\
\mathbf{A_C X}\varphi & := & \neg\mathbf{E_C X}\neg\varphi \\
\mathbf{E_C F}\varphi & := & \mathbf{EF}\big(\varphi \wedge (\mathbf{E_C G}\top)\big) \\
\mathbf{A_C F}\varphi & := & \neg\mathbf{E_C G}\neg\varphi \\
\mathbf{A_C G}\varphi & := & \neg\mathbf{E_C F}\neg\varphi \\
\varphi_1\mathbf{E_C U}\varphi_2 & := & \varphi_1\mathbf{EU}\big(\varphi_2 \wedge (\mathbf{E_C G}\top)\big) \\
\varphi_1\mathbf{A_C U}\varphi_2 & := & \neg(\mathbf{E_C G}\neg\varphi_2) \ \wedge \ \neg\big(\neg\varphi_2\mathbf{E_C U}(\neg\varphi_1 \wedge \neg\varphi_2)\big)
\end{array}$$

Figure 2.1: Relation between CTLFC and CTL connectives

```
Set computeEF(Set $[\varphi]_\mathcal{K}$){
    Set result := $[\varphi]_\mathcal{K}$
    do {
        temp := result
        result := result ∪ $(\mathcal{N};$ result$)$
    } while (result != temp)
    return result
}
```

Figure 2.2: Computing $[\mathbf{EF}\ \varphi]_\mathcal{K}$ from $[\varphi]_\mathcal{K}$

$\mathcal{K} \Vdash_c \varphi$ recursively goes through the structure of φ. For every sub-formula ψ of φ, the procedure determines the set of state of $\mathcal{K}$ that satisfies ψ ($[\psi]_\mathcal{K}$). Eventually, the procedure computes $[\varphi]_\mathcal{K}$ and it checks if all the initial states belong to this set:

$$\mathcal{S}_0 \subseteq [\varphi]_\mathcal{K}$$

For example, Figure 2.2 presents a procedure that computes the set of states that satisfy $\mathbf{EF}\ \varphi$, $[\mathbf{EF}\ \varphi]_\mathcal{K}$, from the set of $[\varphi]_\mathcal{K}$. This procedure computes a fixed-point.

For different connectives of CTL (CTLFC) procedures similar to the one in Figure 2.2 exists.

2.3.3 Reducing Many Fairness Constraints to One

In CTLFC, a Kripke structure may have any finite number of fairness constraints. In this section, we show how a Kripke structure with n fairness constraint(s) can be transformed into a Kripke structure with a single fairness constraint. The transformation we present here was originally presented by Wolper and Vardi to transform a general Büchi automata into a regular one [59, 61].

Given a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ and a set of n fairness constraints $\{\psi_1, \psi_2, .., \psi_n\}$, we construct a Kripke structure $\mathcal{L}$ with a single fairness constraint $\mathcal{FC}$ that behaves the same way as $\mathcal{K}$. The major idea behind this construction is that in every fair path of $\mathcal{K}$, every fairness constraint is satisfied an infinite number of times; therefore, if at some point along a fair path ψ_i is satisfied, at some point after this point ψ_{i+1}[1] is also satisfied.

Lemma 1. Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ be a Kripke structure, $\{\psi_1, \psi_2, .., \psi_n\}$ a set of fairness constraints, and δ a fair path in $\mathcal{K}$ that starts from some state. If $\mathcal{K}, \delta(j) \Vdash_c \psi_i$, then for some $k > j$, we have $\mathcal{K}, \delta(k) \Vdash_c \psi_{i+1}$.

Proof. Proof by contradiction: suppose $\mathcal{K}, \delta(k) \nVdash_c \psi_{i+1}$ for all $k > j$. This means that ψ_{i+1} is only satisfied a finite number of times and this is in contradiction with the fact that δ is a fair path. □

Using Lemma 1, we introduce a "counter" that is incremented whenever a fairness property is satisfied in order. Once all fairness properties are satisfied, we reset the counter to zero. The counter being equal to n becomes the new fairness constraint. The Kripke structure $\mathcal{L}$ is constructed as follows:

$$\mathcal{L} = \langle \mathcal{S} \times \{0, 1, .., n\}, \mathcal{S}_0 \times \{0\}, \mathcal{M}, \mathbb{P} \rangle$$

where $\mathcal{M}$ satisfies the following axiom:

$$\begin{aligned} \forall s, t, i, j \bullet\ \mathcal{M}\big((s,i),(t,j)\big) \ \Leftrightarrow\ & \mathcal{N}(s,t) \quad \wedge & (2.7) \\ & \bigwedge_{l=1}^{n} (i = l - 1\ \wedge\ \psi_l(s) \Rightarrow j = l) \quad \wedge & (2.8) \\ & (i = n \Rightarrow j = 0) \quad \wedge & (2.9) \\ & \bigwedge_{l=1}^{n} (i = l - 1\ \wedge\ \neg\psi_l(s) \Rightarrow j = i) & (2.10) \end{aligned}$$

The above axiom states that in the new Kripke structure $\mathcal{L}$, there is a transition from state (s, i) to state (t, j) iff there is a transition from s to t in $\mathcal{K}$ (Equation 2.7); moreover,

[1] If i is n, we define $i + 1$ to be equal to 1.

if the value of counter in current state, i, is $l-1$ and the lth fairness constraint is also satisfied, the value of counter is incremented by a unit (Equation 2.8 on the preceding page). If the counter reaches n, then it is reset to zero in the next state (Equation 2.9 on the facing page); otherwise, the value of the counter is remained unchanged (Equation 2.10 on the preceding page).

This new Kripke structure $\mathcal{L}$ has a single fairness constraint $\mathcal{FC}$ that satisfies the following axiom:

$$\forall s, i \bullet \ \mathcal{FC}\big((s,i)\big) \quad \Leftrightarrow \quad i = n$$

To relate the behaviour of $\mathcal{K}$ to $\mathcal{L}$, we define a function that converts a path in $\mathcal{L}$ into a path of $\mathcal{K}$:

Definition 17. (Converting paths) Let $\mathcal{L}$ be a Kripke structure derived by the transformation described in this section from a Kripke structure $\mathcal{K}$, and δ a path in $\mathcal{L}$. The operator `remove_counter` converts δ into a path in $\mathcal{K}$ such that:

$$\forall i \in \mathbb{N} \bullet \ \text{remove\_counter}\big(\delta\big)(i) \ \ \textbf{:=} \ \ \texttt{Let} \ (s,l) \ := \ \delta(i) \ \texttt{in} \ s$$

■

The operator `remove_counter` only removes the counter part of each state in $\mathcal{L}$.

Theorem 3. Let $\mathcal{L}$ be a Kripke structure derived by the transformation described in this section from a Kripke structure $\mathcal{K}$. If δ is a fair path in $\mathcal{L}$, then the path `remove_counter`(δ) is a fair path in $\mathcal{K}$; moreover, for every fair path δ' in $\mathcal{K}$ there is fair path δ in $\mathcal{L}$ such that `remove_counter`$(\delta) = \delta'$.

Proof. Proof can be found in [59, 61]. □

2.4 Summary

This chapter was an overview of first-order logic (FOL), transitive closure, and temporal logics. We presented the syntax and semantics of FOL. We also discussed the relationship between the transitive closure operator and FOL. The syntax and semantics of CTL and CTLFC were presented.

Chapter 3

Symbolic Kripke Structures

In the classic presentation of model checking, a system is represented as a Kripke structure and the property that needs to be verified as a temporal logic formula. In Chapters 4 and 6, we present model checking techniques that are based on FOL reasoning. In this chapter, we discuss how we represent Kripke structures in FOL. We call this representation a *symbolic Kripke structure*. The symbolic Kripke structure is a rich formalism for representing systems. It supports both finite and infinite systems. An interesting characteristic of this representation is that a single symbolic Kripke structure can represent a *class* of Kripke structures rather than just a single one. In the following sections, we formally present symbolic Kripke structures and discuss their properties in detail.

3.1 Kripke Structures in FOL

According to Definition 9 on page 14, in a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$, $\mathcal{S}_0$ and every element of $\mathbb{P} = \{\mathcal{P}_1, .., \mathcal{P}_n\}$ are subsets of $\mathcal{S}$, and $\mathcal{N}$ is a binary relation over $\mathcal{S}$. Our key insight is that $\mathcal{K}$ can be considered as an interpretation having $\mathcal{S}$ as its domain for the base $\mathscr{B} = \{S_0, N, P_1, .., P_n\}$. In the base $\mathscr{B}$, we have S_0, P_1, .., P_n as relational symbols of arity 1, and N is a binary relational symbol. The interpretation (Kripke structure) $\mathcal{K}$ maps S_0 to $\mathcal{S}_0$, P_1 to $\mathcal{P}_1$, .., P_n to $\mathcal{P}_n$, and N to $\mathcal{N}$. This insight leads to the following definition:

Definition 18. (Kripke base) A base $\mathscr{B}$ is a Kripke base iff it contains the relational symbols $S_0/1$, $N/2$, and for some positive natural number n, $P_1/1$, .., $P_n/1$ (and possibly more symbols). ■

A Kripke base can be considered as a variable whose values are Kripke structures. The values of a Kripke base is determined by interpretations. This is very similar to the relationship between relational symbols and relations.

Obviously not every base is a Kripke base. For example, the base $\mathscr{B} = \{p, q\}$ where p and q are propositional symbols is not a Kripke base.

Using the concept of a Kripke base in Definition 18 on the previous page, we define the concept of a *symbolic Kripke structure*:

Definition 19. (Symbolic Kripke Structure) A set of FOL formulas Σ over a Kripke base $\mathscr{B}$ is a symbolic Kripke structure. ■

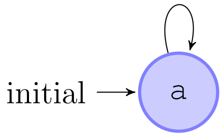
Symbolic Kripke structures (SKSs) can be considered as an axiomatic approach to representing Kripke structures. Every satisfying interpretation of an SKS Σ is a Kripke structure; as a result, an SKS may represent a set of Kripke structures. We define the notation $\mathcal{C}(\Sigma)$ to be the set of Kripke structures represented by Σ:

$$\mathcal{C}(\Sigma) = \{\mathcal{K} \mid \ \mathcal{K} \Vdash \Sigma\}$$

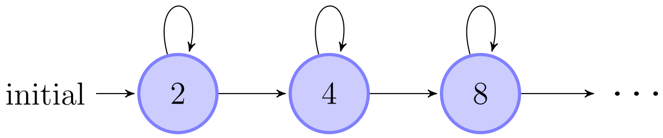
Example 1. Let $\mathscr{B} = \{S_0/1,\ N/2,\ P/1\} \cup \{f,\ a\} \cup \{s,\ s',\ t\}$ be a Kripke base, where f is a binary functional symbol and a is a constant. In this base s, s', and t are variables.

$$\begin{aligned} \Sigma \ = \ \{ \quad & \forall s \bullet \ \ S_0(s) \Leftrightarrow s = f(a,a), \\ & \forall s, s' \bullet \ \ N(s,s') \Leftrightarrow s' = s \vee s' = f(s,s), \\ & \forall s \bullet \ \ P(s) \Leftrightarrow \big(\exists t \bullet s = f(t,t)\big) \ \} \end{aligned}$$

This SKS has three constraints: the first one states that the state s is an initial state if it is $f(a,a)$. The second constraint states that the system moves from state s to s' if they are the same or s' is equal to applying f to (s,s). The third constraint is used to define a single labelling predicate P. This SKS does not uniquely define a Kripke structure. An interpretation $\mathcal{K}_1$ whose domain is $\{\mathtt{a}\}$, and maps a to $\mathtt{a}$, and defines f to be $f^{\mathcal{K}_1}(\mathtt{a},\mathtt{a}) = \mathtt{a}$ denotes a Kripke structure with a single state $\mathtt{a}$. In this Kripke structure, we have $P^{\mathcal{K}_1} = \{\mathtt{a}\}$:



Moreover, an interpretation $\mathcal{K}_2$ whose domain is the set of natural numbers, and maps a to 2, and maps f to the addition function on natural number represents a Kripke structure with an infinite state space:



In $\mathcal{K}_2$, the labelling predicate P is true for even numbers.

In Example 1 on the facing page, the SKS Σ does not uniquely define a Kripke structure because the functional symbol f is uninterpreted. The structure of a $\mathcal{K} \in \mathcal{C}(\Sigma)$ is determined by the values that $\mathcal{K}$ maps to the functional symbols f, and a.

In practice, the following are some reasons for Σ to represent a set of Kripke structures that are not isomorphic to each other:

1. **Under-specification:** the formulae in Σ represent a user's expectations of the underlying system. If a user does not have enough information about the system, then the constraints in Σ do not sufficiently define a single Kripke structure. This is desirable during the early stages of development when a user would like to "explore" different possibilities by under-specification.

2. **Uninterpreted or semi-interpreted symbols:** if there are not enough formulae in Σ to uniquely identify the behaviour of relational and functional symbols used in Σ, then Σ can have satisfying interpretations that assign different values to such symbols.

3. **Theoretical boundaries:** if the description of a system requires the use of operators that are not expressible in FOL, e.g., transitive closure, then a symbolic Kripke structure that represents such a system may not have a unique satisfying interpretation.

3.2 Model Checking of Symbolic Kripke Structures

Model checking a Kripke structure $\mathcal{K}$ against a CTL (CTLFC) formula φ means to determine if $\mathcal{K} \Vdash_c \varphi$ holds. A symbolic Kripke structure Σ may represent more than one Kripke structure. What does it mean to model check a symbolic Kripke structure?

Table 3.1: Summary of the satisfiability and validity notations

Symbol	Meaning	Definition
$\mathcal{I} \quad \Vdash \quad \Phi$	FOL satisfiability	Definition 5 on page 11
$\Gamma \quad \vDash \quad \Phi$	FOL validity	Definition 6 on page 12
$\mathcal{K} \quad \Vdash_c \quad \varphi$	Model checking a single Kripke structure	Definition 13 on page 15
$\Sigma \quad \vDash_c \quad \varphi$	Model checking a symbolic Kripke structure	Definition 20

We define model checking of a symbolic Kripke structure Σ against a CTL (CTLFC) formula φ to determine if *all* satisfying interpretations of Σ satisfy φ:

Definition 20. (Model checking symbolic Kripke structures) Let Σ be a symbolic Kripke structure and φ a temporal property, such as a CTL (CTLFC) formula. Σ satisfies φ, denoted by $\Sigma \vDash_c \varphi$ iff all its satisfying interpretations satisfy φ:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \forall \mathcal{K} \bullet \ \mathcal{K} \Vdash \Sigma \ \Rightarrow \ \mathcal{K} \Vdash_c \varphi$$

or equivalently:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \forall \mathcal{K} \in \mathcal{C}(\Sigma) \bullet \ \mathcal{K} \Vdash_c \varphi$$

The symbolic Kripke structure Σ does not satisfy the CTL (CTLFC) formula φ, denoted by $\Sigma \nvDash_c \varphi$ iff there exists a Kripke structure that satisfies all the formulae in Σ but does not satisfy φ. ■

The definition of $\vDash_c$ is analogous to the definition of validity for FOL (Definition 6 on page 12). Table 3.1 summarizes the satisfiability notations used in this thesis.

3.3 Related Work

State-of-the-art model checkers, such as Spin [40], NuSMV [24], and its successor nuXmv [20], provide languages to represent a single Kripke structure. A user declares a set of variables $v_1 : D_1$, .., $v_n : D_n$ such that the D_i's are finite sets. The state space of such a Kripke structure is $D_1 \times .. \times D_n$, and the transition relation is defined by providing a description of how variables change values based on the current state. Model checkers such as ProB allow a user to use D_i's that represent a set of relations and functions [47]. Such D_i's are still finite.

There are two differences between our representation, symbolic Kripke structures, and the input language of Spin, NuSMV and ProB: 1) a symbolic Kripke structure is capable of representing infinite systems, 2) a symbolic Kripke structure can represent a class of systems rather than a single one.

3.4 Summary

In this chapter, we presented the concept of symbolic Kripke structures as a means of representing Kripke structures in FOL. A symbolic Kripke structure Σ can represent a set of Kripke structures and we defined model checking Σ against a CTL (CTLFC) formula φ as a problem of determining if all the Kripke structures represented by Σ satisfy φ. In the following two chapters, we present methods for using FOL reasoners for model checking symbolic Kripke structures.

Part II

Model Checking in FOL

Chapter 4

Model Checking in FOL: Theory

The semantics of CTL and CTLFC are defined by quantification over paths (Definitions 12 on page 15 and 13 on page 15). Paths are functions from natural numbers to states. Quantification over functions is not possible in FOL; as a result using constraint-based first-order solvers for model checking has remained elusive. In the mean time, the progress in the development of SMT solvers [9] has turned first-order reasoners into powerful, efficient verification tools. In this chapter, we examine the challenge of using FOL to express the model checking problem for an SKS.

Our first contribution in this chapter is to show that model checking an interesting fragment of computational tree logic (CTL), which we call *CTL-Live*, is reducible to validity checking in FOL. To be precise, we show how to turn $\Sigma \models_c \varphi$ into validity checking of $\Gamma \models \Phi$ for some set of FOL formulas Γ and an FOL formula Φ. This reduction implies that model checking a CTL-Live property of an SKS be done completely using deductive techniques of FOL. CTL-Live includes the CTL connectives that are often used to express liveness properties (e.g., **AF**, **AU**, etc.).

Our second contribution is to show the *maximality of CTL-Live*: we prove that CTL-Live is the largest fragment of CTL whose model checking problem is reducible to FOL validity checking.

Finally, we address the verification of safety properties (which are not part of CTL-Live). We show that the inductive invariant approach to verifying invariants is not complete for infinite state systems: an inductive invariant for a safety property does not always exist.

Reduction of CTL-Live model checking to FOL validity checking does not imply any decidability results. Our last contribution in this chapter is to derive decidability results about model checking CTL-Live properties by considering fragments of FOL whose validity checking is decidable.

Most of the content of this chapter has been published [56] and is present in a technical report [57]. This chapter is organized as follows: in Section 4.1 we present the intuition behind our reduction by first focusing on transitive closure and some queries about it that can be formulated as validity problems in FOL despite the fact that transitive closure is not expressible in it. CTL-Live is presented in Sections 4.2. In this section, we also show how CTL-Live model checking is reducible to FOL validity checking. In Section 4.3, we prove the maximality of CTL-Live. In Section 4.4 on page 44, we show that the inductive invariant method for verification of safety properties is not complete. Decidability results are presented in Section 4.5.

4.1 Transitive Closure in FOL

Expressing a property in FOL means to come up with an FOL formula that is only satisfied by those interpretations that have the property. For example, the property "R is a transitive relation" is expressible in FOL by the formula $\forall x, y, z \bullet R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$. For every interpretation $\mathcal{I}$, we have:

$$R^{\mathcal{I}} \text{ is a transitive relation} \quad \text{iff} \quad \mathcal{I} \Vdash \forall x, y, z \bullet R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$$

We know that not every property is expressible in FOL. For example, the property "b is reachable from a via R in a finite number of steps" or equivalently "$R^+(a, b)$" is not expressible in FOL: it is not possible to come up with an FOL formula that is satisfied only by those interpretations that make b reachable from a via R [41].

We are interested in *verification*: given a set of FOL formulas Δ and some property, do all satisfying interpretations of Δ have the property of interest? Obviously, if we can express a property as a FOL formula Ψ, verifying whether Δ has the property of the interest is equivalent to the following validity in FOL:

$$\Delta \models \Psi$$

Despite the fact that $R^+(a, b)$ is not expressible in FOL, we show that verifying whether Δ has the property $R^+(a, b)$ is doable in FOL; in other words, there is a set of FOL formula Γ, and an FOL formula Φ that have the following property:

$$\Delta \models R^+(a, b) \quad \text{iff} \quad \Gamma \models \Phi \tag{4.1}$$

The above equivalence states that verifying whether b is reachable from a via R under all satisfying interpretations of Δ can be formulated as a validity problem in FOL. This enables us to use an FOL theorem prover to verify $\Delta \vDash R^+(a,b)$. Now, we present the Γ and Φ that satisfy Equation 4.1 on the facing page.

According to Definition 8 on page 13, the transitive closure of R, is an RT that has the following three properties:

1. RT includes R: $\forall x, y \bullet\ R(x,y) \Rightarrow RT(x,y)$
2. RT is transitive: $\forall x, y, z \bullet RT(x,y) \wedge RT(y,z) \Rightarrow RT(x,z)$
3. RT is the subset of any relation that satisfies Properties 1 and 2.

We add Properties 1 and 2 to Δ and create Γ:

$$\Gamma \quad := \quad \Delta \cup \{\forall x, y \bullet\ R(x,y) \Rightarrow RT(x,y),\ \forall x, y, z \bullet RT(x,y) \wedge RT(y,z) \Rightarrow RT(x,z)\}$$

Without loss of generality, we assume RT is a binary relational symbol that does not appear in any formulas of Δ. Since Properties 1 and 2 do not uniquely characterize R^+, an interpretation $\mathcal{I}$ that satisfies Δ can be extended to an interpretation $\mathcal{I}'$ that satisfies Γ in different ways. Because of Properties 1 and 2, all those $\mathcal{I}'$ satisfy the following property:

$$R^{\mathcal{I}} = R^{\mathcal{I}'} \quad \text{and} \quad (R^{\mathcal{I}})^+ \subseteq RT^{\mathcal{I}'}$$

The reason RT under each $\mathcal{I}'$ includes the transitive closure of R under $\mathcal{I}$ is that Γ does not include Property 3: the smallest relation that includes R and is transitive. Since the smallest property is not included in Γ, all we can expect from RT is that it includes the transitive closure of R. This is depicted in Figure 4.1 on the following page.

According to the definition of transitive closure, the pair (a,b) belongs to R^+ iff (a,b) belongs to **every** RT that satisfies Properties 1 and 2. This statement has a universal quantifier over all possible values of RT. Such a quantifier is not available in FOL, but it is implicitly used in the definition of validity. This means that we can use the implicit universal quantification in $\Gamma \vDash RT(a,b)$ to check if (a,b) is included in all possible values of RT. The following theorem states our observations:

Theorem 4. (Verifying reachability in FOL) Let Δ be a set of FOL formula, RT a binary relational symbol that does not appear in Δ, and Γ be defined as follows:

$$\Gamma \quad := \quad \Delta \cup \{\forall x, y \bullet\ R(x,y) \Rightarrow RT(x,y),\ \forall x, y, z \bullet RT(x,y) \wedge RT(y,z) \Rightarrow RT(x,z)\}$$

We have the following:

$$\Delta \vDash R^+(a,b) \quad \text{iff} \quad \Gamma \vDash RT(a,b)$$

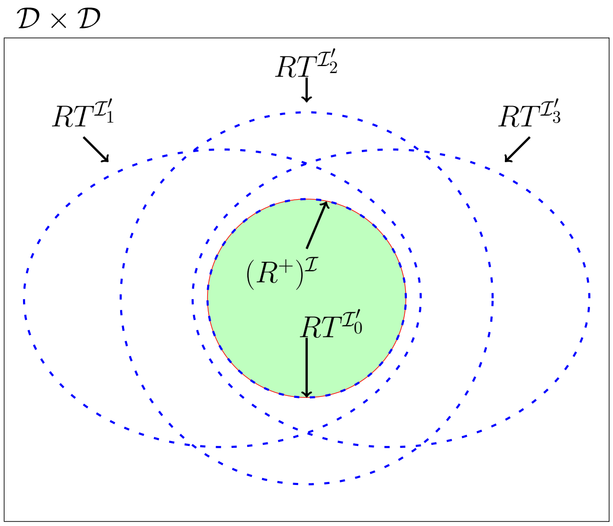


Figure 4.1: Possible values for RT. Let $\mathcal{I} = \langle \mathcal{D}, . \rangle$, $\mathcal{I} \Vdash \Delta$, and $\mathcal{I}' = \mathcal{I}$ plus an interpretation for RT.

Proof. We need to prove two statements:

1. If $\Delta \vDash R^+(a,b)$ then $\Gamma \vDash RT(a,b)$.
2. If $\Delta \nvDash R^+(a,b)$ then $\Gamma \nvDash RT(a,b)$.

- **Case (1)** suppose $\Delta \vDash R^+(a,b)$, and $\mathcal{I}'$ is an interpretation such that $\mathcal{I}' \Vdash \Gamma$. We need to show that $\mathcal{I}' \Vdash RT(a,b)$. Since Δ is a subset of Γ, we can conclude that the interpretation $\mathcal{I} = \mathcal{I}' - \{RT\}$ satisfies Δ: $\mathcal{I} \Vdash \Delta$. Moreover, since $\Delta \vDash R^+(a,b)$, by the definition of validity we have:

  $$\mathcal{I} \Vdash R^+(a,b)$$

  Because of the two constraints that we have added to Δ to derive Γ, for every interpretation $\mathcal{I}'$ we have $R^{+\mathcal{I}'} \subseteq RT^{\mathcal{I}}$, we can conclude that $\mathcal{I}' \Vdash RT(a,b)$.

- **Case (2)** suppose $\Delta \nvDash R^+(a,b)$. We need to show that an interpretation $\mathcal{I}'$ exists such that $\mathcal{I}' \Vdash \Gamma$ and $\mathcal{I}' \nVdash RT(a,b)$. Since $\Delta \nvDash R^+(a,b)$, there exists an interpretation $\mathcal{I}$ such that it satisfies every formula in Δ and $\mathcal{I} \nVdash R^+(a,b)$. Extending $\mathcal{I}$ to an interpretation $\mathcal{I}'$ that is the same as $\mathcal{I}$ except it maps RT to $RT^{+\mathcal{I}}$ results in an interpretation that satisfies Γ, but does not satisfy $RT(a,b)$.

Temporal part		
φ	$::=$	$\pi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$
	$::=$	$\mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi$
	$::=$	$\varphi_1 \mathbf{EU} \varphi_2 \mid \varphi_1 \mathbf{AU} \varphi_2$
Propositional part		
π	$::=$	$P \mid \neg\pi \mid \pi_1 \vee \pi_2$
	where P is a labelling predicate.	

Figure 4.2: CTL-Live

□

Theorem 4 on page 31 states an interesting relation between expressing and verifying in FOL: verifying a property in FOL does not necessarily imply the expressibility of that property in it. It may be possible to verify a property without expressing it. Reachability is an example of a property that is not expressible in FOL and yet it is verifiable.

4.2 CTL-Live Model Checking

The are two major mechanisms behind Theorem 4 on page 31:

1. Validity in FOL provides an implicit higher-order universal quantifier.
2. An element belongs to the smallest set in a family of sets iff it belongs to all the sets in the family.

These two points allowed us to turn a problem of the form $\Delta \models R^+(a, b)$ into $\Gamma \models RT(a, b)$ where everything is in FOL and there is no occurrence of transitive closure. In this section, we apply the same ideas to turn model checking into validity checking. Figure 4.2 presents *CTL-Live*: a subset of CTL whose model checking problem is reducible to FOL validity checking. CTL-Live contains the temporal connectives that are usually used to express liveness properties.

CTL-Live's grammar has two parts: temporal and propositional. CTL-Live disallows a temporal connective to be within the scope of negation ($\neg$); e.g., the CTL formula $\neg(\mathbf{AF}\ P)$ is not part of CTL-Live, but $\mathbf{AF}\ (\neg P)$ is.

Reduction Procedure:

`INPUT:`

Σ : *a symbolic Kripke structure.*

φ : *a CTL-Live formula.*

`OUTPUT:`

$\Sigma \cup \texttt{CTLive2FOL(}\varphi\texttt{)} \vDash \forall s \bullet \ S_0(s) \Rightarrow \lceil \varphi \rceil (s)$

Figure 4.3: Reduction Procedure

We present a function `CTLive2FOL( )` that takes a CTL-Live formula φ as input and generates a finite set of FOL formulae that represent the satisfiability of φ. The function `CTLive2FOL( )` introduces a new relational symbol with arity 1 for every sub-formula of φ including φ itself. We use the notation $\lceil \varphi \rceil$ to refer to the relational symbol introduced by `CTLive2FOL( )` for the formula φ. The formulas generated by `CTLive2FOL( )` are constraints over these new relational symbols.

Figure 4.3 is an overview of our reduction. The input of the reduction is an SKS Σ and a CTL-Live formula φ. The reduction procedure asserts whether the union of Σ with the formulas generated by `CTLive2FOL(`φ`)` entails $\forall s \bullet \ S_0(s) \Rightarrow \lceil \varphi \rceil (s)$. We prove that the validity generated by the reduction procedure of Figure 4.3 holds if and only if the SKS Σ satisfies the CTL-Live formula φ ($\Sigma \vDash_c \varphi$). We dedicate the rest of this section to presenting the details of `CTLive2FOL()` and to prove the correctness of our reduction.

Figure 4.4 on the facing page presents the reduction function. In this function, $\lceil \varphi \rceil$ is a new relational symbol that is introduced by `CTLive2FOL( )` for the formula φ. For every sub-formula φ' of φ, `CTLive2FOL()` defines one or two FOL formulas over each new relational symbol, $\lceil \varphi' \rceil$. The complexity of `CTLive2FOL( )` is linear with respect to the size of φ.

The main contribution of this section is the following equivalence:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \Sigma \cup \texttt{CTLive2FOL(}\varphi\texttt{)} \vDash S_0 \subseteq \lceil \varphi \rceil$$

where Σ is an SKS and φ is a CTL-Live formula. We prove three lemmas that assist us in proving this equivalence. For a Kripke structure $\mathcal{K}$, these lemmas study the relationship between the set of states of $\mathcal{K}$ that satisfy the formula φ ($[\varphi]_{\mathcal{K}}$) and the set of states identified by the labelling predicate $\lceil \varphi \rceil$ ($\lceil \varphi \rceil^{\mathcal{K}}$). In $[\varphi]_{\mathcal{K}}$, we consider $\mathcal{K}$ as a Kripke structure, whereas in $\lceil \varphi \rceil^{\mathcal{K}}$, $\mathcal{K}$ is an interpretation.

```
CTLive2FOL($\varphi$):
    case $\varphi$ of
 1.  $P$                         =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow P(s)\}$            where $P$ is a labelling predicate
 2.  $\neg\varphi_1$             =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \neg\lceil\varphi_1\rceil(s)\}$ $\cup$ CTLive2FOL($\varphi_1$)
 3.  $\varphi_1 \vee \varphi_2$  =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \lceil\varphi_1\rceil(s) \vee \lceil\varphi_2\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$) $\cup$ CTLive2FOL($\varphi_2$)
 4.  $\varphi_1 \wedge \varphi_2$ =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \lceil\varphi_1\rceil(s) \wedge \lceil\varphi_2\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$) $\cup$ CTLive2FOL($\varphi_2$)
 5.  $\mathbf{EX}\varphi_1$      =>  $\{\forall s \bullet\ \big(\exists s' \bullet\ N(s,s') \wedge \lceil\varphi_1\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ $\cup$ CTLive2FOL($\varphi_1$)
 6.  $\mathbf{AX}\varphi_1$      =>  $\{\forall s \bullet\ \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi_1\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ $\cup$ CTLive2FOL($\varphi_1$)
 7.  $\mathbf{EF}\varphi_1$      =>  $\{\lceil\varphi_1\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \big(\exists s' \bullet\ N(s,s') \wedge \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$)
 8.  $\mathbf{AF}\varphi_1$      =>  $\{\lceil\varphi_1\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$)
 9.  $\varphi_1\mathbf{EU}\varphi_2$ =>  $\{\lceil\varphi_2\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \lceil\varphi_1\rceil(s) \wedge \big(\exists s' \bullet\ N(s,s') \wedge \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$) $\cup$ CTLive2FOL($\varphi_2$)
10.  $\varphi_1\mathbf{AU}\varphi_2$ =>  $\{\lceil\varphi_2\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \lceil\varphi_1\rceil(s) \wedge \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\} \cup$
                                       CTLive2FOL($\varphi_1$) $\cup$ CTLive2FOL($\varphi_2$)
```

Figure 4.4: Definition of CTLive2FOL(φ)

If a CTL-Live formula π is derived from only the propositional part of Figure 4.2, the sets $[\pi]_{\mathcal{K}}$ (the set of states in the Kripke structure $\mathcal{K}$ that satisfy the CTL-Live formula π) and $\lceil \pi \rceil^{\mathcal{K}}$ (the set of states assigned to the labelling predicate $\lceil \pi \rceil$ by the interpretation $\mathcal{K}$) for every $\mathcal{K}$ such that $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\pi)$ are equal. This is due to the fact that the constraints that are defined by $\texttt{CTLive2FOL}(\pi)$ for these connectives are necessary and *sufficient* to characterize the set of states that satisfy π:

Lemma 2. Let Σ be an SKS, π a CTL-Live formula derived from the propositional part of Figure 4.2 on page 33. For every Kripke structure $\mathcal{K}$ such that $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\pi)$ the following holds:

$$[\pi]_{\mathcal{K}} = \lceil \pi \rceil^{\mathcal{K}}$$

Proof. Proof by structural induction on π. In the following cases, we assume $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\varphi)$.

- **Base case:** let $\pi = P$ where P is a labelling predicate. For every s, we have:

  $$\begin{array}{lll} s \in [P]_{\mathcal{K}} & \text{iff} \quad s \in P^{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} \quad s \in \lceil P \rceil^{\mathcal{K}} & \textit{by Line 1 in definition of } \texttt{CTLive2FOL( )} \end{array}$$

  therefore $[P]_{\mathcal{K}} = \lceil P \rceil^{\mathcal{K}}$.

- **Induction step:** according to the structure of π, two cases are distinguished having $[\pi_1]_{\mathcal{K}} = \lceil \pi_1 \rceil^{\mathcal{K}}$ and $[\pi_2]_{\mathcal{K}} = \lceil \pi_2 \rceil^{\mathcal{K}}$ as induction hypotheses:

  1. Let $\pi = \neg \pi_1$. For every s, we have:

     $$\begin{array}{lll} s \in [\neg \pi_1]_{\mathcal{K}} & \text{iff} \quad s \notin [\pi_1]_{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} \quad s \notin \lceil \pi_1 \rceil^{\mathcal{K}} & \textit{by induction hypothesis} \\ & \text{iff} \quad s \in \lceil \neg \pi_1 \rceil^{\mathcal{K}} & \textit{by Line 2 in definition of } \texttt{CTLive2FOL()} \end{array}$$

     therefore $[\neg \pi_1]_{\mathcal{K}} = \lceil \neg \pi_1 \rceil^{\mathcal{K}}$.

  2. Let $\pi = \pi_1 \vee \pi_2$. For every s, we have:

     $$\begin{array}{lll} s \in [\pi_1 \vee \pi_2]_{\mathcal{K}} & \text{iff} \quad s \in [\pi_1]_{\mathcal{K}} \vee s \in [\pi_2]_{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} \quad s \in \lceil \pi_1 \rceil^{\mathcal{K}} \vee s \in \lceil \pi_2 \rceil^{\mathcal{K}} & \textit{by induction hypotheses} \\ & \text{iff} \quad s \in \lceil \pi_1 \vee \pi_2 \rceil^{\mathcal{K}} & \textit{by Line 3 in definition of } \texttt{CTLive2FOL()} \end{array}$$

     therefore $[\pi_1 \vee \pi_2]_{\mathcal{K}} = \lceil \pi_1 \vee \pi_2 \rceil^{\mathcal{K}}$.

□

A similar result to Lemma 2 can be proven for the CTL-Live formulas that are derived from the temporal part of Figure 4.2 on page 33. The difference is that the set $[\varphi]_{\mathcal{K}}$

is a subset of $\lceil\varphi\rceil^{\mathcal{K}}$ rather than being equal to it. The reason is that the constraints $\texttt{CTLive2FOL(}\varphi\texttt{)}$ do not completely characterize $[\varphi]_{\mathcal{K}}$: these constraints are necessary but they are not sufficient; as a result, the set $\lceil\varphi\rceil^{\mathcal{K}}$ includes $[\varphi]_{\mathcal{K}}$ and possibly some other states.

Lemma 3. Let Σ be an SKS, φ a CTL-Live. The following holds:

$$\forall\, \mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL(}\varphi\texttt{)} \bullet\ [\varphi]_{\mathcal{K}} \subseteq \lceil\varphi\rceil^{\mathcal{K}}$$

Proof. Proof by structural induction on φ. In the following cases, we assume $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL(}\varphi\texttt{)}$.

- **Base case:** let $\varphi = \pi$ where π is a CTL-Live formula derived from the propositional part of Figure 4.2 on page 33. By Lemma 2 on the preceding page, $[\pi]_{\mathcal{K}} = \lceil\pi\rceil^{\mathcal{K}}$, and therefore $[\pi]_{\mathcal{K}} \subseteq \lceil\pi\rceil^{\mathcal{K}}$.
- **Induction step:** according to the structure of φ, eight cases are distinguished having $[\varphi_1]_{\mathcal{K}} \subseteq \lceil\varphi_1\rceil^{\mathcal{K}}$ and $[\varphi_2]_{\mathcal{K}} \subseteq \lceil\varphi_2\rceil^{\mathcal{K}}$ as induction hypotheses. Since $\mathbf{EF}\varphi$ ($\mathbf{AF}\varphi$) is equivalent to $\top\mathbf{EU}\varphi$ ($\top\mathbf{AU}\varphi$), we only go through six cases:

  1. Let $\varphi = \varphi_1 \vee \varphi_2$. For every s, we have:

$s \in [\varphi_1 \vee \varphi_2]_{\mathcal{K}}$	implies	$s \in [\varphi_1]_{\mathcal{K}} \vee s \in [\varphi_2]_{\mathcal{K}}$	*by semantics of CTL*
	implies	$s \in \lceil\pi_1\rceil^{\mathcal{K}} \vee s \in \lceil\pi_2\rceil^{\mathcal{K}}$	*by induction hypotheses*
	implies	$s \in \lceil\pi_1 \vee \pi_2\rceil^{\mathcal{K}}$	*by Line 3 in definition of `CTLive2FOL( )`*

     therefore $[\varphi_1 \vee \varphi_2]_{\mathcal{K}} \subseteq \lceil\varphi_1 \vee \varphi_2\rceil^{\mathcal{K}}$.
  2. Let $\varphi = \varphi_1 \wedge \varphi_2$. Proof is similar to previous case.
  3. Let $\varphi = \mathbf{EX}\ \varphi_1$. For every s, we have:

$s \in [\mathbf{EX}\ \varphi_1]_{\mathcal{K}}$	implies	$\exists s' \bullet\ N^{\mathcal{K}}(s,s') \wedge s' \in [\varphi_1]_{\mathcal{K}}$	*by semantics of CTL*
	implies	$\exists s' \bullet\ N^{\mathcal{K}}(s,s') \wedge s' \in \lceil\varphi_1\rceil^{\mathcal{K}}$	*by induction hypothesis*
	implies	$s \in \lceil\mathbf{EX}\ \varphi_1\rceil^{\mathcal{K}}$	*by Line 5 in definition of `CTLive2FOL( )`*

     therefore $[\mathbf{EX}\ \varphi_1]_{\mathcal{K}} \subseteq \lceil\mathbf{EX}\ \varphi_1\rceil^{\mathcal{K}}$.
  4. Let $\varphi = \mathbf{AX}\ \varphi_1$. Proof is similar to previous case.
  5. Let $\varphi = \varphi_1\ \mathbf{EU}\ \varphi_2$, and $s \in [\varphi_1\ \mathbf{EU}\ \varphi_2]_{\mathcal{K}}$. By induction on $\texttt{rank(}s\texttt{)}$ (Definition 14 on page 16), we prove that $s \in \lceil\varphi_1\ \mathbf{EU}\ \varphi_2\rceil^{\mathcal{K}}$.

- **Base case:** suppose `rank(`s`)` $= 0$; in this case, the state s itself satisfies φ_2, which means $s \in [\varphi_2]_{\mathcal{K}}$. Using the induction hypotheses from the outer induction, we have $s \in \lceil \varphi_2 \rceil^{\mathcal{K}}$, and according to the first constraint in Line 9 of Figure 4.4 on page 35, we have $s \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$; therefore, if `rank(`$s$`)` $= 0$ then $s \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$.
- **Induction step:** suppose `rank(`s`)` $= m + 1$. The induction hypotheses for this inner induction is: if $s' \in [\varphi_1 \textbf{ EU } \varphi_2]_{\mathcal{K}}$ and `rank(`s'`)` $= m$, then $s' \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$. Since `rank(`$s$`)` $= m + 1 \neq 0$, $s \in [\varphi_1]_{\mathcal{K}}$ and there exists s' such that $N^{\mathcal{K}}(s, s') \wedge \ s' \in [\varphi_1 \textbf{ EU } \varphi_2]_{\mathcal{K}}$, and `rank(`$s'$`)` $= m$. According to the induction hypotheses of the inner induction, we have:

  $$\exists s' \bullet N^{\mathcal{K}}(s, s') \wedge s' \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$$

  Using the induction hypotheses of the outer induction ($[\varphi_1]_{\mathcal{K}} \subseteq \lceil \varphi_1 \rceil^{\mathcal{K}}$), we derive $s \in \lceil \varphi_1 \rceil^{\mathcal{K}}$. According to the second constraint in Line 9 of Figure 4.4 on page 35, and the above property, we have: $s \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$; therefore if `rank(`$s$`)` $= m + 1$ then $s \in \lceil \varphi_1 \textbf{ EU } \varphi_2 \rceil^{\mathcal{K}}$.

6. Let $\varphi = \varphi_1 \textbf{ AU } \varphi_2$. Proof is similar to previous case.

□

Another way of proving Part 5 (6) in the induction step of Lemma 3 on the previous page is to consider the encoding of CTL in mu-calculus [25]: the set of states satisfying $\varphi_1 \textbf{EU} \varphi_2$ ($\varphi_1 \textbf{AU} \varphi_2$) is the smallest set that satisfies the constraints in Line 9 (10) of `CTLive2FOL()`.

Lemma 4. Let Σ be an SKS and φ a CTL-Live formula. For every $\mathcal{K} \Vdash \Sigma$ there exists $\mathcal{K}' \Vdash \Sigma \cup$ `CTLive2FOL(`φ`)` such that:

$$S_0^{\mathcal{K}} = S_0^{\mathcal{K}'} \quad \text{and} \quad [\varphi]_{\mathcal{K}} = [\varphi]_{\mathcal{K}'} = \lceil \varphi \rceil^{\mathcal{K}'}$$

Proof. Suppose Σ is over Kripke base $\mathscr{B}$ and $\mathcal{K} \Vdash \Sigma$. We define $\mathcal{K}'$ as an interpretation that has the same domain as $\mathcal{K}$ and its mapping function is defined as follows:

$$X^{\mathcal{K}'} = \begin{cases} X^{\mathcal{K}} & X \in \mathscr{B}, \\ [\psi]_{\mathcal{K}} & X = \lceil \psi \rceil \text{ where } \psi \text{ is a sub-formula of } \ \varphi \end{cases}$$

According to the semantics of CTL, the constraints of `CTLive2FOL(`φ`)` for each sub-formula ψ of φ are necessary constraints that the sets $[\psi]_{\mathcal{K}}$ satisfy; therefore, $\mathcal{K}'$ is a satisfying interpretation of $\Sigma \cup$ `CTLive2FOL(`φ`)`. □

Now, we present the main contribution of this section:

Theorem 5. (CTL-Live model checking as FOL validity checking) Let Σ be an SKS, and φ a CTL-Live formula from Figure 4.2 on page 33. We have the following property:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \Sigma \cup \texttt{CTLive2FOL}(\varphi) \vDash S_0 \subseteq \lceil \varphi \rceil$$

Proof. We need to prove two statements:

1. If $\Sigma \vDash_c \varphi$ then $\Sigma \cup \texttt{CTLive2FOL}(\varphi) \vDash S_0 \subseteq \lceil \varphi \rceil$.
2. If $\Sigma \nvDash_c \varphi$ then $\Sigma \cup \texttt{CTLive2FOL}(\varphi) \nvDash S_0 \subseteq \lceil \varphi \rceil$.

- **Case (1)** suppose $\Sigma \vDash_c \varphi$, and $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\varphi)$. We need to show that $\mathcal{K} \Vdash S_0 \subseteq \lceil \varphi \rceil$. From $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\varphi)$, we can conclude $\mathcal{K} \Vdash \Sigma$, and since $\Sigma \vDash_c \varphi$, we have the following:

  $$S_0^{\mathcal{K}} \subseteq [\varphi]_{\mathcal{K}}$$

  According to Lemma 3 on page 37, $[\varphi]_{\mathcal{K}} \subseteq \lceil \varphi \rceil^{\mathcal{K}}$. By the transitivity of the subset relation over sets, we can conclude the following:

  $$S_0^{\mathcal{K}} \subseteq \lceil \varphi \rceil^{\mathcal{K}}$$

  and by the semantics of FOL, we have our goal:

  $$\mathcal{K} \Vdash S_0 \subseteq \lceil \varphi \rceil$$

- **Case (2)** suppose $\Sigma \nvDash_c \varphi$. We need to show that there exists an interpretation $\mathcal{K}$ such that $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\varphi)$ and $\mathcal{K} \nVdash S_0 \subseteq \lceil \varphi \rceil$. Since $\Sigma \nvDash_c \varphi$, there exists $\mathcal{K}_0 \Vdash \Sigma$ and $S_0^{\mathcal{K}_0} \nsubseteq [\varphi]_{\mathcal{K}_0}$. By Lemma 4 on the facing page, there exists $\mathcal{K}$ such that $\mathcal{K} \Vdash \Sigma \cup \texttt{CTLive2FOL}(\varphi)$ and it has the following property:

  $$S_0^{\mathcal{K}_0} = S_0^{\mathcal{K}} \quad \text{and} \quad [\varphi]_{\mathcal{K}_0} = [\varphi]_{\mathcal{K}} = \lceil \varphi \rceil^{\mathcal{K}}$$

  from the above property, we can conclude that $S_0^{\mathcal{K}} \nsubseteq \lceil \varphi \rceil^{\mathcal{K}}$; therefore, $\mathcal{K} \nVdash S_0 \subseteq \lceil \varphi \rceil$.

□

4.3 Maximality of CTL-Live

Theorem 5 on the previous page shows that model checking CTL-Live is reducible to validity checking in FOL. The logical connectives that are not included in CTL-Live are **EG**, **AG**, and $\neg$ over temporal connectives. To show model checking of these three connectives is not reducible to FOL validity checking, we reduce the complement of the halting problem on an empty tape for a deterministic Turing machines (DTM) to the model checking of **EG** and **AG** against an SKS. The complement of the halting problem is not recursively enumerable [28] but FOL validity checking is; therefore, model checking of **EG** and **AG** cannot be reduced to FOL validity checking, otherwise, the complement of halting problem would be recursively enumerable. We call this result the *maximality of CTL-Live*.

In the following, we assume a DTM $\mathcal{M} = \langle \mathcal{Q}, \delta \rangle$ is a pair where $\mathcal{Q} = \{q_0, \ldots, q_n\}$ is a finite set of states, the tape alphabet is $\{b, 0\}$ and δ, the transition function, is a total function from $\mathcal{Q} \times \{b, 0\}$ to $\mathcal{Q} \times \{b, 0\} \times \{L, R\}$. For example, the transition $\delta(q_9, 0) = (q_2, b, L)$ means that if the DTM $\mathcal{M}$ is at state q_9 and the tape head is scanning 0, in the next step, $\mathcal{M}$ goes to state q_2, writes b on the tape, and moves the head to the Left.

A DTM $\mathcal{M} = \langle \mathcal{Q}, \delta \rangle$ starts in the state q_0. We consider $\mathcal{M}$ to have halted if it reaches state q_n. The tape is one way infinite. In the initial state, the head tape is on the left most square of the tape, and every square on the tape is blank (b).

The idea behind reducing the complement of the halting problem on an empty tape for a DTM to model checking of **EG** or **AG** is that the set of all the configurations of a DTM can be considered as the state space for a Kripke structure and the next state relation of this Kripke structure can be derived from the transition function of the DTM. Since the underlying DTM is deterministic, this Kripke structure has only one computation path, and therefore, satisfying **EG** and **AG** would be equivalent. The **G**lobaly part of **EG** and **AG** can be used to state that no state along the path is a halting state.

Lemma 5. Let $\mathcal{M} = \langle \mathcal{Q}, \delta \rangle$ be a DTM; the complement of the halting problem on an empty tape for $\mathcal{M}$ is reducible to model checking of an **EG** formula against an SKS.

Proof. To prove this lemma, we create an SKS Σ from $\mathcal{M}$ such that Σ satisfies an **EG** formula iff $\mathcal{M}$ does not halt on an empty tape. The SKS that encodes $\mathcal{M}$ is created over the following Kripke base:

$$\mathscr{B} = \{0,\ inc/1,\ dec/1,\ Q/1,\ H/1\} \cup \{b/2,\ S_0/1,\ N/2,\ halt/1\}$$

The constant 0 represents number zero. The functional symbols $inc/1$ and $dec/1$ are used to model increment and decrement operations on natural numbers. We can refer to a certain natural number by applying *inc* to 0; e.g., number 2 is represented as $inc(inc(0))$, for short $inc^2(0)$. In this lemma and the following, natural numbers are short forms of their representations using this base; e.g., in the formula $Q(t) = 2$, the symbol 2 is a short form of $inc^2(0)$.

The natural numbers are used to represent configurations of $\mathcal{M}$: the position of the tape head, the current state of $\mathcal{M}$, and to point to different squares of the tape. The binary relational symbol $b(t, i)$ represents whether at step t the i^{th} square is blank or 0. The functional symbol $Q(t) = i$ represents that the state of $\mathcal{M}$ at step t is q_i, and the functional symbol $H(t) = i$ represents that the tape head of $\mathcal{M}$ at step t is on the i^{th} square. The relational symbols S_0 and N are used to model the initial state and the next state relation of the SKS. The relational symbol *halt* is a labelling predicate used to represent whether a state is a halting state.

The constraints in the SKS Σ are divided into 5 parts:

1. Constraints for encoding an "acceptable" semantics for 0, *inc*, and *dec*:
   - $\forall i \bullet \; inc(i) \neq 0$
   - $\forall i, i' \bullet \; inc(i) = inc(i') \Rightarrow i = i'$
   - $\forall i \bullet \; i \neq 0 \Rightarrow \big(\exists i' \bullet \; inc(i') = i\big)$
   - $dec(0) = 0$
   - $\forall i \bullet \; dec(inc(i)) = i$
   - $\forall i \bullet \; i \neq 0 \Rightarrow inc(dec(i)) = i$
2. Constraint stating that at each step of computation at most one position of the tape can be changed:
   $$\forall t, i \bullet \; H(t) \neq i \Rightarrow \big(b(t, i) \Leftrightarrow b(inc(t), i)\big)$$
3. Constraints for encoding the initial configuration of $\mathcal{M}$:
   - $Q(0) = 0$: at step 0, $\mathcal{M}$ is at state q_0,
   - $H(0) = 0$: at step 0, the tape head of $\mathcal{M}$ is at position 0,
   - $\forall i \bullet \; b(0, i)$: at step 0, every position of the tape is blank.
4. Constraints for encoding the transition function δ: for every pair in $\mathcal{Q} \times \{b, 0\}$ we have a formula that mimics the computation of $\mathcal{M}$. For example, the formula that simulates $\delta(q_9, 0) = (q_2, b, L)$ is

$\forall t \bullet\ Q(t) = 9 \ \wedge\ \neg b(t, H(t)) \ \Rightarrow$
$$Q(inc(t)) = 2 \ \wedge\ b(inc(t), H(t)) \ \wedge\ H(inc(t)) = dec(H(t))$$
and the formula that simulates $\delta(q_6, b) = (q_7, 0, R)$ is
$\forall t \bullet\ Q(t) = 6 \ \wedge\ b(t, H(t)) \ \Rightarrow$
$$Q(inc(t)) = 7 \ \wedge\ \neg b(inc(t), H(t)) \ \wedge\ H(inc(t)) = inc(H(t))$$

5. Constraints for the initial state, next state relation, and halting state of the corresponding Kripke structure. We use natural numbers as the state space of a Kripke structure. The configuration of $\mathcal{M}$ at state (step) t is represented by $Q(t)$, $H(t)$, and $b(t, .)$:
   - $\forall t \bullet\ S_0(t) \Leftrightarrow t = 0$: initial state,
   - $\forall t, t' \bullet\ N(t, t') \Leftrightarrow t' = inc(t)$: next state relation,
   - $\forall t \bullet\ halt(t) \Leftrightarrow Q(t) = n$: halting states.

We claim that the following holds:

$$\Sigma \vDash_c \mathbf{EG}\neg halt \quad \text{iff} \quad \mathcal{M} \text{ does not halt on an empty tape.}$$

We need to prove two statements:

1. If $\Sigma \vDash_c \mathbf{EG}\neg halt$, then $\mathcal{M}$ does not halt on an empty tape.
2. If $\mathcal{M}$ does not halt on an empty tape, then $\Sigma \vDash_c \mathbf{EG}\neg halt$.

- **Case (1)** $\Sigma \vDash_c \mathbf{EG}\neg halt$ means that every Kripke structure that satisfies Σ, satisfies $\mathbf{EG}\neg halt$. The standard interpretation of natural numbers, which satisfies Σ, corresponds to the computation of $\mathcal{M}$. Since $\mathbf{EG}\neg halt$ means there exists a path along which halt is never true, and the DTM $\mathcal{M}$ is deterministic, and therefore has only one path, we can conclude that $\mathcal{M}$ does not halt on an empty tape.

- **Case (2)** By induction on the number of steps, we can prove that if at step t, $\mathcal{M}$ is at state q_i, every Kripke structure $\mathcal{K}$ that satisfies Σ, then $\mathcal{K} \Vdash Q(t) = i$. Assuming $\mathcal{M}$ does not halt on an empty tape, we can conclude that for every $\mathcal{K} \Vdash \Sigma$ has an infinite path starting at 0:
$$0 \mapsto 1 \mapsto 2 \mapsto 3 \mapsto ..$$
where none of them is the halting state q_n:
$$Q(0) \neq n,\ Q(1) \neq n,\ Q(2) \neq n,\ Q(3) \neq n, ..$$

Therefore, every $\mathcal{K} \Vdash \Sigma$ satisfies $\mathbf{EG}\neg \mathit{halt}$:

$$\Sigma \vDash_c \mathbf{EG}\neg \mathit{halt}$$

□

Lemma 6. Let $\mathcal{M} = \langle \mathcal{Q}, \delta \rangle$ be a DTM; the complement of the halting problem on an empty tape for $\mathcal{M}$ is reducible to model checking of an **AG** formula against an SKS.

Proof. To prove this lemma, we create an SKS Σ from $\mathcal{M}$ such that $\Sigma \vDash_c \mathbf{AG}\neg \mathit{halt}$ iff $\mathcal{M}$ does not halt on an empty tape. The SKS that we need to build for this reduction is same as the one in Lemma 5 on page 40. We need to prove two statements:

1. If $\Sigma \vDash_c \mathbf{AG}\neg \mathit{halt}$, then $\mathcal{M}$ does not halt on an empty tape.
2. If $\Sigma \nvDash_c \mathbf{AG}\neg \mathit{halt}$, then $\mathcal{M}$ halts on an empty tape.

- **Case (1)** similar to Case 1 of Lemma 5 on page 40.
- **Case (2)** since $\Sigma \vDash_c \mathbf{AG}\neg \mathit{halt}$, there exists a Kripke structure $\mathcal{K} \Vdash \Sigma$ that does not satisfy $\mathbf{AG}\neg \mathit{halt}$. This means that there is a finite path from an initial state of $\mathcal{K}$ that reaches a state satisfying *halt*. By induction on the length of this path, we can show that this finite path corresponds to a finite sequence of configurations in $\mathcal{M}$ that results in a halting configuration; as a result, $\mathcal{M}$ halts on an empty tape.

□

Using the two lemmas proved in this section, we present the main contribution of this section:

Theorem 6. (Maximality of CTL-Live) It is not possible to reduce model checking of **EG**, **AG**, or $\neg$ to FOL validity checking for all SKSs Σ; as a result, CTL-Live is the "largest" fragment of CTL that its model checking is reducible to FOL validity checking.

Proof. In Lemma 5 on page 40 (Lemma 6), we showed that the complement of the halting problem on an empty tape for a DTM is reducible to model checking of **EG** (**AG**). This problem is not recursively enumerable, as a result, it cannot be reduced to validity checking in FOL, which is a recursively enumerable problem. We also know that $\mathbf{EG}\varphi$ is equivalent to $\neg(\mathbf{AF}\neg\varphi)$. Since **AF** is include in this fragment, $\neg$ cannot be added as well. □

4.4 Incompleteness of Inductive Invariant Method

The verification of invariants is often of interest for safety properties of models. A property P is an *invariant* iff it holds in every *reachable* state of a Kripke structure. According to the semantics of CTL, P being an invariant of a Kripke structure $\mathcal{K}$ is equivalent to $\mathcal{K}$ satisfying **AG** P.

A property P is an *inductive invariant* for a Kripke structure $\mathcal{K}$ iff it satisfies the following two constraints:

1. $\forall s \bullet S_0(s) \Rightarrow P(s)$
2. $\forall s, s' \bullet \; P(s) \wedge N(s, s') \Rightarrow P(s')$

The first constraint states that every initial state satisfies P, and the second one states that if the state s satisfies P and s' is reachable from s in one step, then s' satisfies P. It is easy to see that every inductive invariant is also an invariant of a Kripke structure, but every invariant is not necessarily an inductive invariant. Checking if a property is an inductive invariant is computationally easier than checking if it is an invariant.

According to Theorem 6, model checking **AG** is not recursively enumerable, whereas, inductive invariant checking is. Motivated by this fact, the inductive invariant method to check if a property P is an invariant has gained popularity for both finite and infinite Kripke structures. Many researchers have found inductive invariants by hand. The method of IC3 [16] is a way to find automatically inductive invariants for finite systems, and this approach has been generalized in nuXmv [20] as an incomplete approach to find automatically inductive invariants for infinite state systems.

Generally speaking, the goal is to find an inductive invariant that is strong enough to prove the original invariant of interest. This method is essentially as follows: to prove that P is an invariant, first check if it is an inductive invariant; if it is not, then try to compute or guess an R so that $P \wedge R$ is an inductive invariant, and therefore, P is proved to be an invariant. The formula R tries to eliminate unreachable states that do not allow P to be an inductive invariant.

An important question is "does an R always exist when P is an invariant?" For finite Kripke structures, the answer is "yes" since the number of states is finite, R can enumerate all reachable states. However, for infinite state systems, we can now show that R is not guaranteed to exist.

Theorem 7. (Incompleteness of inductive invariant method) There exists a Kripke structure $\mathcal{K}$ and a property P such that P is an invariant of $\mathcal{K}$ and there is no formula R such that $P \wedge R$ is an inductive invariant for $\mathcal{K}$.

Proof. According to Lemma 6 on page 43, proving a DTM does *not* halt on an empty tape is reducible to proving that a formula named $\neg halt$ is an invariant. If an R exists then we can enumerate all R's and check if $\neg halt \wedge R$ is an inductive invariant in parallel; therefore, a semi-decision procedure for the complement of the halting problem exists and it is recursively enumerable. This is a contradiction, and as a result, such an R does not always exist. □

4.5 Some Decidability Results

In Theorem 5 on page 39, we show how to reduce model checking an SKS against a CTL-Live formula to FOL validity checking. Because validity for FOL is recursively enumerable, a semi-decision procedure can be used to generate a proof in a finite number of steps if the property holds. However, if the property does not hold, we cannot guarantee termination of the deduction process. In this section, we draw on decidability results for fragments of FOL to consider some restricted versions of the CTL-Live model checking problem that are decidable, and therefore guaranteed to terminate (with enough resources).

A fragment of FOL achieves decidability by putting restrictions on the allowed signatures and/or formulae. These restrictions reduce the expressive power of the fragment, and as a result, not all fragments of FOL can be used for CTL-Live model checking based on our reduction in Theorem 5 on page 39.

The FOL formulae that are generated by our `CTLive2FOL()` method have two main characteristics: 1) the use of quantifiers and 2) the introduction of unary relational symbols, which limit the relevant decidable fragments of FOL. For example, the theory of uninterpreted functions (UIF), which is decidable, includes the use of new relational/functional symbols but does not allow quantifiers, therefore UIF cannot be used for CTL-Live model checking based on Theorem 5 on page 39. Another decidable logic that cannot be used is Presburger arithmetic (PA). Unlike UIF, PA includes quantifiers, but it does not permit the introduction of a new relational/functional symbol. In the following, we show how the **AE** fragment of FOL [38] and a family of description logics (**DL**s) [6] can be used for decidable CTL-Live model checking based on Theorem 5 on page 39.

$$\begin{array}{|lll|}
\hline
\varphi & ::= & \pi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\
 & ::= & \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \varphi_1 \mathbf{EU} \varphi_2 \\
\hline
\pi & ::= & P \mid \neg\pi \mid \pi_1 \vee \pi_2 \\
 & \multicolumn{2}{l|}{\text{where } P \text{ is a labelling predicate.}} \\
\hline
\end{array}$$

Figure 4.5: Decidable fragment of CTL-Live based on **AE**

4.5.1 AE for CTL-Live Model Checking

The **AE** fragment of FOL includes restrictions on both the allowed signatures and formulas. A formula Φ is an **AE** formula iff no functional symbol with arity ≥ 1 occurs in Φ, and no existential quantifier is within the scope of a universal quantifier when Φ is in normal negation form. Equivalently, Φ belongs to the **AE** fragment iff Φ has a prenex normal form (PNF):

$$\exists x_1, .., \exists x_n, \forall y_1, .., \forall y_m : \ \Psi \ ,$$

where no functional symbol with arity ≥ 1 occurs in Ψ. Checking the satisfiability of a set of **AE** formulae is decidable.

If every formula in Σ belongs to **AE** and φ is a formula that belongs to the fragment of CTL-Live presented in Figure 4.5, then the model checking problem falls within **AE** and is therefore decidable. We must limit the CTL-Live connectives to those that have a PNF format that belongs to **AE**; as a result, the connectives **AX**, **AF**, and **AU** are left out.

Theorem 8. Let Σ be an SKS such that every formula in Σ belongs to the **AE** fragment of FOL, and φ a formula derived from the grammar in Figure 4.5. Checking $\Sigma \models_c \varphi$ is reducible to satisfiability checking of a set of **AE** formulas, and as a result, it is decidable.

Proof. Based on Theorem 5 on page 39 and the semantics of FOL, $\Sigma \models_c \varphi$ holds iff the following set of FOL formulas is unsatisfiable:

$$\Sigma \cup \texttt{CTLive2FOL(}\varphi\texttt{)} \cup \{S_0 \not\subseteq \lceil\varphi\rceil\}$$

Since every formulae in Σ belongs to **AE**, all we need to show is that `CTLive2FOL(`φ`)` and $S_0 \not\subseteq \lceil\varphi\rceil$ also belongs to **AE**. The formula $S_0 \not\subseteq \lceil\varphi\rceil$ in PNF is $\exists s \bullet \ S_0(s) \wedge \neg\lceil\varphi\rceil$ and every formula in `CTLive2FOL(`φ`)`, where φ is derived from the grammar in Figure 4.5, belongs to **AE** as well. For example, the formulas generated for **EF** by `CTLive2FOL()` are:

1. $\lceil\varphi_1\rceil \subseteq \lceil\varphi\rceil$

2. $\forall s \bullet \; \big(\exists s' \bullet \; N(s,s') \wedge \lceil \varphi \rceil (s')\big) \Rightarrow \lceil \varphi \rceil (s)$

and their PNF belongs to **AE**:

1. $\forall s \bullet \; \neg \lceil \varphi_1 \rceil (s) \vee \lceil \varphi \rceil (s)$
2. $\forall s, s' \bullet \; \neg N(s,s') \vee \neg \lceil \varphi \rceil (s') \vee \lceil \varphi \rceil (s)$

□

The **AE** fragment of FOL is decidable because it has the *finite model property*: a set of **AE** formula is satisfiable iff it has a finite satisfying interpretation. The size of this interpretation can be calculated based on the structure of an **AE** formula. From a model checking perspective, the finite model property of **AE** implies that an SKS expressed in the **AE** fragment is always isomorphic to a finite Kripke structure; in other words, a "truly" infinite Kripke structure is not expressible in the **AE** fragment. In the next subsection, we consider another decidable fragment of FOL that does not have the finite model property and it can be used for CTL-Live model checking based on Theorem 5 on page 39.

4.5.2 DLs for CTL-Live Model Checking

The formulas in a description logic (**DL**) are made from atomic concepts and atomic roles. *Atomic concepts* and *roles* are relational symbols having arity 1 and 2 respectively. Intuitively, an atomic concept is a set and an atomic role is a binary relation. Different **DL**s provide different sets of operators to create general concepts (sets) and roles (binary relations) from the existing ones.

The DL $\mathcal{ALC}$ provides the operators: complement ($\neg$), intersection ($\sqcap$), union ($\sqcup$), join ($\exists\,\mathbf{.}$), and universal join ($\forall\,\mathbf{.}$). An interpretation $\mathcal{I}$ with the domain $\mathcal{D}$, maps an atomic concept A to a subset of $\mathcal{D}$ and an atomic role R to a subset of $\mathcal{D} \times \mathcal{D}$ (as in FOL). Figure 4.6 on the following page presents the syntax and the semantics of general concepts extending the mapping for atomic concepts and roles.

For two general concepts C_1 and C_2, $C_1 \sqsubseteq C_2$ is a *general concept inclusion* (GCI) formula, which intuitively states that the set C_1 is a subset of C_2[1]. Formally, an interpretation $\mathcal{I}$ satisfies $C_1 \sqsubseteq C_2$ iff C_1 under $\mathcal{I}$ is a subset of C_2 under $\mathcal{I}$:

$$\mathcal{I} \Vdash C_1 \sqsubseteq C_2 \;\; \text{iff} \;\; C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$$

[1] The notation $C_1 \equiv C_2$ is a short form for $C_1 \sqsubseteq C_2,\ C_2 \sqsubseteq C_1$.

Syntax

$$\begin{array}{rcl} C & ::= & A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \\ & ::= & \exists R \,.\, C \mid \forall R \,.\, C \end{array}$$

Semantics

$$\begin{array}{rcl} (\neg C)^{\mathcal{I}} & := & \mathcal{D} \setminus C^{\mathcal{I}} \\ (C_1 \sqcap C_2)^{\mathcal{I}} & := & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\ (C_1 \sqcup C_2)^{\mathcal{I}} & := & C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\ (\exists R \,.\, C)^{\mathcal{I}} & := & \{a \mid \exists b \in C^{\mathcal{I}} \bullet \ (a,b) \in R^{\mathcal{I}}\} \\ (\forall R \,.\, C)^{\mathcal{I}} & := & \{a \mid \forall b \bullet \ (a,b) \in R^{\mathcal{I}} \Rightarrow b \in C^{\mathcal{I}}\} \end{array}$$

Figure 4.6: $\mathcal{ALC}$: A and R are atomic concepts and roles.

Given a set of GCIs Γ and a general concept C, the *concept satisfiability* problem, denoted by $\Gamma \Vdash_{dl} C$, is to determine if there exists an interpretation $\mathcal{I}$ that satisfies every GCI in Γ and $C^{\mathcal{I}}$ is not empty:

$$\Gamma \Vdash_{dl} C \ \text{ iff } \ \exists \mathcal{I} \Vdash \Gamma \bullet \ C^{\mathcal{I}} \neq \emptyset$$

The concept satisfiability problem for $\mathcal{ALC}$ is decidable.

For the fragment of CTL-Live at the top of Figure 4.7 on the next page, the model checking problem is reducible to concept satisfiability in $\mathcal{ALC}$ and therefore decidable. The formulas generated by `CTLive2FOL()` for a CTL-Live formula φ can be considered as set inclusion formulas and as a result, we can use the operators of $\mathcal{ALC}$ to express them. For example, the formulas generated for **EF** by `CTLive2FOL()` can be expressed in $\mathcal{ALC}$ as follows:

$$\begin{array}{rcl} \lceil \varphi_1 \rceil \subseteq \lceil \varphi \rceil & \rightsquigarrow & \lceil \varphi_1 \rceil \sqsubseteq \lceil \varphi \rceil \\ \forall s \bullet \ \big(\exists s' \bullet \ N(s,s') \wedge \lceil \varphi \rceil(s')\big) \Rightarrow \lceil \varphi \rceil(s) & \rightsquigarrow & \exists N \,.\, \lceil \varphi \rceil \sqsubseteq \lceil \varphi \rceil \end{array}$$

The bottom of Figure 4.7 on the facing page shows the function `CTLive2DL()` that is the $\mathcal{ALC}$ version of `CTLive2FOL()` (defined in Figure 4.4 on page 35).

Theorem 9. Let Σ be an SKS such that every formula in it is a GCI, and φ be a CTL-Live formula derivable from the grammar of Figure 4.7 on the facing page. We have:

$$\Sigma \models_c \varphi \quad \text{iff} \quad \Sigma \cup \texttt{CTLive2DL}((\texttt{)}\varphi) \nVdash_{dl} S_0 \sqcap \neg \lceil \varphi \rceil$$

Proof. By the definition of $\Vdash_{dl}$ and Theorem 5 on page 39. □

$$\begin{array}{lll}
\varphi & ::= & \pi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\
 & ::= & \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi \\
\hline
\pi & ::= & P \mid \neg\pi \mid \pi_1 \vee \pi_2 \\
 & \multicolumn{2}{l}{\text{where } P \text{ is a labelling predicate.}}
\end{array}$$

```
CTLive2DL($\varphi$):
   case $\varphi$ of
1. $P$              => $\lceil P \rceil := P$
2. $\neg\varphi_1$            => $\{\neg\lceil\varphi_1\rceil \equiv \lceil\varphi\rceil\}$ ∪ CTLive2DL($\varphi_1$)
3. $\varphi_1 \vee \varphi_2$      => $\{\lceil\varphi_1\rceil \sqcup \lceil\varphi_2\rceil \equiv \lceil\varphi\rceil\}$ ∪ CTLive2DL($\varphi_1$) ∪ CTLive2DL($\varphi_2$)
4. $\varphi_1 \wedge \varphi_2$      => $\{\lceil\varphi_1\rceil \sqcap \lceil\varphi_2\rceil \equiv \lceil\varphi\rceil\}$ ∪ CTLive2DL($\varphi_1$) ∪ CTLive2DL($\varphi_2$)
5. $\mathbf{EX}\ \varphi_1$        => $\{\exists N.\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil\}$ ∪CTLive2DL(()$\varphi_1$)
6. $\mathbf{AX}\ \varphi_1$        => $\{\forall N.\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil\}$ ∪CTLive2DL(()$\varphi_1$)
7. $\mathbf{EF}\ \varphi_1$        => $\{\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil,\ \exists N.\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil\}$ ∪ CTLive2DL($\varphi_1$)
8. $\mathbf{AF}\ \varphi_1$        => $\{\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil,\ \forall N.\lceil\varphi_1\rceil \sqsubseteq \lceil\varphi\rceil\}$ ∪ CTLive2DL($\varphi_1$)
```

Figure 4.7: Decidable fragment of CTL-Live based on $\mathcal{ALC}$

$\mathcal{ALCFI}$ is another **DL** that extends $\mathcal{ALC}$ by *functionality* and *role-inverse* operators [6]. Since $\mathcal{ALCFI}$ includes $\mathcal{ALC}$, and the concept satisfiability problem is decidable for $\mathcal{ALCFI}$, we can also use it for decidable CTL-Live model checking. An interesting property of $\mathcal{ALCFI}$ is that it does not have the finite model property and yet it is decidable. From the model checking perspective, this means that we can describe "truly" infinite Kripke structures in $\mathcal{ALCFI}$ and have a decision procedure for verifying them.

4.6 Related Work

K-induction is a technique for unbounded model checking of safety properties of finite systems [53]. This technique extends bounded model checking by proving that bounded model checking for bound K is sufficient. The number K is dominated by the diameter of a Kripke structure. The diameter is computed iteratively using a SAT solver to check the equivalence of two formulae: the equivalence holds iff no new state can be reached by taking more than K steps. In [53], termination is guaranteed due to the finiteness of the Kripke structures under study.

Bultan, Gerber, and Pugh used Presburger formulae to represent infinite sets of states symbolically [17]. Their model checking approach requires a fixed-point calculation, and termination is achieved by using conservative approximation. This approach allows false negatives.

Kesten and Pnueli presented a sound and relatively complete (oracle based) deductive system for CTL* [45] to provide proof-like evidence for a model that satisfies a property. CTL-Live is less expressive than CTL* but based on the completeness of FOL, CTL-Live has a sound and complete deductive system, whereas CTL* does not have a complete deductive system.

Beyene, Popeea, and Rybalchenko encoded CTL model checking of infinite state systems into forall-exists quantified Horn clauses (which we call ExQH) [12]. The contribution of [12] is to develop a solver for ExQH and demonstrate its use for model checking CTL properties. Their method requires the models and the model checking constraints to be expressed in ExQH and to satisfy some well-foundedness conditions, whereas our results hold for any set of FOL constraints, which may describe multiple Kripke structures. Termination of their method is not guaranteed.

Existing decidability results for infinite state model checking, e.g., [34], are derived by restricting the form of Kripke structures and temporal properties. For example, Esparza shows that model checking **EF** for Petri nets having an effectively semilinear reachability relation is decidable. Our decidability results are by-products of our reduction of model checking to validity checking, and considering the decidable fragments of FOL. It would be interesting to compare the expressive power of decidable fragments of FOL for modelling symbolic Kripke structures with the restricted Kripke structures introduced in the existing decidability results.

Ben-David, Trefler, and Weddell reduced CTL model checking of finite Kripke structures to $\mathcal{ALC}$ concept satisfiability checking [11]. Their reduction relies on the finiteness of the Kripke structure, whereas ours does not and our decidability result with respect to **DL**s is for infinite systems as well.

Model checking a parametrized system means checking if every member of a (infinite) family of finite Kripke structures satisfies a temporal property [25]. Each member of such a family is derived by fixing a parameter, e.g., the number of processes that can execute. A symbolic Kripke structure can represent a family of Kripke structures, and our model checking approach is complete for parametrized systems as long as the model and its parameter are expressible in FOL. In most cases, the parameter is an integer and proving properties about all members requires induction and the generation of an inductive invariant; in these cases, FOL reasoning alone is not sufficient. Investigating the combination of our model checking technique with verification methods for parametrized systems that are based on abstraction and invariants [46, 60] could be a basis for verifying infinite families of infinite systems.

4.7 Summary

We introduced CTL-Live: a fragment of CTL whose model checking problem is reducible to FOL validity checking. CTL-Live includes CTL connectives that are used to express liveness properties (e.g., **AF**, **AU**, etc.). Our reduction shows that FOL deductive techniques are sufficient for model checking CTL-Live formulas, without the need for iteration, abstraction, or induction. Our theory provides the basis for using first-order reasoners directly for model checking CTL-Live properties of abstract and infinite Kripke structures expressed symbolically in FOL. By avoiding iteration, the tool can reuse its internal deductions to increase productivity. The rapid improvements in the efficiency of SMT solvers, FOL automated theorem proving, etc. have a direct effect on the practical application of our results.

The key insight in our approach is to use the implicit higher-order quantifier in the definition of validity to require that all initial states of a Kripke structure are within all the sets of states that satisfy an overapproximation of a CTL-Live temporal operator, and thereby, reducing model checking to validity in FOL. Validity checking for FOL is r.e.; as a result, this reduction ensures that a proof can be automatically generated when a CTL-Live formula is satisfied by a model.

An implicit result of our work is that some reachability queries can be verified and expressed as a validity argument in FOL even though the reachability relation itself is not expressible in FOL.

We proved that CTL-Live is maximal in the sense that if any other CTL connective is added, non-FOL reasoning techniques would be required and the model checking problem becomes harder than an r.e. problem.

We also examined the decidability of CTL-Live model checking for SKSs. We showed how to derive decidability results for CTL-Live model checking by focusing on decidable fragments of FOL, namely the **AE** fragment and some dialects of description logics.

Chapter 5

Model Checking in FOL: Practice

In Chapter 4, we introduced CTL-Live, a fragment of CTL whose model checking problem is reducible to FOL validity checking (Theorem 5 on page 39). CTL-Live consists of operators that are commonly used to describe liveness properties (e.g., **AF**, **AU**). We also showed that CTL-Live is maximal with respect to FOL in the sense that the model checking problem of CTL operators that are not within CTL-Live (e.g., invariants) are not reducible to FOL validity checking (Theorem 6 on page 43).

Theorem 5 on page 39 creates the possibility of the following practical use: model the system as an SKS, add automatically generated constraints based on the CTL-Live property, and give the problem to an SMT solver to solve by itself. If the property is valid, theoretically with enough resources, the SMT solver can complete the analysis because FOL is recursively enumerable. This method is elegant in its simplicity: no iteration or abstraction is required, and no user intervention is needed to determine reachability constraints (inductive invariants).

An SMT solver differs from a general-purpose FOL satisfiability checker in one major way: if a built-in type such as `Integer` is used in a formula, the SMT solver considers only the "standard" interpretation for that type and the defined operations over it. SMT-LIB is a standard notation that state-of-the-art SMT solvers accept as input [10]. A specification of a problem in SMT-LIB consists of four parts: 1) declaration of user-defined types, 2) declaration of functional symbols used in the model, 3) definitions that are used to simplify the model, and 4) a set of constraints, where each constraint is a formula. SMT-LIB does not distinguish between terms and formulae. A formula is a term of type `Bool`. To ease the parsing of SMT-LIB specifications by SMT solvers, each SMT-LIB specification is a sequence of S-expressions.

We evaluate the practical application of our theory through a set of case studies. In this chapter, we focus on the following research questions:

1. Will this method work in practice? In other words, are state-of-the-art SMT solvers efficient enough to analyze properties of the dynamic behaviour of infinite state systems?
2. How efficient is the model checking of an infinite state model in comparison to the analysis of a finite version of the same model?
3. Are there modelling techniques that facilitate the use of SMT solvers for model checking?

We have chosen a varied collection of four case studies drawn from different sources. Each has an infinite state space through the use of integers or more complex data types. Our results show that our approach does work in practice and can verify liveness properties of infinite state models quickly using the SMT solvers Z3 [29] and CVC4 [8]. In fact, for some of the case studies, we show that the verification of the infinite state system completes more quickly than the verification of the same problem with limited ranges in finite solvers such as the Alloy Analyzer [44] and Cadence SMV [49].

Most of the content of this chapter has been published [58]. This chapter is organized as follows: Section 5.1 describes the case studies and the chain of tools that we use to verify our case studies. In Section 5.2, we discuss modelling choices that have an effect on the performance of the tools we use.

5.1 Case Studies

The approach that we use to implement our method is described in Figure 5.1 on the next page. A model is created in FOL using a tool that we developed called Avestan. Our current version of Avestan is a complete reengineering of our earlier tool [54] (also called Avestan), which was a language and tool to support the creation of models in SMT-LIB. It was strongly based on Alloy [43], but the tool translated the model into an SMT-LIB specification. Our new tool is implemented in Python [2] and uses Python as both the object and meta-language for expressing models in FOL. It produces specifications in SMT-LIB for analysis by an SMT solver.

We implemented the function `CTLive2FOL()` (Figure 4.4 on page 35) in Avestan to create the constraints needed for the verification of a CTL-Live property. Using Avestan,

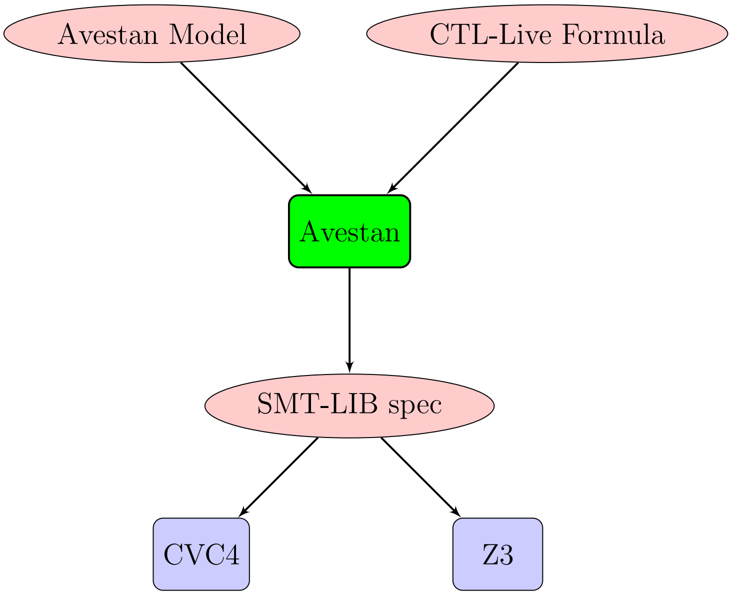


Figure 5.1: Overview of our method

we transform a model plus these constraints into an SMT-LIB specification and check the validity problem as a satisfiability problem using both CVC4 (version 1.3) and Z3 (version 4.3.1). All our experiments were run on an Intel®Core^{TM}i7-3667U machine running Ubuntu 12.04 64-bit with up to 7.5GB of user memory. To analyze the case studies, we used the solvers in their default mode, without any flags or a customized configuration. The SMT-LIB specifications of the case studies and other models developed for this chapter are available on-line[1].

5.1.1 Case Study 1: Leader Election Protocol

The leader election model is a protocol to "elect" a process as the leader amongst a finite set of processes that form a ring [21]. A finite instance of it was previously verified by Jackson using the Alloy Analyzer [44]. In the leader election model, each process in the ring can only communicate with its successor and predecessor, and there is no centralized controller. Each process has a unique identifier (ID) and a value to represent who this process thinks is the leader of the ring (`my_lead`). The goal of the protocol is that every

[1] https://cs.uwaterloo.ca/~nday/models/fse14/vakili-day-fse14-models.zip

process (including the leader) will eventually recognize that the process with the greatest ID is the leader. We modelled a synchronous version of this protocol: at each moment, every process passes to its predecessor its value for `my_lead` and receives from its successor the successor's value for `my_lead`. If the received value is greater than the process's current value of `my_lead`, the process updates its value with the received one, otherwise, it is left unchanged. In the initial state, the value passed by a process is its own ID.

We used unbounded integers to model IDs and time. For each process, we declared a functional symbol `my_lead` of type `Int -> Int`. We have a fixed number of processes. The ring topology is enforced by an ordering on the processes, where the successor of the last process is the 0^{th} process and for any other processes such as i, the successor is $i + 1$.

The properties we verified are that every process will eventually recognize the leader:

$$\mathbf{AF}\ \texttt{(my\_lead\_i = lead\_id)}$$

where `lead_id` is the largest ID amongst the current processes, `my_lead_i` the value of `my_lead` for the $\texttt{i}^{\text{th}}$ process. Thus, for i processes, we have i properties, which we conjuncted together and checked. In this model, the set `Int`, which is used to represent time, is also the state space of this system. The following table shows the performance of Z3 for different numbers of processes:

# of Processes	12	14	16	18	20
Time	8.48s	44.38s	3m24.64s	50m44.09s	2h37m11.69s

CVC4 with even 2 processes could not finish the verification.

When we modelled this problem with an unbounded number of processes, the verification in either SMT solver does not complete. Verification for an unbounded number of processes would likely require user intervention to deduce an invariant that would help the SMT solver verify the problem.

We also modelled this synchronous version of the algorithm in Alloy [44]. To verify the liveness properties using Alloy, we needed to finitize all sets, including time. We set the bounds on time and IDs to be the number of processes. Figure 5.2 on the next page, which is in logarithmic scale to increase the readability of the plot, compares the performance of Z3 on models where there are no bounds on time and IDs to the performance of the Alloy Analyzer (version 4.2 using minisat) where time and IDs are bounded. In the Alloy models, the properties were conjuncted together and verified (as in Z3). As this figure shows, our approach to the verification of this protocol with an infinite state space is much faster than Alloy where every set needs to be finitized.

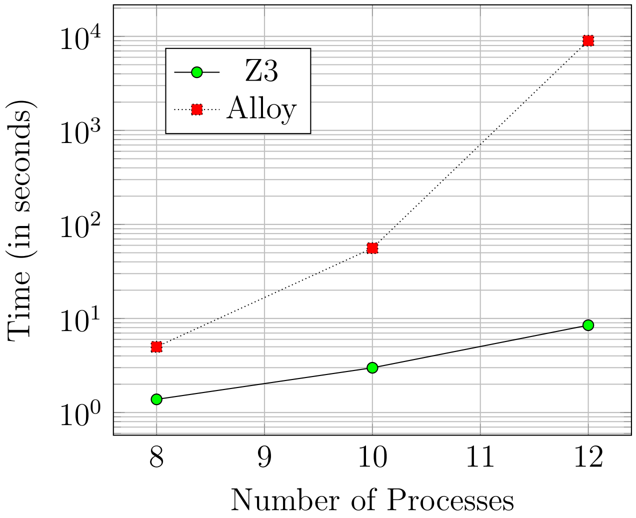


Figure 5.2: Leader election model: Z3 vs Alloy

While our verification does require a bound on the number of processes, it is significant that it does not require a bound on time. When we finitize time (as in the Alloy model), we are doing bounded model checking (BMC) [14]. When using BMC to verify a liveness property, spurious counterexamples can result because the bound is insufficient to conclude liveness. In general, computing a sufficient bound to get a reliable result is hard, and in some infinite cases it is impossible. In our SMT-LIB models, we use unbounded integers to represent time. Since SMT solvers check satisfiability with respect to standard interpretations and this interpretation for integers guaranties that `Int` has an infinite set of elements, our technique does not produce spurious counterexamples. For this case study, we generated 559 SMT-LIB specifications. These specifications differ in the number of processes that the process of interest. All of them are available on-line. As an example, we have included one of them in Appendix A.1.

5.1.2 Case Study 2: Bakery Algorithm

The bakery algorithm ensures mutual exclusion between two processes that run concurrently and asynchronously [17]. Bultan, Gerber, and Pugh verified that in this algorithm the two processes cannot get into their critical sections at the same time [17]. Their method is an iterative approach that uses a Presburger arithmetic solver.

In the bakery algorithm model, the state of a process is determined by its control state value and a ticket. The value of a control state is either `Thinking`, `Waiting`, or `Critical`. A ticket is a non-negative unbounded integer. Since we have two processes, the state space of this system, S, is the following:

$$S = \{\ \texttt{T, W, C}\ \} \times\ \texttt{Int}\ \times \{\ \texttt{T, W, C}\ \} \times\ \texttt{Int}$$

We modelled the set $\{\texttt{T, W, C}\}$ as an uninterpreted type, named `ControlState`, where `T`, `W`, and `C` are three distinct constants of type `ControlState`. The following is a fragment of the SMT-LIB specification that models `ControlState` ensuring that each value is distinct:

```
(declare-sort ControlState 0)
(declare-fun T () ControlState)
(declare-fun W () ControlState)
(declare-fun C () ControlState)
(assert (not (= T W)))
(assert (not (= T C)))
(assert (not (= W C)))
```

Besides comparing the value of the tickets, this algorithm also manipulates the value of tickets using the addition operation on integers; as a result, an uninterpreted type with a total ordering would not be sufficient to express this model. Each transition in our model is defined as a functional symbol of type $S \times S$ `->` `Bool`. By combining these transitions, we modelled the transition relation.

For this case study, we verified that any process, e.g., process 1, that is waiting to get into its critical section, will eventually succeed:

$$\mathbf{AG}\left(\texttt{c1 = W} \Rightarrow \mathbf{AF}\,\texttt{c1 = C}\right) \tag{5.1}$$

This is an invariant property, therefore to verify this property, we needed to show that every reachable state satisfies $\texttt{c1 = W} \Rightarrow \mathbf{AF}\,\texttt{c1 = C}$. **AG** is not part of CTL-Live, so we cannot ask the SMT solver to prove this property directly. Instead, we created a more general property that implies the formula of Equation 5.1: we proved that the set of *all* states, which includes the reachable states, satisfies the following property:

$$\texttt{c1 = W} \Rightarrow \mathbf{AF}\,(\texttt{c1 = C} \vee \text{dead\_end})$$

where `dead_end` is true of a state iff that state does not have any next state. This model has a non-total transition relation, however, according to the semantics of CTL, correct

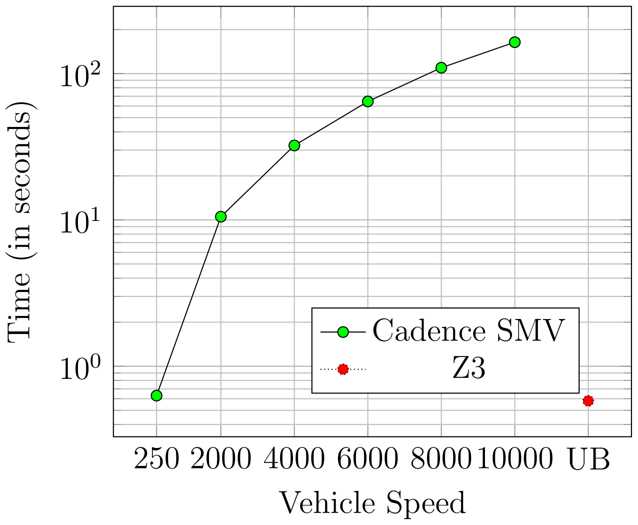


Figure 5.3: Collision Avoidance model in Cadence SMV (UB = UnBounded)

paths of the model must be infinite and only those should be considered when checking a property. Rather than making the transition relation total, we introduced the idea of a "dead-end" state: an state from which there are no next states and thus it satisfies a CTL formula that has a universal path quantifier.

We stated this property by making the set of initial states be the set of all states. This revised property is part of CTL-Live. Z3 verified this property in 0.08 seconds and CVC4 in 8.64 seconds. The generated SMT-LIB specification for this case study is provided in Appendix A.2.

Another algorithm studied by Bultan, Gerber, and Pugh is the ticket mutual exclusion algorithm [5, 17]. We tried to verify an invariant property similar to Equation 5.1 on the preceding page for this model using a similar technique to the Bakery algorithm; however, neither SMT solver terminates within a threshold of 3 hours. It is likely that this property of this model is only satisfied within the reachable set of states and therefore it does not hold for the entire set of states.

5.1.3 Case Study 3: Collision Avoidance State-Flow Model

Our third case study is a Stateflow model of a collision avoidance feature used in a modern vehicle [32]. It was previously used with other feature models to check for feature interactions using Cadence SMV.

This case study has control state complexity in a hierarchical, non-concurrent state-transition model with 9 basic states. However, there are two variables manipulated by the transitions of the model: speed and threshold, which determine when collision avoidance needs to be engaged. These variables are used in the triggers of transitions and thus, affect the control logic of the system and therefore are not removed by standard cone of influence reductions. In our model, the speed of a vehicle is modelled as an unbounded integer, and threshold is a constant positive integer. We verified that every basic state is reachable without a bound on speed and threshold. This property is a conjunction of 9 **EF** formulae. Z3 verifies all these properties together in 0.58 seconds. CVC4 terminates in 0.89 seconds having `UNKNOWN` as output. The `UNKNOWN` means the solver cannot verify nor refute the property.

Because we had access to the original models, we can compare our results to using Cadence SMV to analyze the Stateflow models for different finite bounds on speed and threshold. Figure 5.3 on the preceding page presents these results. As this figure shows, the performance of Cadence SMV degrades as the size of speed and threshold is increased. The SMT-LIB specification of this case study is provide in Appendix A.3.

5.1.4 Case Study 4: File System

Our last case study is a file system that was originally modelled in Z [62]. Woodcock and Davies use natural deduction to prove properties manually about this model.

The state of the file system is represented as a partial function from `Keys` to `Data` named `content`. There are three operations that change the state of the file system: `add`ing a new entry, `delet`ing an existing entry and `writ`ing new data to an existing key.

The major difference between the file system model and our other case studies is in its state space: each state is a function whereas in the other case studies, a state is a tuple that includes an infinite element. Since quantification over functions is not allowed in FOL, we cannot directly use our technique to model check a CTL-Live property of this model.

Borrowing a technique used in Alloy models [44], in our model, we explicitly introduced the state space as a new uninterpreted set `State` and declared `content` as follows:

```
content:  State × Key -> Data
```

where content(s, k) = d is interpreted as the content of the file system at state s for the key k is d. To model the fact that content is a *partial* function from Key to Data, we declare a constant NULL of type Data: the value of content(s, k) being equal to NULL means that the content of the file system at state s for the key k is empty. In Alloy, this technique manifests itself in the use of a new set "State" to encapsulate the elements of the state.

The disadvantage of explicitly introducing the set State is that it is uninterpreted, and it may result in spurious counterexamples. For example, the following property is not entailed by this model:

$$\texttt{content(s, k)} \neq \texttt{NULL} \Rightarrow \exists s' \bullet \ \texttt{delete(k, s, }s'\texttt{)} \tag{5.2}$$

This property states that if at state s the content of key k is not empty, then we can delete k from it and go to some state s'. The spurious counterexample for this property is a single state with non-empty content. We need to ensure that interpretations that do not include enough states are eliminated from the analysis. To eliminate these spurious counterexamples, we need to "interpret" State by adding some axioms to the model. These axioms are called *generator axioms* [44]. For our file system model where only a performed operation can change the state, a set of standard generator axioms exist: for every operation we needed to add a formula stating that if an operation OP is applicable on a state s1, then there exists another state such as s2 that is the result of performing OP on s1; in other words, we needed to state that all the operations are total. For example the generator axiom for delete is same as the formula in Equation 5.2 except s and k are bounded by universal quantifiers.

We verified a bisimilarity property that the operation write can be simulated by some combination of add and delete for all possible states of the file system. For this purpose, we remove the write operation from the model, and we assume that some state s2 is the result of writing something to the file system at some state s1; then, we check in the new model, which does not have the write operation, if s2 is reachable from s1:

$$\big(\texttt{write(k, d, s1, s2)} \ \wedge \ \texttt{s = s1}\ \big) \Rightarrow \ \mathbf{EF}\texttt{s = s2}$$

Z3 verified this property in 0.15 seconds and CVC4 in 0.69 seconds. The SMT-LIB specification of this case study is provided in Appendix A.4.

Our case studies show that our method is practical for a variety of different examples. In all our models, we were able to leave some element of the model state unbounded and complete verification of a property in CTL-Live. We used unbounded integers, user-declared sorts, and a partial function as part of the state. Table 5.1 on the following page

Table 5.1: Run time of Z3 and CVC4 for each case study in seconds (DNV: Did Not Verify)

Case study	Z3	CVC4
Leader election, 12 processes	8.48	DNV
Leader election, 14 processes	44.38	DNV
Leader election, 16 processes	204.64	DNV
Leader election, 18 processes	3044.09	DNV
Leader election, 20 processes	9431.69	DNV
Bakery algorithm	0.08	8.64
Collision avoidance	0.58	DNV
File system	0.15	0.69

summarizes the run times of Z3 and CVC4 for all the case studies. Z3 clearly performs better than CVC4 for the data types used in our case studies.

5.2 Modelling for Better Performance

In this section, we provide some insights about factors that can be used by a modeller to develop models that are more efficient to analyze in SMT solvers.

First, we consider the trade-off in the number of variables and the number of constraints. For the leader election case study, we have two choices for expressing the ID of the leader:

1. Declare a new constant and assert that this constant is equal to the ID of some process and that it is greater or equal to all the IDs; or

2. Use the if-then-else construct in SMT-LIB and compare all the IDs with each other to determine the largest.

For example, if we have two processes, the maximum is defined using the following constraint based on the first option:

$$(\texttt{max} = \texttt{id1} \vee \texttt{max} = \texttt{id2}) \wedge (\texttt{max} >= \texttt{id1}) \wedge (\texttt{max} >= \texttt{id2})$$

and in the second option, maximum is the following term:

$$\texttt{if (id1} > \texttt{id2) then id1 else id2}$$

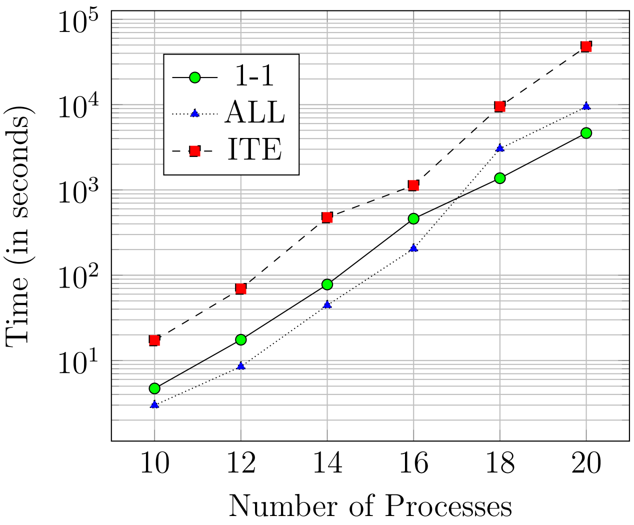


Figure 5.4: Z3 on different models for the leader election problem

The first approach adds 2 × n constraints and a new variable, where n is the number of processes. The second approach does not introduce any new constraints or variables, but the term that represents the greatest ID is complex. Plot ITE of Figure 5.2 shows the sum of the times for verifying n properties using the ITE modelling approach, where n is the number of processes. Plot 1-1 shows the same problem using the first modelling approach. Clearly, the approach of creating a single more complicated constraint performed less efficiently that having a number of simple constraints with more variables in this case.

In addition, we can compare verifying n properties together (as a conjunction of constraints) to verifying each property individually and summing the total time of verification. In Figure 5.2, plot ALL is the result of verifying the conjunction of the properties. For larger numbers, ALL performs more poorly than plot 1-1, which is the sum of the times to verify each property individually[2]. This result again supports the hypothesis that simple constraints are better for SMT solvers than complex ones.

Next, we consider the effect of the use of quantifiers in these problems. Since we use integers to model time in the leader election case study, rather than using our CTL-Live

[2] Since the Alloy models were analyzed using the ALL approach, in Subsection 5.1.1, we have reported the results of the ALL approach using Z3 even though the 1-1 approach performs better.

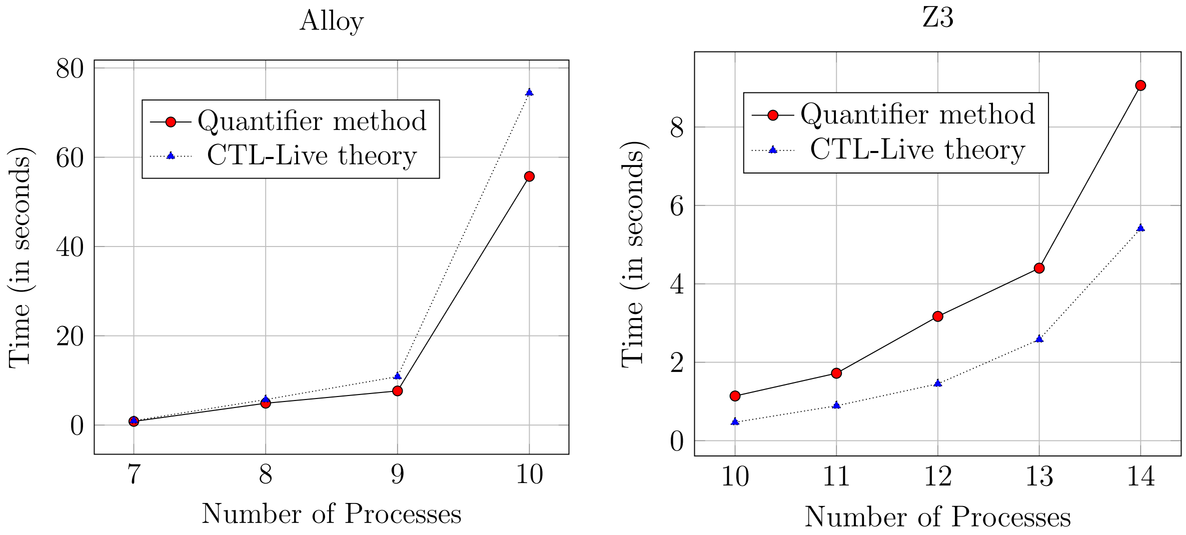


Figure 5.5: Alloy and Z3 with different approaches for the leader election case study

`CTLive2FOL()`, the eventuality property can be expressed using an existential quantifier as in:

$$\exists \texttt{t:Int} \bullet \;\; \texttt{t > 0} \;\wedge\; \texttt{my\_lead\_i(t) = leader\_id}$$

Since Alloy's input language is as expressive as FOL, we can use our `CTLive2FOL()` function to model check a CTL-Live property using the Alloy Analyzer. We set the size of all the sets in the Alloy model equal to the number of processes. Figure 5.5 presents the result of trying these two approaches both for Alloy and Z3. As this figure shows, the Alloy Analyzer is on average 1.27X faster when using the quantifier method to express the properties compared to our CTL-Live theory in Alloy. On the other hand, Z3 on the SMT-LIB models that used our CTL-Live theory was on average 1.98X faster than using the quantifier method on the model. Our conclusion from this observation is that the modelling methods also depend on the analysis tool that is used. However, Z3 using the CTL-Live theory with unbounded integers was the most efficient method by far.

5.3 Summary

In this chapter, we have shown that it is practical to use SMT solvers, in particular Z3, to verify CTL-Live properties of infinite state models without the need for iteration, abstraction, or human intervention. The system is modelled as a (potentially infinite) symbolic

Kripke structure in FOL, a set of FOL constraints is automatically generated based on the property, and the problem is given to an SMT solver to solve by itself. Because FOL is recursively enumerable, with enough resources, the analysis will terminate if the property is valid. We have also shown that the analysis of infinite state systems using an SMT solver can be more efficient than the analysis of a finite version of the model. SMT solvers use deductive analysis (rather than just state space search) and therefore can take advantage of structures found in abstract models. We discussed modelling techniques that facilitate efficient model checking using SMT solvers.

Part III

Model Checking in FOLTC

Chapter 6

Model Checking in FOLTC: Theory

In Chapter 4, we introduced CTL-Live: a fragment of CTL whose model checking problem is reducible to FOL validity checking. According to Theorem 6 on page 43, CTL-Live is the largest fragment of CTL whose model checking problem is reducible to FOL validity checking. This implies that in order to formulate the model checking problem of CTL connectives that are not included in CTL-Live (**AG** and **EG**) as validity checking, we should consider a more expressive logic than FOL or (and) restrict the class of systems under study.

In this chapter, we focus on FOL plus transitive closure (FOLTC). As our first step, FOLTC allows us to formalize the model checking problem for **AG**. Our major insight is that transitive closure contains reachability information and **AG** is about reachable states. Next, we focus on finite Kripke structures. The finiteness restriction allows us to encode all CTLFC formulas in FOLTC.

The rest of this chapter is organized as follows: in Section 6.1, we present CTL\EG, which includes all CTL connectives except **EG**. In Section 6.2, we discuss why FOLTC is sufficient for encoding **EG**. In Section 6.3, we show how the finiteness assumption allows us to express all CTLFC formula in FOLTC. This reduction is only done when we have one fairness constraint.

6.1 Model Checking CTL\\EG

The grammar of CTL\EG is presented in Figure 6.1 on the next page. CTL\EG is same as CTL-Live except it has **EF** in its propositional part.

Temporal part		
φ	$::=$	$\pi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$
	$::=$	$\mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi$
	$::=$	$\varphi_1 \mathbf{EU} \varphi_2 \mid \varphi_1 \mathbf{AU} \varphi_2$
Propositional part + **EF**		
π	$::=$	$P \mid \neg\pi \mid \pi_1 \vee \pi_2 \mid \mathbf{EF}\ \pi$
	where P is a labelling predicate.	

Figure 6.1: CTL\EG

```
CTLEG2FOLTC($\varphi$):
     case $\varphi$ of
 1.  $P$                        =>  $\{\}$          where $P$ is a labelling predicate
 2.  $\neg\varphi_1$            =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \neg\lceil\varphi_1\rceil(s)\}$ ∪ CTLEG2FOLTC($\varphi_1$)
 3.  $\varphi_1 \vee \varphi_2$ =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \lceil\varphi_1\rceil(s) \vee \lceil\varphi_2\rceil(s)\}$ ∪
                                      CTLEG2FOLTC($\varphi_1$) ∪ CTLEG2FOLTC($\varphi_2$)
 4.  $\varphi_1 \wedge \varphi_2$ => $\{\forall s \bullet\ \lceil\varphi\rceil(s) \Leftrightarrow \lceil\varphi_1\rceil(s) \wedge \lceil\varphi_2\rceil(s)\}$ ∪
                                      CTLEG2FOLTC($\varphi_1$) ∪ CTLEG2FOLTC($\varphi_2$)
 5.  $\mathbf{EX}\varphi_1$     =>  $\{\forall s \bullet\ \big(\exists s' \bullet\ N(s,s') \wedge \lceil\varphi_1\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ ∪ CTLEG2FOLTC($\varphi_1$)
 6.  $\mathbf{AX}\varphi_1$     =>  $\{\forall s \bullet\ \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi_1\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ ∪ CTLEG2FOLTC($\varphi_1$)
 7.  $\mathbf{EF}\varphi_1$     =>  $\{\forall s \bullet\ \lceil\varphi\rceil(s)\ \Leftrightarrow\ \lceil\varphi_1\rceil(s) \vee \big(\exists s' \bullet\ N^+(s,s') \wedge \lceil\varphi_1\rceil(s')\big)\}$ ∪ CTLEG2FOLTC($\varphi_1$)
 8.  $\mathbf{AF}\varphi_1$     =>  $\{\lceil\varphi_1\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ ∪
                                      CTLEG2FOLTC($\varphi_1$)
 9.  $\varphi_1\mathbf{EU}\varphi_2$ => $\{\lceil\varphi_2\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \lceil\varphi_1\rceil(s) \wedge \big(\exists s' \bullet\ N(s,s') \wedge \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ ∪
                                      CTLEG2FOLTC($\varphi_1$) ∪ CTLEG2FOLTC($\varphi_2$)
10.  $\varphi_1\mathbf{AU}\varphi_2$ => $\{\lceil\varphi_2\rceil \subseteq \lceil\varphi\rceil,\ \forall s \bullet\ \lceil\varphi_1\rceil(s) \wedge \big(\forall s' \bullet\ N(s,s') \Rightarrow \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)\}$ ∪
                                      CTLEG2FOLTC($\varphi_1$) ∪ CTLEG2FOLTC($\varphi_2$)
```

Figure 6.2: Definition of CTLEG2FOLTC(φ)

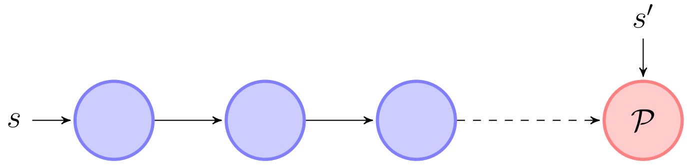


Figure 6.3: State s satisfies **EF** $\mathcal{P}$

The propositional part of CTL-Live has an important property that its temporal part does not have: Lemma 2 on page 36. According to this lemma, the constraints introduced by the reduction function `CTLive2FOL()` for a propositional formula fully capture the semantics of those formulas, whereas the constraints generated for the temporal part ensure an over-approximation of the set of states that satisfy the given formula. Now, by using FOLTC, we are capable of fully capturing the semantics of **EF**. As a result, the reduction of CTL\EG to FOLTC, `CTLEG2FOLTC()`, is the same as `CTLive2FOL()` except that the formula that we introduce for **EF** is different. The function `CTLEG2FOLTC()` is presented in Figure 6.2 on the preceding page.

The intuition behind the formula that encodes **EF** in FOLTC is as follows: For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ and a labelling predicate $\mathcal{P} \in \mathbb{P}$ we know that a state s satisfying **EF** $\mathcal{P}$ means that it is possible to reach a state s' that satisfies $\mathcal{P}$ from s in a finite number of steps. This is depicted in Figure 6.3. From the definition of transitive closure, we know that two states x and y belong to $\mathcal{N}^+$ iff y is reachable from x by taking a finite number of steps. These observations lead to the following conclusion:

$$s \in [\mathbf{EF}\ \mathcal{P}] \quad \text{iff} \quad s \in \mathcal{P} \ \vee \ \exists s' \in \mathcal{P} \bullet \ (s, s') \in \mathcal{N}^+$$

s satisfies **EF** $\mathcal{P}$, $s \in [\mathbf{EF}\ \mathcal{P}]$, iff either s satisfies $\mathcal{P}$ itself, $s \in \mathcal{P}$, or it can reach a state s' that satisfies $\mathcal{P}$, $\exists s' \in \mathcal{P} \bullet \ (s, s') \in \mathcal{N}^+$. This means that **EF** can be formalized in a logic that has transitive closure, such as FOLTC. The lesson here is that model checking is about reachability of states from each other. Transitive closure contains the reachability information, which can be used to formalize temporal logic properties.

Lemma 7. Let Σ be an SKS, π a CTL\EG formula derived from the propositional + **EF** part of Figure 6.1 on the preceding page. For every Kripke structure $\mathcal{K}$ such that $\mathcal{K} \Vdash \Sigma \cup$ `CTLEG2FOLTC(`π`)` the following holds:

$$[\pi]_{\mathcal{K}} = \lceil \pi \rceil^{\mathcal{K}}$$

Proof. Proof by structural induction on π. Due to Lemma 2 on page 36, we only need to cover the induction step for **EF** π. By assuming $[\pi]_{\mathcal{K}} = \lceil \pi \rceil^{\mathcal{K}}$, we need to prove the following:

$$s \in [\mathbf{EF}\ \pi]_{\mathcal{K}} \quad \text{iff} \quad s \in \lceil \mathbf{EF}\ \pi \rceil^{\mathcal{K}}$$

According to the semantics of CTL, $s \in [\mathbf{EF}\ \pi]_{\mathcal{K}}$ iff there is a finite path from s to a state s' that satisfies π. Constraint 7 of Figure 6.1 on page 70 expresses this fact. □

Since **AG** φ is equivalent to $\neg(\mathbf{EF}\ \neg\varphi)$, and $\neg$ is included in the propositional part+**EF** of Figure 6.1 on page 70, CTL\EG includes **AG** as well.

Theorem 10. (CTL\\EG model checking as FOLTC validity checking) Let Σ be an SKS, and φ a CTL\EG formula from Figure 6.1 on page 70. We have the following property:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \Sigma \cup \texttt{CTLEG2FOLTC}(\varphi) \vDash S_0 \subseteq \lceil \varphi \rceil$$

Proof. Proof of this theorem is same as Theorem 5 on page 39, except every use of Lemma 2 on page 36 is replaced with Lemma 7 on the preceding page. □

6.2 EG and FOLTC

FOLTC "seems" to be insufficient for expressing the model checking of **EG**. We do not have a proof for this but the intuition behind this is that a state satisfies an **EG** P formula iff it is on an infinite path such that every state along that path satisfies P. Using transitive closure, we can only express if two states are reachable from each other using a "finite" number of steps. This is not enough for expressing infinite paths. We call the fragment of CTL whose model checking is definitely reducible to FOLTC validity checking CTL\EG. We do not have a maximality result similar to Theorem 6 on page 43 for CTL\EG. Immerman and Vardi also hypothesis that it is not possible to encode all **EG** in FOLTC [42]. The intuition is that an infinite system can have an infinite path such that no state along the path has appeared before. It seems like these kind of paths cannot be formalized in FOLTC.

6.3 Reducing CTLFC to FOLTC

In order to express model checking of **EG** in FOLTC, we assume the Kripke structures under study are finite. The finiteness restriction ensures that the only way to have an infinite

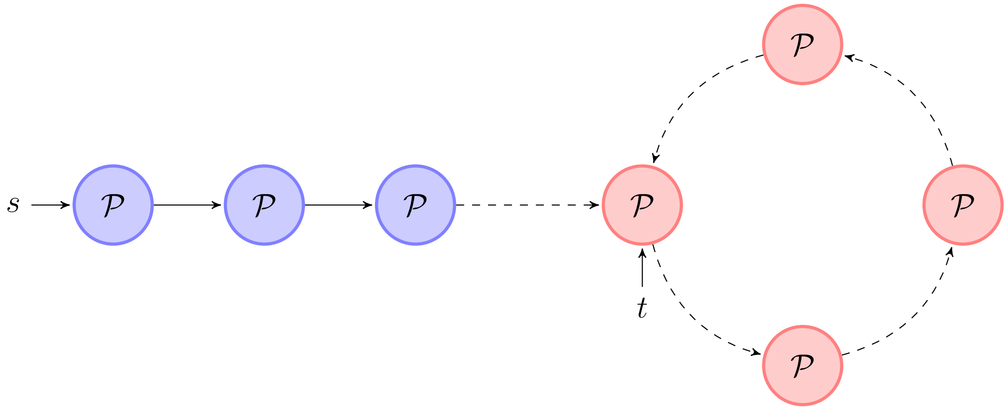


Figure 6.4: State s satisfies **EG** $\mathcal{P}$

path is through repetition of states. This implies that every path has a finite representation, which is expressible using transitive closure; moreover, this finiteness assumption allows us to express the model checking problem of all CTLFC formulas in FOLTC. Our finiteness restriction does not require the user to provide the size of Kripke structures under study. The size can be an unknown number; as long as it is finite, our reduction works. Next, we assume that the SKS under-study only represent finite Kripke structures. This implies that the only way to have an infinite path is through repetition of states. Here is how we encode **EG** in FOLTC assuming that $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ is a finite Kripke structure and $\mathcal{P} \in \mathbb{P}$ a labelling predicate: A state s satisfies **EG** $\mathcal{P}$ iff there exists an infinite computation path starting from s such that all the states along this path satisfy $\mathcal{P}$. Since the state space is finite, the only way to have an infinite computation path is through *repetition* of states; in particular, state s satisfies **EG** $\mathcal{P}$ iff it is on a *lasso-shape* path [13], Figure 6.4.

As depicted in Figure 6.4, a lasso has two parts: a cycle and a tail. If a state s satisfies **EG** $\mathcal{P}$, it should either belong to the cycle section:

$$(s, s) \in \left(\mathcal{P} \lhd \mathcal{N}\right)^+$$

or it should belong to the tail section:

$$\exists t \bullet \; (s, t) \in \left(\mathcal{P} \lhd \mathcal{N}\right)^+ \; \wedge \; (t, t) \in \left(\mathcal{P} \lhd \mathcal{N}\right)^+$$

In these constraints, $\lhd$ is the domain restriction operator of Table 2.1 on page 10, and $^+$ is the transitive closure operator. Putting these together, we have the following conclusion:

$$s \in [\mathbf{EF}\ \mathcal{P}] \quad \text{iff} \quad \texttt{Let}\ \mathcal{M} := \mathcal{P} \lhd \mathcal{N}\ \texttt{in}\ (s, s) \in \mathcal{M}^+ \ \vee\ \exists t \bullet \; (s, t) \in \mathcal{M}^+ \wedge (t, t) \in \mathcal{M}^+$$

$$\begin{array}{llll}
1. & \texttt{FC2TC(}P\texttt{, }FC\texttt{)}(s) & := & P(s) \\
2. & \texttt{FC2TC(}\neg\varphi\texttt{, }FC\texttt{)}(s) & := & \neg\,\texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s) \\
3. & \texttt{FC2TC(}\varphi \vee \psi\texttt{, }FC\texttt{)}(s) & := & \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s) \ \vee\ \texttt{FC2TC(}\psi\texttt{, }FC\texttt{)}(s) \\
4. & \texttt{FC2TC(}\mathbf{EX}\varphi\texttt{, }FC\texttt{)}(s) & := & \exists t \bullet\ N(s,t) \wedge \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(t) \\
5. & \texttt{FC2TC(}\varphi\mathbf{EU}\psi\texttt{, }FC\texttt{)}(s) & := & \texttt{Let } M(x,y) \texttt{ := } \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(x) \ \wedge\ N(x,y) \texttt{ in} \\
 & & & \quad \texttt{FC2TC(}\psi\texttt{, }FC\texttt{)}(s) \vee \\
 & & & \quad \big(\exists t \bullet\ M^{+}(s,t) \wedge \texttt{FC2TC(}\psi\texttt{, }FC\texttt{)}(t)\big) \\
6. & \texttt{FC2TC(}\mathbf{E_C G}\varphi\texttt{, }FC\texttt{)}(s) & := & \texttt{Let } M(x,y) \texttt{ := } \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(x) \ \wedge\ N(x,y) \texttt{ in} \\
 & & & \quad \exists \bullet\ FC(t) \ \wedge\ M^{+}(t,t) \wedge \big(s = t \ \vee\ M^{+}(s,t)\big)
\end{array}$$

Figure 6.5: Definition of FC2TC(P, FC)

The finiteness assumption allows us to encode CTLFC in FOLTC. CTLFC is more expressive than CTL. We present our encoding when we have a single fairness constraint. We define a translation function FC2TC that recursively goes through a CTLFC formula and generates a relational symbol plus an axiom for each sub-formula. Each generated axiom encodes the semantics of the corresponding sub-formula. According the equalities of Figure 2.3.2, we only need to present the encoding for ¬, ∨, **EX**, **EU**, and $\mathbf{E_C G}$. The translation function FC2TC is presented in Figure 6.5.

Theorem 11. (Correctness of FC2TC function) Let $\mathscr{B}$ be a Kripke base, Σ a symbolic Kripke structure over $\mathscr{B}$, $FC/1$ a relational symbol that represents a fairness constraint, and φ a CTLFC formula. For every Kripke structure $\mathcal{K}$ that satisfies Σ, we have:

$$[\varphi]_{\mathcal{K}} = \big(\texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}\big)^{\mathcal{K}}$$

In other words, FC2TC(φ, FC) represents the set of states of $\mathcal{K}$ that satisfy φ, $[\varphi]_{\mathcal{K}}$.

Proof. Proof by structural induction on φ:

- **Base case:** let $\varphi = P$ where P is a labelling predicate. For every s, we have:

$$\begin{array}{llll}
s \in [P]_{\mathcal{K}} & \text{iff} & s \in P^{\mathcal{K}} & \textit{by semantics of CTL} \\
 & \text{iff} & s \in \big(\texttt{FC2TC(}P\texttt{, }FC\texttt{)}\big)^{\mathcal{K}} & \textit{by Line 1 in FC2TC}
\end{array}$$

  therefore $[P]_{\mathcal{K}} = \big(\texttt{FC2TC(}P\texttt{, }FC\texttt{)}\big)^{\mathcal{K}}$.

- **Induction step:** according to the definition of FC2TC five cases are distinguished having $[\varphi_1]_{\mathcal{K}} = \big(\texttt{FC2TC(}\varphi_1\texttt{, }FC\texttt{)}\big)^{\mathcal{K}}$ and $[\varphi_2]_{\mathcal{K}} = \big(\texttt{FC2TC(}\varphi_2\texttt{, }FC\texttt{)}\big)^{\mathcal{K}}$ as induction hypotheses:

1. Let $\varphi = \neg\varphi_1$. For every s, we have:

   $$\begin{array}{llll} s \in [\neg\varphi_1]_{\mathcal{K}} & \text{iff} & s \notin [\varphi_1]_{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} & s \notin \big(\texttt{FC2TC(}\varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by induction hypothesis} \\ & \text{iff} & s \in \big(\texttt{FC2TC(}\neg\varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by Line 2 of } \texttt{FC2TC} \end{array}$$

   therefore $[\neg\varphi_1]_{\mathcal{K}} = \big(\texttt{FC2TC(}\neg\varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}}$.

2. Let $\varphi = \varphi_1 \vee \varphi_2$. For every s, we have:

   $$\begin{array}{llll} s \in [\varphi_1 \vee \varphi_2]_{\mathcal{K}} & \text{iff} & s \in [\varphi_1]_{\mathcal{K}} \vee s \in [\varphi_2]_{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} & s \in \big(\texttt{FC2TC(}\varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} \vee & \\ & & s \in \big(\texttt{FC2TC(}\varphi_2\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by induction hypotheses} \\ & \text{iff} & s \in \big(\texttt{FC2TC(}\varphi_1 \vee \varphi_2\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by Line 3 of } \texttt{FC2TC} \end{array}$$

   therefore $[\varphi_1 \vee \varphi_2]_{\mathcal{K}} = \big(\texttt{FC2TC(}\varphi_1 \vee \varphi_2\texttt{, } FC\texttt{)}\big)^{\mathcal{K}}$.

3. Let $\varphi = \mathbf{EX}\ \varphi_1$. For every s, we have:

   $$\begin{array}{llll} s \in [\mathbf{EX}\ \varphi_1]_{\mathcal{K}} & \text{iff} & \exists s' \bullet\ N(s,s') \wedge s' \in [\varphi_1]_{\mathcal{K}} & \textit{by semantics of CTL} \\ & \text{iff} & \exists s' \bullet\ N(s,s') \wedge & \\ & & s' \in \big(\texttt{FC2TC(}\varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by induction hypothesis} \\ & \text{iff} & s \in \big(\texttt{FC2TC(}\mathbf{EX}\ \varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}} & \textit{by Line 4 of } \texttt{FC2TC} \end{array}$$

   therefore $[\mathbf{EX}\ \varphi_1]_{\mathcal{K}} = \big(\texttt{FC2TC(}\mathbf{EX}\ \varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}}$.

4. Let $\varphi = \varphi_1\ \mathbf{EU}\ \varphi_2$. Proof of this case is same as the proof of Lemma 7 on page 71.

5. Let $\varphi = \mathbf{E_C G}\ \varphi_1$. We need to prove two cases:

   - **Case 1:** if $s \in [\mathbf{E_C G}\ \varphi_1]_{\mathcal{K}}$, then $s \in \big(\texttt{FC2TC(}\mathbf{E_C G}\ \varphi_1\texttt{, } FC\texttt{)}\big)^{\mathcal{K}}$. By the semantics of CTLFC, we can conclude that there exists a path starting at s, such that every state along the path satisfy φ_1, and FC is satisfied an infinite number of times. Since the state space is finite, there must exists a state t that satisfies FC, and it appears more than once along the path. Since t appears more than once, it can reach itself. Now there are two cases: 1) s is equal to t, 2) s is not equal to t, which in this case, t is reachable from s. This observation is formalized by the constraint in Line 6 of Figure 6.5 on the preceding page:

     $$\exists t \bullet\ FC(t)\ \wedge\ M^+(t,t) \wedge \big(s = t\ \vee\ M^+(s,t)\big),$$

     where, M is equal to restricting the domain of N to those states that satisfy φ_1:

     $$M(x,y)\texttt{:=}\ \texttt{FC2TC(}\varphi_1\texttt{, } FC\texttt{)}(x)\ \wedge\ N(x,y)$$

- **Case 2:** if $s \in \left(\texttt{FC2TC(}\mathbf{E_C G}\ \varphi_1\texttt{,}\ FC\texttt{)}\right)^{\mathcal{K}}$, then $s \in [\mathbf{E_C G}\ \varphi_1]_{\mathcal{K}}$. Because of the constraint in Line 6 of Figure 6.5 on page 74, there exists a state t that satisfies the fairness constraint, and it can reach itself (t is on a loop). Moreover, s is either t or it can reach t. In both cases, we can conclude that s is on an infinite fair path such that every state along the path satisfies φ_1.

By putting the two cases together, we have the following:

$$[\mathbf{E_C G}\ \varphi_1]_{\mathcal{K}} = \left(\texttt{FC2TC(}\mathbf{E_C G}\ \varphi_1\texttt{,}\ FC\texttt{)}\right)^{\mathcal{K}}$$

□

Corollary 1. (Model checking finite symbolic Kripke structures) Let $\mathscr{B}$ be a Kripke base, Σ a symbolic Kripke structure over $\mathscr{B}$, $FC/1$ a relational symbol that represents a fairness constraint, and φ a CTLFC formula. The following holds:

$$\Sigma \vDash_c \varphi \quad \text{iff} \quad \Sigma \vDash \forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{,}\ FC\texttt{)}(s)$$

6.4 Related Work

Our translation of CTLFC to FOLTC was inspired by Immerman and Vardi's work on translating CTL* to FOLTC [42]. The key difference from the work of Immerman and Vardi is that in our work each formula can be defined directly; support for CTL* would require the introduction of a new Boolean variable into the transition system for each sub-formula of the property. Our translation of CTLFC to FOLTC does not require the introduction of Boolean variables and it is linear with respect to the size of input formula.

The `ordering` module of Alloy can be used for bounded model checking of safety properties. This approach does not support model checking liveness properties or even safety with fairness constraints. Our approach, which is available as `ctlfc` and module in Alloy, supports much more sophisticated temporal properties (see next chapter).

A declarative relational modelling language for Kripke structures has been proposed by Chang and Jackson [22]. They augment the traditional languages of model checkers by sets and relations and declarative constructs to specify a transition system. They have developed a BDD-based model checker that supports relations as a data-type. Our approach to model checking is to reduce it to validity checking and use constraint solvers as model checkers. This allows one to explore structural properties along with dynamic ones; moreover; their technique is not capable of model checking a class of models.

B [4] is a modelling language that has many similarities with Alloy. Models developed in B are called B-machines, and the variables used to define the state space can be sets and relations. ProB [48] is a tool for analyzing finite B machines, in particular, model checking and automatic refinement checking of B machines. ProB provides LTL model checking. LTL properties are checked by explicit state-space search. Since each single state in a B machine represents some sets and relations, computing the set of the next states of a single state is computationally very costly. The focus of ProB, similar to Chang and Jackson [22], is to provide a more convenient language form expressing a single Kripke structure.

The Abstract State Machine (ASM) method [15] is for high-level system design and analysis. The ASM method can be used to specify an infinite transition system. Analysis techniques for the ASM method include theorem proving [31,50], and model checking [30], which consists of translating an ASM to SMV by fixing the size of the scopes in the ASM.

DynAlloy is an extension to Alloy for describing the dynamic properties of programs by using actions [35]. It provides partial correctness analysis of DynAlloy models by using the Alloy Analyzer. The major difference between a program and a Kripke structure is that Kripke structures are used to express systems that do not terminate, whereas programs need to terminate.

6.5 Summary

In this chapter, we introduced CTL\EG: a fragment of CTL whose model checking is reducible to validity checking of FOLTC formulas. The transitive closure in FOLTC allowed us to encode more temporal formulas than was possible in FOL. CTL\EG does not include **EG**.

In order to express the model checking of **EG**, we restricted our attention to finite Kripke structures. The finiteness assumption implies that the only way to have an infinite path in a finite Kripke structure is through repetition of states. Using the finiteness assumption and the transitive closure operator, we were able to encode all CTLFC formulas in FOLTC. First, we presented our reduction when we have a single fairness constraint.

Chapter 7

Model Checking in FOLTC: Practice

Alloy is a formal modelling language that provides FOLTC for modelling and specification [43]. The Alloy Analyzer is used to check the consistency of Alloy models for finite scopes: checking if a set of formulas has a satisfying interpretation with a certain *finite* size [44]. Since the number of interpretations that have a certain finite size is finite, finite scope analysis is decidable. The Alloy Analyzer makes it possible to "explore" systematically finite instances of abstract models. These explorations are justified by the *small scope hypothesis*: if there is an error in a model, it can be revealed by studying small instances of the model.

In this chapter, we use our encoding of CTLFC in FOLTC to model check and analyze temporal properties of some abstract models using the Alloy Analyzer. Most of the content of this chapter has been published [55]. This chapter is organized as follows: Section 7.1 shows how relational constructs of Alloy can be used to concisely represent CTLFC formulas in Alloy. In Section 7.2, we present evaluation results on how the Alloy Analyzer scales in the presence of CTLFC formulas. Section 7.3 presents an application of our reduction of CTLFC to FOLTC beyond model checking symbolic Kripke structures.

7.1 CTLFC in Alloy

Alloy provides FOL, set/relational operators similar to the ones on Table 2.1 on page 10, and transitive closure for modelling. Alloy and FOLTC have the same expressive power, and every set/relational operator in Alloy can be expressed in FOLTC. For example, if $R_1/2$ and $R_2/2$ are two binary relational symbols, in Alloy one can write $R_1 \mathbin{.} R_2$ to form

Table 7.1: Alloy's set and relational operators

Alloy syntax	$(A \texttt{\&} B)^{\mathcal{I}}$	$(A\texttt{+}B)^{\mathcal{I}}$	$(A\texttt{-}B)^{\mathcal{I}}$	$(A\,\texttt{.}\,B)^{\mathcal{I}}$	$(A\texttt{<:}B)^{\mathcal{I}}$	$(A\texttt{:>}B)^{\mathcal{I}}$	$(\texttt{\^{}}A)^{\mathcal{I}}$	`iden`
Semantics	$A^{\mathcal{I}} \cap B^{\mathcal{I}}$	$A^{\mathcal{I}} \cup B^{\mathcal{I}}$	$A^{\mathcal{I}} \setminus B^{\mathcal{I}}$	$A^{\mathcal{I}} ; B^{\mathcal{I}}$	$A^{\mathcal{I}} \lhd B^{\mathcal{I}}$	$A^{\mathcal{I}} \rhd B^{\mathcal{I}}$	$(A^{\mathcal{I}})^{+}$	identity relation

a new binary relational symbol, and its semantics for an interpretation $\mathcal{I}$ is defined as follows:

$$(R_1 \,\texttt{.}\, R_2)^{\mathcal{I}} \quad \texttt{:=} \quad R_1^{\mathcal{I}} ; R_2^{\mathcal{I}}$$

In FOLTC, one can introduce a new binary relational symbol R_3 and have the following axiom:

$$\forall x, y \bullet \quad R_3(x, y) \;\Leftrightarrow\; \exists z \bullet\; R_1(x, z) \wedge R_2(z, y)$$

With this axiom, R_3 is equivalent to $R_1 \,\texttt{.}\, R_2$. As this example shows, the benefit of set/relational operators is the concision that they provide in modelling. Table 7.1 presents a subset of the set/relational operators and their semantics in Alloy.

We rewrite the translation function `FC2TC` in Figure 6.5 on page 74 using the operators of Table 7.1. We call this function `FC2ALLOY`: it takes a CTLFC formula φ and a fairness constraint $FC/1$ as input, and it generates a set expression `FC2ALLOY(`φ`,` FC`)` in Alloy that represents the set of states that satisfy φ with fairness constraint FC.

Definition 21. (Translation of CTLFC to Alloy) Let φ a CTLFC formula and $FC/1$ a predicate for a fairness constraint, and N a next-state relation. We have the following:

```
FC2ALLOY($\varphi$, $FC$) :=
  case $\varphi$ of
    $P$                  =>  $P$
    $\neg\varphi$                 =>  $S$ - FC2ALLOY($\varphi$, $FC$)
    $\varphi \vee \psi$                =>  FC2ALLOY($\varphi$, $FC$) + FC2ALLOY($\psi$, $FC$)
    $\mathbf{EX}\varphi$                =>  $N$.FC2ALLOY($\varphi$, $FC$)
    $\varphi\mathbf{EU}\psi$               =>  Let $M$ :=  FC2ALLOY($\varphi$, $FC$)<:$N$ in
                             FC2ALLOY($\psi$, $FC$) + ((^$M$).FC2ALLOY($\psi$, $FC$))
    $\mathbf{E_C G}\varphi$               =>  Let $M$ :=  FC2ALLOY($\varphi$, $FC$)<:$N$ in
                             Let $L$ :=  $FC$ & ($S$.(iden & ^$M$)) in
                             $L$ + ((^$M$).$L$)
```

■

Given a symbolic Kripke structure Σ over a Kripke base $\mathscr{B}$ and a CTLFC formula φ, the Alloy Analyzer can be used to check the following property, which is equivalent to $\Sigma \vDash_c \varphi$:

$$\Sigma \vDash S_0 \; \texttt{in FC2ALLOY(} \varphi \texttt{,} \; FC \texttt{)}$$

To make model checking in Alloy easy and accessible, we have developed a parametrized Alloy module `ctlfc.als` so that users can import the definitions of the temporal logic operators. The parameter of this module is the set of states. The universal path quantifiers, $\mathbf{A_C X}$, $\mathbf{A_C G}$, and $\mathbf{A_C U}$ have been defined in terms of the existential operators. This module is provided as Appendix B.1 on page 101.

7.2 Case Studies

We completed several examples to show that our method makes it possible to model check CTLFC specifications of symbolic Kripke structures in the Alloy Analyzer, thereby validating the simplicity and utility of our approach. We used three examples from different domains.: 1) the address book from Jackson [44], 2) a features interaction (FI) between call-waiting and call-forwarding, 3) a traffic light controller [49]. These models satisfy their temporal specifications. Our parametrized Alloy module for CTLFC hides the details of model checking in Alloy for a user, so that temporal specifications can be added to models smoothly. We used the Alloy Analyzer 4.2 along with the MiniSat SAT-solver [33]. The experiments were run on an Intel Core 2 Due 2.40 GHz machine running Ubuntu 10.04 with up to 3G of user-space memory. In the following, we discuss the case studies. The Alloy models are found in Appendix B.

7.2.1 Address Book

The subject of this case study is an address book system, which is an association between names and addresses. The system moves from one state to another by applying operations, such as add and delete, on the address book. We checked a safety property that was originally analyzed for bounded computation paths [44]. The safety property states that there is no state of the address book system where a name in the address book does not have an associated address; in other words, the operations on the address book preserve its integrity. Using our method, the Alloy Analyzer was able to analyze the address book model with unbounded paths up to the same scope as bounded paths.

Table 7.2: Experimental results. SS: Scope Size, min: minute, sec: seconds

Address Book		Features Interaction		Traffic Light Controller	
SS	Time	SS	Time	SS	Time
14	1 min 14 sec	10	14.28 sec	7	4.71 sec
15	2 min 57 sec	11	2 min 7.6 sec	8	36.81 sec
16	9 min 15 sec	12	20 min 51 sec	9	12 min 42 sec
17	13 min 43 sec	13	> 1 hour	10	> 1 hour
Safety		Safety		Safety with fairness	

7.2.2 Features Interaction

In this case study, we analyzed the interaction between two telephony features, call-waiting and call-forwarding. Each state in this case study is represented by 6 relations: 1) idle: set of phone numbers that are idle, 2) calling: a binary relation representing phone numbers that are trying to reach others, 3) talkingTo: a binary relation representing phone numbers that are successfully connected to each other, 4) waitingFor: a binary relation representing phone numbers that call waiting has made them to wait, 5) forwardedTo: a binary relation representing phone numbers that call forwarding has forwarded, 6) busy: a binary relation for modelling busy phone numbers. We checked a safety property that stated no call is being forwarded and is also made to wait by a priority-based protocol. We made this case study ourselves.

7.2.3 Traffic Light Controller

This case study is about a three-way traffic light controller. We checked a safety property with 3 fairness constraints. The safety property stated that no two lights are green at the same time. The fairness constraints were used to enforce that there is a request for the green light each way infinitely often.

7.2.4 Scalability of Case Studies

Table 7.2 presents data on the types of properties, scope size, and the Alloy Analyzer time to check the property. With respect to scalability, we found that temporal specifications can be analyzed up to the size of the scopes that non-temporal specifications are often analyzed in Alloy. Thus, our method is immediately valuable to those who use Alloy for modelling and analysis now relying on the *small scope hypothesis*. These models are not as large as those that can be checked using a model checker such as SMV [49], however, the declarative and relational aspects of Alloy have significant advantages for creating abstract, concise models, and we now provide the ability to check CTLFC specifications directly on small scopes of these models.

7.3 Beyond Model Checking CTLFC

Theorem 11 on page 74 states that $\texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}$ represents the set of states that φ. We have used this result to reduce verifying $\Sigma \models_c \varphi$ to the following validity problem in FOLTC:

$$\Sigma \models \forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s)$$

The formula $\forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s)$ states that every initial state satisfies φ. This formula could also be used as part of the description of a symbolic Kripke structure and "enforce" the property φ declaratively. In this way, a modeller states "what" the behaviour of the system is rather that "how" it is achieved.

Adding the formula $\forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s)$ to the description of a model along with a finite scope analyzer, such as the Alloy Analyzer, could be used to answer some interesting questions. The rest of this section discusses an application of adding formulas of the form $\forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s)$ to a symbolic Kripke structure.

Given a symbolic Kripke structure Σ and a CTLFC formula φ, Σ existentially satisfies φ iff there is at least one Kripke structure $\mathcal{K}$ that satisfies Σ and φ. We define this as *existential model checking* [55]. The difference between existential model checking and $\Sigma \models_c \varphi$ is that in the latter case every $\mathcal{K}$ that satisfies Σ must satisfy φ.

Existential model checking is useful for design exploration. For example, when details to be added in the future will constrain a symbolic Kripke structure of interest, a user needs to know whether the abstract model can be extended into a more detailed model that satisfies the property.

To check if Σ existentially satisfies φ, all we need to do is to check for the satisfiability of the following sef of FOLTC formulas:

$$\Sigma \cup \{\forall s \bullet\ S_0(s) \Rightarrow \texttt{FC2TC(}\varphi\texttt{, }FC\texttt{)}(s)\}$$

We did a case study to show the value of existential model checking by encoding the semantics of the untyped λ-calculus as a symbolic Kripke structure. We used existential model checking to generate a λ-term that does not have a normal form, e.g., $(\lambda x.xx)(\lambda x.xx)$, and a term that has a normal form but not necessarily every reduction path terminates, e.g., $(\lambda x.(\lambda x.xx))((\lambda x.xx)(\lambda x.xx))$. As this example suggests, one way of using existential model checking is to generate interesting instances. In general, existential model checking can help a user to have a better understanding about a declarative and abstract model by checking the *existence* of specific instances; in other words, existential model checking can be considered as an approach to "exploring" a declarative symbolic Kripke structure. The Alloy model for this case study is available in Appendix B.5.

7.4 Summary

In this chapter, we showed how our reduction of CTLFC to FOLTC can be used to analyze temporal properties in the Alloy Analyzer. Our case studies showed that the Alloy Analyzer can be used to analyze CTLFC formulas up to the same scopes that Alloy models are analyzed. We also used this encoding to solve the existential model checking problem. We have developed an Alloy module that can be used to express CTLFC formulas in Alloy (Appendix B.1). This module hides the details of the translation.

Chapter 8

Conclusion

In this thesis, we focused on formulating the unbounded model checking problem as a theorem proving problem for logics that have sophisticated automated provers. We showed that for a fragment of computational tree logic (CTL), which we called *CTL-Live*, the model checking problem is reducible to FOL validity checking. CTL-Live includes the CTL connectives that are often used to express liveness properties (e.g., **AF**, **AU**, etc.). Our key insight in this reduction is the use of the implicit higher-order universal quantifier in the definition of FOL validity. This universal quantifier allows us to describe the semantics of CTL-Live formulas.

Based on this reduction, we used SMT solvers for model checking some case studies. Our case studies showed that SMT solvers, in particular Z3, are effective in verifying CTL-Live properties of infinite systems. Our case studies also showed that SMT solvers are faster in model checking some infinite systems than model checking a finite version of them. We also showed that there are some modelling techniques that alter the performance of solvers, and these techniques are solver dependent.

By examining the **AE** fragment of FOL and some dialects of description logics, we derived decidability results for CTL-Live model checking.

We also showed that CTL-Live is the largest fragment of CTL whose model checking is reducible to FOL validity checking. The practical implication of this result is that in order to reduce model checking of more temporal operators than those found in CTL-Live to validity checking, one needs to consider more expressive logics than FOL. For this reason, we then focused on FOL plus transitive closure (FOLTC). We reduced model checking of a more expressive fragment of CTL, which we called CTL**EG**, to validity checking for FOLTC. CTL**EG** is more expressive than CTL-Live and yet less expressive than CTL. By adding a finiteness restriction, we reduced model checking all of CTL with fairness constraints (CTLFC) formulas to validity checking. The finiteness restriction requires that the system under study must have a finite number of states, but it does not require this number to be known. There are two major insights in this reduction: 1) transitive closure contains reachability information, 2) given a finite number of states, the only way to have an infinite path is through repetition of states.

We used the Alloy Analyzer to analyze CTLFC properties of some Alloy models. Our case studies have shown that CTLFC properties can be used in Alloy and analyzed up to the same scopes that the Alloy models are analyzed.

The following lists the contributions of this thesis:

- Introducing CTL-Live: a fragment of CTL whose model checking problem is reducible to FOL validity checking.
- Using SMT solvers, we have shown the effectiveness of SMT solvers in model checking CTL-Live properties.
- Proving the maximality of CTL-Live: the model checking of CTL connectives that are not included in CTL-Live is not reducible to FOL validity checking.
- Showing that the inductive invariant method for verification of safety properties is not complete.
- Deriving decidability results for CTL-Live model checking by examining some decidable fragments of FOL.
- Showing how CTL connectives, except **EG**, can be encoded as FOLTC formula.
- Showing how all of CTLFC formula can be encoded in FOLTC for finite systems.
- Using the Alloy Analyzer, we have shown CTLFC properties can be used in Alloy and analyzed up to the same scopes that the Alloy models are analyzed.

8.1 Future Work

Some future directions for research from the results of this research are the followings:

- **Counterexample generation:** The output of our model checking techniques is a yes-no answer. In the case of no, we do not provide counterexamples, which are effective in revealing the source of a bug. Techniques for counterexample generation will increase the adaptability of our techniques in practice.

- **Evidence generation from proofs:** SMT solvers are capable of generating a proof when a set of formulas is unsatisfiable. In the context of CTL-Live model checking, this means that if a system satisfies a CTL-Live formula, the SMT solver will generate a proof. We are interested in converting such a proof into a finite path or tree that shows to the user why a CTL-Live formula holds. Such evidence can improve one's understanding of why a system satisfies a CTL-Live formula.

- **Practicality of decidability results for CTL-Live model checking:** the decidability results of Section 4.5 on page 45 were derived by examining decidable fragments of FOL. These fragments are less expressive than FOL. None of our case studies fall into these decidable categories. We are planing to study the expressive power of these fragments in formalizing systems.

8.2 Final Word

The complexity of the tools being used in the world of computer-aided verification is constantly increasing. The birth of "solver competitions", such as SAT, SMT, Hardware Model Checking, Software Verification etc., is evidence of this fact. This complexity is making it hard to be proficient in both developing a tool and applications of a tool; as a result, research in computer-aided verification can be roughly divided into two categories: 1) tool development, 2) those who use tools for solving verification problems. Both categories benefit from each other's advancements: advancements in tools result in improvements in solutions that are based on them and applications of tools reveal their relevance and can guide future improvements.

This thesis belongs to the latter category: applications of FOL reasoners in model checking. We believe this kind of research is very important in the success of computer-aided verification in practice.

Appendix A

SMT-LIB Models

A.1 Leader Election Protocol

```
; Copyright (C) 2014 Amirhossein Vakili and
; Nancy A. Day <https://cs.uwaterloo.ca/~nday>
;
; Permission is hereby granted, free of charge, to any person obtaining a
; copy of this software and associated documentation files (the "Software"),
; to deal in ; the Software without restriction, including without
; limitation the rights to use, copy, modify, merge, publish, distribute,
; sublicense, and/or sell copies of the Software, and to permit persons to
; whom the Software is furnished to do so, subject to the following
; conditions:
;
; The above copyright notice and this permission notice shall be included in
; all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
; THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
; FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
; DEALINGS IN THE SOFTWARE.
;
;
(declare-fun LID () Int)
(declare-fun AFP (Int) Bool)
(declare-fun token3 (Int) Int)
```

```
(declare-fun token2 (Int) Int)
(declare-fun token1 (Int) Int)
(declare-fun token0 (Int) Int)
(declare-fun token7 (Int) Int)
(declare-fun token6 (Int) Int)
(declare-fun token5 (Int) Int)
(declare-fun token4 (Int) Int)
(declare-fun pid1 () Int)
(declare-fun pid0 () Int)
(declare-fun pid3 () Int)
(declare-fun pid2 () Int)
(declare-fun pid5 () Int)
(declare-fun pid4 () Int)
(declare-fun pid7 () Int)
(declare-fun pid6 () Int)
(assert (not (= pid0 pid1)))
(assert (not (= pid0 pid2)))
(assert (not (= pid0 pid3)))
(assert (not (= pid0 pid4)))
(assert (not (= pid0 pid5)))
(assert (not (= pid0 pid6)))
(assert (not (= pid0 pid7)))
(assert (not (= pid1 pid2)))
(assert (not (= pid1 pid3)))
(assert (not (= pid1 pid4)))
(assert (not (= pid1 pid5)))
(assert (not (= pid1 pid6)))
(assert (not (= pid1 pid7)))
(assert (not (= pid2 pid3)))
(assert (not (= pid2 pid4)))
(assert (not (= pid2 pid5)))
(assert (not (= pid2 pid6)))
(assert (not (= pid2 pid7)))
(assert (not (= pid3 pid4)))
(assert (not (= pid3 pid5)))
(assert (not (= pid3 pid6)))
(assert (not (= pid3 pid7)))
(assert (not (= pid4 pid5)))
(assert (not (= pid4 pid6)))
(assert (not (= pid4 pid7)))
(assert (not (= pid5 pid6)))
(assert (not (= pid5 pid7)))
(assert (not (= pid6 pid7)))
(assert (or (= LID pid0) (= LID pid1) (= LID pid2) (= LID pid3) (= LID pid4)
     (= LID pid5) (= LID pid6) (= LID pid7)))
```

```
(assert (>= LID pid0))
(assert (>= LID pid1))
(assert (>= LID pid2))
(assert (>= LID pid3))
(assert (>= LID pid4))
(assert (>= LID pid5))
(assert (>= LID pid6))
(assert (>= LID pid7))
(assert (= (token0 0) pid0))
(assert (= (token1 0) pid1))
(assert (= (token2 0) pid2))
(assert (= (token3 0) pid3))
(assert (= (token4 0) pid4))
(assert (= (token5 0) pid5))
(assert (= (token6 0) pid6))
(assert (= (token7 0) pid7))
(assert (forall ((t Int)) (=> (>= t 0) (= (token0 (+ t 1)) (ite (> (token0 t) (token1 t)) (token0 t) (token1 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token1 (+ t 1)) (ite (> (token1 t) (token2 t)) (token1 t) (token2 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token2 (+ t 1)) (ite (> (token2 t) (token3 t)) (token2 t) (token3 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token3 (+ t 1)) (ite (> (token3 t) (token4 t)) (token3 t) (token4 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token4 (+ t 1)) (ite (> (token4 t) (token5 t)) (token4 t) (token5 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token5 (+ t 1)) (ite (> (token5 t) (token6 t)) (token5 t) (token6 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token6 (+ t 1)) (ite (> (token6 t) (token7 t)) (token6 t) (token7 t))))))
(assert (forall ((t Int)) (=> (>= t 0) (= (token7 (+ t 1)) (ite (> (token7 t) (token0 t)) (token7 t) (token0 t))))))
(assert (forall ((t Int)) (=> (and (>= t 0) (= (token0 t) LID)) (AFP t))))
(assert (forall ((t Int)) (=> (AFP (+ t 1)) (AFP t))))
(assert (not (AFP 0)))
(check-sat)
```

A.2 Bakery Algorithm

```
; Copyright (C) 2014 Amirhossein Vakili and
; Nancy A. Day <https://cs.uwaterloo.ca/~nday>
;
```

```
; Permission is hereby granted, free of charge, to any person obtaining a
; copy of this software and associated documentation files (the "Software"),
; to deal in ; the Software without restriction, including without
; limitation the rights to use, copy, modify, merge, publish, distribute,
; sublicense, and/or sell copies of the Software, and to permit persons to
; whom the Software is furnished to do so, subject to the following
; conditions:
;
; The above copyright notice and this permission notice shall be included in
; all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
; THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
; FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
; DEALINGS IN THE SOFTWARE.
;
;
(declare-sort State 0)
(declare-fun C () State)
(declare-fun T () State)
(declare-fun W () State)
(declare-fun AF (State Int State Int) Bool)
(define-fun init ((c1 State) (t1 Int) (c2 State) (t2 Int)) Bool (and (= c1 W
   ) (> t1 0) (or (= c2 T) (= c2 W) (= c2 C)) (>= t2 0)))
(define-fun eT ((c1 State) (t1 Int) (c2 State) (t2 Int) (c1p State) (t1p Int
   ) (c2p State) (t2p Int)) Bool (and (= c1 T) (= c1p W) (= t1p (+ t2 1)) (=
    c2 c2p) (= t2 t2p)))
(define-fun eW ((c1 State) (t1 Int) (c2 State) (t2 Int) (c1p State) (t1p Int
   ) (c2p State) (t2p Int)) Bool (and (= c1 W) (or (< t1 t2) (= t2 0)) (=
   c1p C) (= t1 t1p) (= c2 c2p) (= t2 t2p)))
(define-fun eC ((c1 State) (t1 Int) (c2 State) (t2 Int) (c1p State) (t1p Int
   ) (c2p State) (t2p Int)) Bool (and (= c1 C) (= c1p T) (= t1p 0) (= c2 c2p
   ) (= t2 t2p)))
(define-fun next ((c1 State) (t1 Int) (c2 State) (t2 Int) (c1p State) (t1p
   Int) (c2p State) (t2p Int)) Bool (and (or (= c1 T) (= c1 W) (= c1 C)) (>=
    t1 0) (or (= c2 T) (= c2 W) (= c2 C)) (>= t2 0) (or (= c1p T) (= c1p W)
   (= c1p C)) (>= t1p 0) (or (= c2p T) (= c2p W) (= c2p C)) (>= t2p 0) (or (
   eT c1 t1 c2 t2 c1p t1p c2p t2p) (eW c1 t1 c2 t2 c1p t1p c2p t2p) (eC c1
   t1 c2 t2 c1p t1p c2p t2p) (eT c2 t2 c1 t1 c2p t2p c1p t1p) (eW c2 t2 c1
   t1 c2p t2p c1p t1p) (eC c2 t2 c1 t1 c2p t2p c1p t1p))))
```

```
(define-fun dead_end ((c1 State) (t1 Int) (c2 State) (t2 Int)) Bool (forall ((c1p State) (t1p Int) (c2p State) (t2p Int)) (not (next c1 t1 c2 t2 c1p t1p c2p t2p))))
(assert (not (= T W)))
(assert (not (= T C)))
(assert (not (= W C)))
(assert (forall ((c1 State) (t1 Int) (c2 State) (t2 Int)) (=> (or (dead_end c1 t1 c2 t2) (= c1 C)) (AF c1 t1 c2 t2))))
(assert (forall ((c1 State) (t1 Int) (c2 State) (t2 Int)) (=> (forall ((c1p State) (t1p Int) (c2p State) (t2p Int)) (=> (next c1 t1 c2 t2 c1p t1p c2p t2p) (AF c1p t1p c2p t2p))) (AF c1 t1 c2 t2))))
(assert (not (forall ((c1 State) (t1 Int) (c2 State) (t2 Int)) (=> (init c1 t1 c2 t2) (AF c1 t1 c2 t2)))))
(check-sat)
```

A.3 Collision Avoidance State-Flow Model

```
; Copyright (C) 2014 Amirhossein Vakili and
; Nancy A. Day <https://cs.uwaterloo.ca/˜nday>
;
; Permission is hereby granted, free of charge, to any person obtaining a
; copy of this software and associated documentation files (the "Software"),
; to deal in ; the Software without restriction, including without
; limitation the rights to use, copy, modify, merge, publish, distribute,
; sublicense, and/or sell copies of the Software, and to permit persons to
; whom the Software is furnished to do so, subject to the following
; conditions:
;
; The above copyright notice and this permission notice shall be included in
; all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
; THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
; FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
; DEALINGS IN THE SOFTWARE.
;
;
(declare-sort State 0)
(declare-sort PRNDL 0)
```

```
(declare-sort Threat 0)
(declare-fun EF_IDLE (State) Bool)
(declare-fun WAHN () State)
(declare-fun EF_MITIGATE (State) Bool)
(declare-fun EF_DISABLED (State) Bool)
(declare-fun HALT () State)
(declare-fun MITIGATE () State)
(declare-fun G4 () PRNDL)
(declare-fun G3 () PRNDL)
(declare-fun G2 () PRNDL)
(declare-fun G1 () PRNDL)
(declare-fun G0 () PRNDL)
(declare-fun IDLE () State)
(declare-fun EF_OVERRIDE (State) Bool)
(declare-fun OVERRIDE () State)
(declare-fun DISENGAGED () State)
(declare-fun AVOID () State)
(declare-fun T2 () Threat)
(declare-fun T3 () Threat)
(declare-fun T0 () Threat)
(declare-fun T1 () Threat)
(declare-fun DISABLED () State)
(declare-fun THRESHOLD () Int)
(declare-fun EF_HALT (State) Bool)
(declare-fun FAIL () State)
(declare-fun EF_AVOID (State) Bool)
(declare-fun EF_WAHN (State) Bool)
(declare-fun EF_FAIL (State) Bool)
(declare-fun EF_DISENGAGED (State) Bool)
(define-fun in_engaged ((s State)) Bool (or (= s IDLE) (= s AVOID) (= s WAHN
   ) (= s MITIGATE)))
(define-fun in_enabled ((s State)) Bool (or (= s DISENGAGED) (in_engaged s)
   (= s HALT)))
(define-fun t14 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (= s DISABLED) ca_enable (= sp DISENGAGED)))
(define-fun t38 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (= s OVERRIDE) (not ca_enable) (= sp DISABLED)))
(define-fun t39 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (= s OVERRIDE) error (= sp FAIL)))
(define-fun t37 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (= s OVERRIDE) (< accel_pedal 35) (= sp DISENGAGED)))
```

```
(define-fun t15 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (in_enabled s) (not ca_enable) (= sp DISABLED)))
(define-fun t36 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (in_enabled s) (>= accel_pedal 35) (= sp OVERRIDE)))
(define-fun t27 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (in_enabled s) error (= sp FAIL)))
(define-fun t24 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (= s HALT) (not (not ca_enable)) (not (>= accel_pedal
    35)) (not error) (> brake_pedal 10) (= sp DISENGAGED)))
(define-fun t16 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (= s DISENGAGED) (not (not ca_enable)) (not (>=
    accel_pedal 35)) (not error) (> speed THRESHOLD) (= prndl G3) (= sp IDLE)
    ))
(define-fun t17 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (in_engaged s) (not (not ca_enable)) (not (>=
    accel_pedal 35)) (not error) (or (and (not (= speed 0)) (<= speed
    THRESHOLD)) (not (= prndl G3))) (= sp DISENGAGED)))
(define-fun t25 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (in_engaged s) (not (not ca_enable)) (not (>=
    accel_pedal 35)) (not error) (= speed 0) (= sp HALT)))
(define-fun t19 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (or (= s IDLE) (= s AVOID) (= s MITIGATE)) (not (not
    ca_enable)) (not (>= accel_pedal 35)) (not error) (not (or (and (not (=
    speed 0)) (<= speed THRESHOLD)) (not (= prndl G3)))) (not (= speed 0)) (=
     threat T1) (= sp WAHN)))
(define-fun t20 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (or (= s AVOID) (= s WAHN) (= s MITIGATE)) (not (not
    ca_enable)) (not (>= accel_pedal 35)) (not error) (not (or (and (not (=
    speed 0)) (<= speed THRESHOLD)) (not (= prndl G3)))) (not (= speed 0)) (=
     threat T0) (= sp IDLE)))
(define-fun t22 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) Bool (and (or (= s WAHN) (= s IDLE) (= s MITIGATE)) (not (not
    ca_enable)) (not (>= accel_pedal 35)) (not error) (not (or (and (not (=
    speed 0)) (<= speed THRESHOLD)) (not (= prndl G3)))) (not (= speed 0)) (=
     threat T2) (= sp AVOID)))
```

```
(define-fun t29 ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (or (= s WAHN) (= s IDLE) (= s AVOID)) (not (not
   ca_enable)) (not (>= accel_pedal 35)) (not error) (not (or (and (not (=
   speed 0)) (<= speed THRESHOLD)) (not (= prndl G3)))) (not (= speed 0)) (=
    threat T3) (= sp MITIGATE)))
(define-fun next ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) Bool (and (or (= s DISABLED) (= s OVERRIDE) (= s FAIL) (= s
   DISENGAGED) (= s HALT) (= s IDLE) (= s WAHN) (= s AVOID) (= s MITIGATE))
    (>= brake_pedal 0) (<= brake_pedal 100) (>= accel_pedal 0) (<=
   accel_pedal 100) (>= speed 0) (or (= threat T0) (= threat T1) (= threat
   T2) (= threat T3)) (or (= prndl G0) (= prndl G1) (= prndl G2) (= prndl G3
   ) (= prndl G4)) (or (t14 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t38 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t39 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t37 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t15 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t36 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t27 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t24 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t16 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t17 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t25 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t19 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t20 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t22 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp) (t29 error ca_enable brake_pedal accel_pedal speed
   threat prndl s sp))))
(assert (>= THRESHOLD 0))
(assert (not (= DISABLED OVERRIDE)))
(assert (not (= DISABLED FAIL)))
(assert (not (= DISABLED DISENGAGED)))
(assert (not (= DISABLED HALT)))
(assert (not (= DISABLED IDLE)))
(assert (not (= DISABLED WAHN)))
(assert (not (= DISABLED AVOID)))
(assert (not (= DISABLED MITIGATE)))
(assert (not (= OVERRIDE FAIL)))
(assert (not (= OVERRIDE DISENGAGED)))
(assert (not (= OVERRIDE HALT)))
(assert (not (= OVERRIDE IDLE)))
(assert (not (= OVERRIDE WAHN)))
(assert (not (= OVERRIDE AVOID)))
(assert (not (= OVERRIDE MITIGATE)))
```

```
(assert (not (= FAIL DISENGAGED)))
(assert (not (= FAIL HALT)))
(assert (not (= FAIL IDLE)))
(assert (not (= FAIL WAHN)))
(assert (not (= FAIL AVOID)))
(assert (not (= FAIL MITIGATE)))
(assert (not (= DISENGAGED HALT)))
(assert (not (= DISENGAGED IDLE)))
(assert (not (= DISENGAGED WAHN)))
(assert (not (= DISENGAGED AVOID)))
(assert (not (= DISENGAGED MITIGATE)))
(assert (not (= HALT IDLE)))
(assert (not (= HALT WAHN)))
(assert (not (= HALT AVOID)))
(assert (not (= HALT MITIGATE)))
(assert (not (= IDLE WAHN)))
(assert (not (= IDLE AVOID)))
(assert (not (= IDLE MITIGATE)))
(assert (not (= WAHN AVOID)))
(assert (not (= WAHN MITIGATE)))
(assert (not (= AVOID MITIGATE)))
(assert (not (= T0 T1)))
(assert (not (= T0 T2)))
(assert (not (= T0 T3)))
(assert (not (= T1 T2)))
(assert (not (= T1 T3)))
(assert (not (= T2 T3)))
(assert (not (= G0 G1)))
(assert (not (= G0 G2)))
(assert (not (= G0 G3)))
(assert (not (= G0 G4)))
(assert (not (= G1 G2)))
(assert (not (= G1 G3)))
(assert (not (= G1 G4)))
(assert (not (= G2 G3)))
(assert (not (= G2 G4)))
(assert (not (= G3 G4)))
(assert (forall ((s State)) (=> (= s DISABLED) (EF_DISABLED s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
   State)) (=> (and (EF_DISABLED sp) (next error ca_enable brake_pedal
   accel_pedal speed threat prndl s sp)) (EF_DISABLED s))))
(assert (forall ((s State)) (=> (= s OVERRIDE) (EF_OVERRIDE s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
   accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
```

```
    State)) (=> (and (EF_OVERRIDE sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_OVERRIDE s))))
(assert (forall ((s State)) (=> (= s FAIL) (EF_FAIL s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_FAIL sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_FAIL s))))
(assert (forall ((s State)) (=> (= s DISENGAGED) (EF_DISENGAGED s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_DISENGAGED sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_DISENGAGED s))))
(assert (forall ((s State)) (=> (= s HALT) (EF_HALT s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_HALT sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_HALT s))))
(assert (forall ((s State)) (=> (= s IDLE) (EF_IDLE s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_IDLE sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_IDLE s))))
(assert (forall ((s State)) (=> (= s WAHN) (EF_WAHN s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_WAHN sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_WAHN s))))
(assert (forall ((s State)) (=> (= s AVOID) (EF_AVOID s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_AVOID sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_AVOID s))))
(assert (forall ((s State)) (=> (= s MITIGATE) (EF_MITIGATE s))))
(assert (forall ((error Bool) (ca_enable Bool) (brake_pedal Int) (
    accel_pedal Int) (speed Int) (threat Threat) (prndl PRNDL) (s State) (sp
    State)) (=> (and (EF_MITIGATE sp) (next error ca_enable brake_pedal
    accel_pedal speed threat prndl s sp)) (EF_MITIGATE s))))
(assert (not (and (EF_DISABLED DISABLED) (EF_OVERRIDE DISABLED) (EF_FAIL
    DISABLED) (EF_DISENGAGED DISABLED) (EF_HALT DISABLED) (EF_IDLE DISABLED)
    (EF_WAHN DISABLED) (EF_AVOID DISABLED) (EF_MITIGATE DISABLED))))
(check-sat)
```

A.4 File System

```
; Copyright (C) 2014 Amirhossein Vakili and
; Nancy A. Day <https://cs.uwaterloo.ca/~nday>
;
; Permission is hereby granted, free of charge, to any person obtaining a
; copy of this software and associated documentation files (the "Software"),
; to deal in ; the Software without restriction, including without
; limitation the rights to use, copy, modify, merge, publish, distribute,
; sublicense, and/or sell copies of the Software, and to permit persons to
; whom the Software is furnished to do so, subject to the following
; conditions:
;
; The above copyright notice and this permission notice shall be included in
; all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
; THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
; FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
; DEALINGS IN THE SOFTWARE.
;
;
(declare-sort State 0)
(declare-sort Data 0)
(declare-sort Key 0)
(declare-fun EF (State) Bool)
(declare-fun S0 () State)
(declare-fun SW () State)
(declare-fun content (State Key) Data)
(declare-fun K0 () Key)
(declare-fun NULL () Data)
(declare-fun D0 () Data)
(define-fun empty ((s State)) Bool (forall ((k Key)) (= (content s k) NULL))
    )
(define-fun write ((k Key) (d Data) (s State) (sp State)) Bool (and (not (=
    d NULL)) (not (= (content s k) NULL)) (= (content sp k) d) (forall ((kp
    Key)) (=> (not (= kp k)) (= (content s kp) (content sp kp))))))
(define-fun add ((k Key) (d Data) (s State) (sp State)) Bool (and (not (= d
    NULL)) (= (content s k) NULL) (= (content sp k) d) (forall ((kp Key)) (=>
     (not (= kp k)) (= (content s kp) (content sp kp))))))
```

```
(define-fun delete ((k Key) (s State) (sp State)) Bool (and (not (= (content s k) NULL)) (= (content sp k) NULL) (forall ((kp Key)) (=> (not (= kp k)) (= (content s kp) (content sp kp))))))
(define-fun next ((s State) (sp State)) Bool (exists ((k Key) (d Data)) (or (add k d s sp) (delete k s sp))))
(assert (forall ((s State) (k Key) (d Data)) (=> (and (not (= d NULL)) (= (content s k) NULL)) (exists ((sp State)) (add k d s sp)))))
(assert (forall ((s State) (k Key)) (=> (not (= (content s k) NULL)) (exists ((sp State)) (delete k s sp)))))
(assert (not (= D0 NULL)))
(assert (write K0 D0 S0 SW))
(assert (EF SW))
(assert (forall ((s State) (sp State)) (=> (and (EF sp) (next s sp)) (EF s))))
(assert (not (EF S0)))
(check-sat)
```

Appendix B

Alloy Models

B.1 CTLFC to FOLTC Module in Alloy

```
/*
 * Copyright (c) 2012, Amirhossein Vakili
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *    1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 *    2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

module temporal_logics/ctlfc[S]

private one sig TS{
    S0: some S,
    sigma: S -> S,
    FC: set S
}

fun initialState: S {TS.S0}

fun nextState: S -> S{TS.sigma}

fun fc: S {TS.FC}

// Helper functions for model checking

private fun bound[R: S -> S, X: S]: S -> S{X <: R}

private fun id[X:S]: S->S{bound[iden,X]}

private fun loop[R: S -> S]: S {S.(^R & id[S])}

// Fair is EcG true

private fun Fair: S{
    let R= TS.sigma|
    let LoopFC = loop[R] & TS.FC|
        (*R).LoopFC
 }

// Temporal operators of CTL and CTLFC

fun not_ctlfc[phi: S]: S {S - phi}

fun and_ctlfc[phi, si: S]: S {phi & si}

fun or_ctlfc[phi, si: S]: S {phi + si}

fun imp_ctlfc[phi, si: S]: S {not_ctlfc[phi] + si}

fun ex[phi: S]: S {TS.sigma.phi}

fun ecx[phi: S]: S {ex[phi & Fair]}
fun ax[phi:S]:S {not_ctlfc[ex[not_ctlfc[phi]]]}

fun acx[phi:S]:S {not_ctlfc[ecx[not_ctlfc[phi]]]}

fun ef[phi: S]: S {(*(TS.sigma)).phi }

fun ecf[phi: S]: S {ef[phi & Fair]}

fun eg[phi: S]: S {
    let R= bound[TS.sigma,phi]|
    let Loop = loop[R]|
        (*R).Loop
}

fun ecg[phi:S]:S {
    let R= bound[TS.sigma,phi]|
    let LoopFC = loop[R] & TS.FC|
        (*R).LoopFC
}

fun af[phi: S]: S {not_ctlfc[eg[not_ctlfc[phi]]]}

fun acf[phi: S]: S {not_ctlfc[ecg[not_ctlfc[phi]]]}

fun ag[phi: S]: S {not_ctlfc[ef[not_ctlfc[phi]]]}

fun acg[phi: S]: S {not_ctlfc[ecf[not_ctlfc[phi]]]}

fun eu[phi, si: S]: S {(*(bound[TS.sigma, phi])).si}

fun ecu[phi, si: S]: S {eu[phi, si & Fair]}

fun acu[phi, si: S]:S {
    not_ctlfc[or_ctlfc[ecg[not_ctlfc[si]],
                       ecu[not_ctlfc[si],
                           not_ctlfc[or_ctlfc[phi, si]]]]]
}

// model checking constraint

pred ctlfc_mc[phi: S]{TS.S0 in phi}
```

B.2 Address Book

```
module tour/addressBook3d ----- this is the final model in fig 2.18

--open util/ordering [Book] as BookOrder
open temporal_logics/ctl[Book]

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }

sig Alias, Group extends Name { }

sig Book {
        names: set Name,
        addr: names->some Target,
    sigma: set Book
} {
        no n: Name | n in n.^addr
        all a: Alias | lone a.addr
}

pred add [b, b’: Book, n: Name, t: Target] {
        t in Addr or some lookup [b, Name&t]
        b’.addr = b.addr + n->t
        b != b’
}

pred del [b, b’: Book, n: Name, t: Target] {
        b != b’
        no b.addr.n or some n.(b.addr) - t
        b’.addr = b.addr - n->t
}

fun lookup [b: Book, n: Name] : set Addr { n.^(b.addr) & Addr }

pred init [b: Book]  { no b.addr }

fact traces {
--      init [first]
--      all b: Book-last |
--        let b’ = b.next |
--          some n: Name, t: Target |
--            add [b, b’, n, t] or del [b, b’, n, t]
        all b,b’:Book|
                ((b' in nextState[b]) implies
                (some n: Name, t: Target | add [b, b', n, t] or del [b, b', n, t]))
        all b,b':Book|((b.addr=b'.addr) implies b=b')
}

------------------------------------------------------

assert delUndoesAdd {
        all b, b', b'': Book, n: Name, t: Target |
                no n.(b.addr) and add [b, b', n, t] and del [b', b'', n, t]
                implies
                b.addr = b''.addr
}

// This should not find any counterexample.
--check delUndoesAdd for 3

------------------------------------------------------

assert addIdempotent {
        all b, b', b'': Book, n: Name, t: Target |
                add [b, b', n, t] and add [b', b'', n, t]
                implies
                b'.addr = b''.addr
}

// This should not find any counterexample.
--check addIdempotent for 3

------------------------------------------------------

assert addLocal {
        all b, b': Book, n, n': Name, t: Target |
                add [b, b', n, t] and n != n'
                implies
                lookup [b, n'] = lookup [b', n']
}

// This should not find any counterexample.
--check addLocal for 3 but 2 Book

------------------------------------------------------

assert lookupYields {
        all b: Book, n: b.names | some lookup [b,n]
}

sig P in Book{}

fact{
        all b:Book| (b in P) iff (all n: b.names | some lookup [b,n])
        init[initialState]
}

assert MC{
        CTL_MC[AG[P]]
//       S0 in Book - (*sigma).(Book - P)
}

// This should not find any counterexample.
--check lookupYields for 3 but 4 Book
// Scope 14: 1 min 14 sec
--check MC for 10 but 4 Book

// Scope 15: 2 min 57 sec
--check MC for 11 but 4 Book

// Scope 16: 9 min 15 sec
--check MC for 12 but 4 Book

//Scope 17: > 1 h
check MC for 13 but 4 Book

// This should not find any counterexample.
--check lookupYields for 6
--check MC for 6
```

B.3 Feature Interaction

```
module featureInteraction

open util/boolean

// Feature={CW,CF} is the set of features.
abstract sig Feature{}
one sig CW,CF extends Feature{}
```

```
// Each phone number can have some features. If a number has call-forwarding (CF),
// fw points to forwarded number.
sig PhoneNumber{ feature: set Feature, fw: lone PhoneNumber}
fact{
        all n:PhoneNumber| CF in n.feature iff some n.fw
}


// Used to model the global states.
sig Environment{
        // Numbers that are idle,
        idle: set PhoneNumber,

        // (a->b) in calling iff a is trying to call b
        calling: PhoneNumber -> PhoneNumber,

        // (a->b) in talking iff a is talking to b
        talkingTo: PhoneNumber -> PhoneNumber,

        // (a->b) in busy iff a wants to talk to b, but b is not idle
        busy: PhoneNumber -> PhoneNumber,

        // (a->b) in waitingFor iff a is waiting for b
        waitingFor: PhoneNumber -> PhoneNumber,

        // (a->b) in forwardedTo iff a is forwarded to b
        forwardedTo: PhoneNumber -> PhoneNumber,

        // represents whether a global state is safe or not
        safe: Bool,

        // the transition relation between global states
        sigma: set Environment
}{
        safe = True iff (no waitingFor.PhoneNumber & forwardedTo.PhoneNumber)
}

// IE is the initial state.

one sig IE extends Environment{}
fact{initial[IE]}
```

```

pred initial[e:Environment]{
        e.idle = PhoneNumber
        no e.calling
        no e.talkingTo
        no e.busy
        no e.waitingFor
        no e.forwardedTo
}

// The following predicates are used

pred idle_calling[e,e’: Environment,n,n’:PhoneNumber]{
        n != n’
        n in e.idle
        e’.idle = ((e.idle) - n)
        (n->n’) in e’.calling
        e.calling = (e’.calling - (n->n’))
        e.talkingTo = e’.talkingTo
        e.busy = e’.busy
        e.waitingFor = e’.waitingFor
        e.forwardedTo = e’.forwardedTo
}

pred calling_talkingTo[e,e’:Environment,n,n’:PhoneNumber]{
        n’ in e.idle
        e’.idle = e.idle - n’
        n -> n’ in (e.calling & e’.talkingTo)
        e’.calling = e.calling - (n -> n’)
        e.talkingTo = e’.talkingTo - (n -> n’)
        e.busy = e’.busy
        e.waitingFor = e’.waitingFor
        e.forwardedTo = e’.forwardedTo
}

pred talkingTo_idle[e,e’:Environment,n,n’:PhoneNumber]{
        n -> n’ in e.talkingTo
        (n + n’) in e’.idle
        e.idle = e’.idle - (n + n’)
        e.busy = e’.busy
        e.calling = e’.calling
        e.waitingFor = e’.waitingFor
        e.forwardedTo = e’.forwardedTo
}

```

```
pred calling_busy[e,e’:Environment,n,n’:PhoneNumber]{
        n’ !in e.idle
        e.idle = e’.idle
        n -> n’ in e.calling & e’.busy
        e’.calling = e.calling - (n -> n’)
        e.talkingTo = e’.talkingTo
        e.busy = e’.busy - (n -> n’)
        e.waitingFor = e’.waitingFor
        e.forwardedTo = e’.forwardedTo
}

pred busy_waitingFor[e,e’:Environment,n,n’:PhoneNumber]{
        CW in n’.feature
        n’ !in PhoneNumber.(e.waitingFor)
        e.idle = e’.idle
        e.calling = e’.calling
        e.talkingTo = e’.talkingTo
        n -> n’ in (e.busy & e’.waitingFor)
        e’.busy = e.busy - (n -> n’)
        e.waitingFor = e’.waitingFor - (n -> n’)
        // if this line is commented, there will be feature interaction
        e.forwardedTo = e’.forwardedTo
}

pred waitingFor_idle[e,e’:Environment,n,n’:PhoneNumber]{
        n -> n’ in e.waitingFor
        n in e’.idle
        e.idle = e’.idle - n
        e.calling = e’.calling
        e.talkingTo = e’.talkingTo
        e.busy = e’.busy
        e’.waitingFor = e.waitingFor - (n -> n’)
}

pred waitingFor_talkingTo[e,e’:Environment,n,n’:PhoneNumber]{
        n -> n’ in (e.waitingFor & e’.talkingTo)
        n’ in e.talkingTo.n’
        e’.waitingFor = e.waitingFor - (n -> n’)
        e’.idle = e.idle - n’
        e.talkingTo = e’.talkingTo - (n -> n’)
        e.busy = e’.busy
        e.forwardedTo = e’.forwardedTo
}

pred busy_forwardedTo[e,e’:Environment,n,n’:PhoneNumber]{
        CF in n'.feature
        n -> n' in e.busy
        e'.busy = e.busy - (n -> n')
        e.idle = e'.idle
        e'.forwardedTo = e.forwardedTo + (n -> n'.fw)
        e.talkingTo = e'.talkingTo
        e.calling = e'.calling
        // if this line is commented, there will be feature interaction
        e.waitingFor = e'.waitingFor
}

pred forwardedTo_calling[e,e':Environment,n,n':PhoneNumber]{
        e.idle = e'.idle
        n -> n' in (e.forwardedTo & e'.calling)
        e'.forwardedTo = e.forwardedTo - (n->n')
        e.calling = e'.calling - (n -> n')
        e.busy = e'.busy
        e.talkingTo = e'.talkingTo
        e.waitingFor = e'.waitingFor
}

pred busy_idle[e,e':Environment,n,n':PhoneNumber]{
        n -> n' in e.busy
        no n'.feature
        e'.busy = e.busy - (n -> n')
        n in e'.idle
        e.idle = e'.idle - n
        e.talkingTo = e'.talkingTo
        e.waitingFor = e'.waitingFor
        e.forwardedTo = e'.forwardedTo
        e.calling = e'.calling
}

fact TRANSITIONS{

        no iden & (^fw)
        all e,e': Environment|
                ((e->e') in sigma) iff
                (some n,n':PhoneNumber|(
                        idle_calling[e,e',n,n'] or calling_talkingTo[e,e',n,
                           n'] or talkingTo_idle[e,e',n,n'] or
                        calling_busy[e,e',n,n'] or busy_waitingFor[e,e',n,n
                           '] or busy_forwardedTo[e,e',n,n'] or
                        busy_idle[e,e',n,n'] or waitingFor_idle[e,e',n,n']
                           or waitingFor_talkingTo[e,e',n,n'] or
```

```
                                forwardedTo_calling[e,e',n,n']))


        all e,e':Environment|(
                ((e.idle = e'.idle) and (e.calling = e'.calling) and
                (e.talkingTo = e'.talkingTo) and (e.busy = e'.busy) and
                (e.waitingFor = e'.waitingFor) and (e.forwardedTo = e'.forwardedTo)) implies (e =e'))

        Environment = IE.(*sigma)

        all e:Environment| blackBox[e]
}

//These are the constraints that are not implemented in sigma, but sigma needs
// to satisfy them.
pred blackBox[e:Environment]{
        no (e.calling.PhoneNumber) & (PhoneNumber.(e.calling))
        no e.idle & (e.calling.PhoneNumber + e.busy.PhoneNumber + PhoneNumber.(e.busy))
        no e.idle & (e.talkingTo.PhoneNumber + PhoneNumber.(e.talkingTo))
        no e.idle & (e.waitingFor.PhoneNumber + PhoneNumber.(e.waitingFor))
        all n:PhoneNumber| lone e.waitingFor.n
}

// AG safe

assert No_Feature_Interaction{
        IE in Environment - (*sigma).(safe.False)
}
//Scope 10: 14.28 sec
--check  No_Feature_Interaction for 6 Environment, 4 PhoneNumber

//Scope 11: 2 min 7.6 sec
--check  No_Feature_Interaction for 7 Environment, 4 PhoneNumber

//Scope 12: 20 min 51 sec
--check  No_Feature_Interaction for 8 Environment, 4 PhoneNumber

//Scope 13: > 1 h
check  No_Feature_Interaction for 9 Environment, 4 PhoneNumber
```

B.4 Traffic Light

```
module TrafficLightController

open util/boolean
open temporal_logics/ctlfc[State]

// There are 3 sensors
abstract sig Sense{}
one sig N_Sense, S_Sense, E_Sense extends Sense{}

// Go is for modeling which direction is allowed to go
abstract sig Go{}
one sig N_Go, S_Go, E_Go extends Go{}

// Request is to latch the traffic sensors input.
abstract sig Request{}
one sig N_Req, S_Req, E_Req extends Request{}

sig State{
        input: set Sense,
        output: set Go,

        req: set Request,
        NS_Lock: Bool // NS_Lock is true iff East is not allowed to go

        //sigma: some State // the transition relation
}

pred initial[s:State]{
        no s.output
        no s.req
        s.NS_Lock = False
}

// setting the initial states
fact{ all s:State| initial[s] iff (s in initialState)}


// Predicates for N_Go
pred N_Go_True[s:State]{
        N_Req in s.req
        N_Go !in s.output
        E_Req !in s.req
}
```

```

pred N_Go_False[s:State]{
        N_Go in s.output
        N_Sense !in s.input
}

pred N_Go_[s,s':State]{
        N_Go_True[s] implies N_Go in s'.output else (N_Go_False[s] implies
           N_Go !in s'.output else (N_Go in s.output iff N_Go in s'.output))
}

// Predicates for S_Go
pred S_Go_True[s:State]{
        S_Req in s.req
        S_Go !in s.output
        E_Req !in s.req
}

pred S_Go_False[s:State]{
        S_Go in s.output
        S_Sense !in s.input
}

pred S_Go_[s,s':State]{
        S_Go_True[s] implies S_Go in s'.output else (S_Go_False[s] implies
           S_Go !in s'.output else (S_Go in s.output iff S_Go in s'.output))
}

// Predicates for E_Go
pred E_Go_True[s:State]{
        E_Req in s.req
        E_Go !in s.output
        s.NS_Lock = False
}

pred E_Go_False[s:State]{
        E_Go in s.output
        E_Sense !in s.input
}

pred E_Go_[s,s':State]{
        E_Go_True[s] implies E_Go in s'.output else (E_Go_False[s] implies
           E_Go !in s'.output else (E_Go in s.output iff E_Go in s'.output))
}

```

```
// Predicates for N_Req
pred N_Req_True[s:State]{
        N_Sense in s.input
}

pred N_Req_False[s:State]{
        N_Go_False[s]
}

pred N_Req_[s,s’:State]{
        N_Req_True[s] implies N_Req in s’.req else (N_Req_False[s] implies
           N_Req !in s’.req else  (N_Req in s.req  iff N_Req in s’.req))
}

// Predicates for S_Req
pred S_Req_True[s:State]{
        S_Sense in s.input
}

pred S_Req_False[s:State]{
        S_Go_False[s]
}

pred S_Req_[s,s’:State]{
        S_Req_True[s] implies S_Req in s’.req else (S_Req_False[s] implies
           S_Req !in s’.req else  (S_Req in s.req  iff S_Req in s’.req))
}

// Predicates for E_Req
pred E_Req_True[s:State]{
        E_Sense in s.input
}

pred E_Req_False[s:State]{
        E_Go_False[s]
}

pred E_Req_[s,s’:State]{
        E_Req_True[s] implies E_Req in s’.req else (E_Req_False[s] implies
           E_Req !in s’.req else  (E_Req in s.req  iff E_Req in s’.req))
}

// Predicates for NS_Lock
pred NS_Lock_True[s:State]{
        N_Go_True[s] or S_Go_True[s]
```

```
}

pred NS_Lock_False[s:State]{
        (N_Go_False [s] and S_Go !in s.output) or (S_Go_False [s] and N_Go !in s.output)
}

pred NS_Lock_[s,s’:State]{
        NS_Lock_True[s] implies s’.NS_Lock = True else (NS_Lock_False[s] implies s’.NS_Lock = False else s.NS_Lock=s’.NS_Lock)
}

fact TransitionRelation{
//      all s,s’:State| (s.input = s’.input and s.output = s’.output and s.req = s’.req and s.NS_Lock = s’.NS_Lock) implies s = s’
        all s,s’:State| s’ in nextState[s] iff (N_Go_[s,s’] and S_Go_[s,s’] and E_Go_[s,s’] and N_Req_[s,s’] and S_Req_[s,s’] and E_Req_[s,s’] and NS_Lock_[s,s’])
}

// Modeling fairness constraints:

fun N_fair[]:State{
        State - (input.N_Sense & output.N_Go)
}

fun S_fair[]:State{
        State - (input.S_Sense & output.S_Go)
}

fun E_fair[]:State{
        State - (input.E_Sense & output.E_Go)
}

fact{
        fc1 = N_fair
        fc2 = S_fair
        fc3 = E_fair
}

// Helper functions for model checking

fun bound[R:State->State,X:State]
:State->State{
  X <: R
```

```
}

fun id[X:State]
:State->State{
  bound[iden,X]
}

fun loop[R: State->State]
:State{
  State.(^R & id[State])
}

// Set of states that satisfy E_CG true
// This is used for detecting fair paths
/*fun ECG_True[]:State{
        let R = *sigma, idN_fair = id[N_fair], idS_fair = id[S_fair], idE_fair = id[E_fair] |
        R.(loop[sigma] & loop[R.idN_fair.R.idS_fair.R.idE_fair.R])
}

// safety: ~ECF E_Go & (N_Go | S_Go) = ~ EF( E_Go & (N_Go | S_Go) & E_CG true)
fun safety[]:State{
        State - (*sigma).(output.E_Go & output.(N_Go + S_Go) & ECG_True)
}*/

/*
pred show[]{
        S0 in ECG_True
        some S0
}

run show for 9 State
*/

assert MC{
        CTLFC_MC[not_ctlfc[ECF[output.E_Go & output.(N_Go + S_Go)]]]
}

//Scope 7: 4.71 sec
--check MC for 7 State

//Scope 8: 36.81 sec
--check MC for 8 State

```

```
// Scope 9: 12 min 42 sec
--check MC for 9 State

// Scope 10: > 1 h
check MC for 10 State
```

B.5 Lambda Terms

```
module lambda

open util/boolean

// Backslash (\) is used to represent lambda
// In this example, each lambda term represents a state

// ct is a functions that maps each variable to its corresponding term
sig Variable {ct: Term}

// A lambda term has one of the following three types:
// VARiable, ABStraction, APPlication
abstract sig TermType{}
one sig VAR, ABS, APP extends TermType{}


// A lambda term is represented by its abstract syntax tree (AST)
// if t= v then t.var = v and t.right = t.left = NULL
// if t=\x.M then t.var = v and t.right = M and t.left = NULL
// if t=MN then t.var = NULL and t.left = M and t.right = N
sig Term{

        // type is used to represnt the type of a term
        type: TermType,

        // if type is VAR or ABS, var refers to the variable used.
        var: lone Variable,

        // right is used for ABS and APP, left is only used for APP.
        right, left: lone Term,

        // alpha conversion is modeled by AlphaC, a binary relation over terms.
        // beta reduction is modeled by BetaR, a binary relation over terms
        BetaR, AlphaC: set Term,

        // pv represents the variables that are present in term
        pv: set Variable,

        // bv represents the bounded variables of a term
        bv: set Variable,

        //fv represents the free variables of a term
        fv: set Variable,

        // sub is used to define the substitution relation
        // [t'/v]t=t'' if and only if (t -> t' -> t'' -> v) is in sub.
        sub:  Term ->  Term ->  Variable,

        //normal represents whether a term is in normal form or not
        normal: Bool
}


// This fact block is used to defined the well-formedness constraints.
fact WellFormedTerms{
        all v:Variable| v.ct.type = VAR and v.ct.var = v
        all t:Term| (t.type = VAR implies (((t.var).ct=t) and (no t.(right+left)))) and
        (t.type = ABS implies ((one t.var) and (one t.right) and (no t.left))) and
        (t.type = APP implies ((no t.var) and (one t.right) and (one t.left))) and
        (t !in t.(^(right+left)))
}

fact VARIABLES{
        all t:Term| (t.type = VAR implies t.pv = t.var) and
                (t.type!=VAR implies t.pv = t.(right+left).pv)

        all t:Term| (t.type = VAR implies (no t.bv)) and
                (t.type = ABS implies t.bv = t.right.bv + t.var) and
                (t.type = APP implies t.bv = t.right.bv + t.left.bv)

        all t:Term| t.fv = t.pv - t.bv
}


// This fact block is used to define the substitution relation (sub)
```

```
fact SUBSTITUTION{
        all t,t’,t’’:Term,v:Variable|
                t -> t’ -> t’’ -> v in sub iff (subVAR[t,t’,t’’,v] or subAPP
                    [t,t’,t’’,v] or subABS[t,t’,t’’,v])
}

pred subVAR[t,t’,t’’:Term,v:Variable]{
        t.type = VAR
        subVAR1[t,t’,t’’,v] or subVAR2[t,t’,t’’,v]
}

pred subVAR1[t,t’,t’’:Term,v:Variable]{
        t.var = v
        t’ = t’’
}

pred subVAR2[t,t’,t’’:Term,v:Variable]{
        t.var != v
        t’’ = t
}

pred subAPP[t,t’,t’’:Term,v:Variable]{
        t.type = APP
        t’’.type = APP
        t.right -> t’ -> t’’.right -> v in sub
        t.left -> t’ -> t’’.left -> v in sub
}

pred subABS[t,t’,t’’:Term,v:Variable]{
        t.type = ABS
        t’’.type = ABS
        subABS1[t,t’,t’’,v] or subABS2[t,t’,t’’,v] or subABS3[t,t’,t’’,v] or
            subABS4[t,t’,t’’,v]
}

pred subABS1[t,t’,t’’:Term,v:Variable]{
        v = t.var
        t = t’’
}

pred subABS2[t,t’,t’’:Term,v:Variable]{
        v != t.var
        v !in t.right.fv
        t = t’’
}
```

```
pred subABS3[t,t’,t’’:Term,v:Variable]{
        v != t.var
        v in t.right.fv
        t.var !in t’.fv
        t’’.var = t’.var
        t.right -> t’ -> t’’.right -> v in sub
}

pred subABS4[t,t’,t’’:Term,v:Variable]{
        v != t.var
        v in t.right.fv
        t.var in t’.fv
        some z:Variable,tt:Term|
                (t’’.var = z )and(t.right->z.ct->tt->t.var in sub)and(tt->t’->t’’.right->v in sub)
}
// End of substitution definition


// Constraint for defining alpha conversion

fact ALPHACONVERSION{
        all t,t’:Term| t->t’ in AlphaC iff ((t=t’) or alphac1[t,t’] or alphac2[t,t’] or alphac3[t,t’])
}

pred alphac1[t,t’:Term]{
        t.type = ABS
        t’.type = ABS
        t’.var !in t.right.fv
        t.right -> t’.var.ct -> t’.right -> t.var in sub
}

pred alphac2[t,t’:Term]{
        t.type = ABS
        t’.type = ABS
        t.var = t’.var
        t.right -> t’.right in AlphaC
}

pred alphac3[t,t’:Term]{
        t.type = APP
        t’.type = APP
        t.left -> t’.left in AlphaC
        t.right -> t'.right in AlphaC
}
//End of alpha

// The following is used to define the beta reduction by using
// the substitution relation
// (\x.M)N is beta reducable to [N/x]M
fact BETAREDUCTION{
        all t,t':Term|  t->t' in BetaR iff (betaABS1[t,t'] or betaABS2[t,t']
            or betaAPP1[t,t'] or betaAPP2[t,t'])
}

pred betaABS1[t,t':Term]{
        t.type=APP
        t.left.type = ABS
        t.left.right->t.right->t'->t.left.var in sub
}

pred betaABS2[t,t':Term]{
        t.type = ABS
        t'.type = ABS
        t.var = t'.var
        t.right -> t'.right in BetaR
}

pred betaAPP1[t,t':Term]{
        t.type = APP
        t'.type = APP
        t.left = t'.left
        t.right -> t'.right in BetaR
}

pred betaAPP2[t,t':Term]{
        t.type = APP
        t'.type = APP
        t.right = t'.right
        t.left -> t'.left in BetaR
}
// End of beta reduction


// A term is in normal form if it has no beta-redex
pred isInNormalForm[t:Term]{
        all st:t.(*(right+left))| st.type = APP implies st.left.type != ABS
}
```

```

fact {
        all t:Term| t.normal = True iff isInNormalForm[t]
}

// We use existential model checking to produce a term
// that does not have any normal form.
// The beta-reduction (BetaR) is used as the transition
// relation.
// In this case, a term that does not have a normal
// form can be expressed as: AG ~normal

fun bound[R: Term-> Term, X: Term]
:Term->Term{
        X <: R
}

fun id[X:Term]: Term -> Term {
        bound[iden,X]
}

fun loop[R: Term->Term]: Term{
        Term.(^R & id[Term])
}


// infinite is a term that does not have a normal form
// The output in this case is: infinite = (\x.xx)(\x.xx)
one sig infinite, infinite' in Term {}

pred IrreducableTerm []{
        infinite in Term - (*BetaR).(normal.True)

        all t:infinite.(*BetaR)| some t.BetaR
}
--run IrreducableTerm for 4 but 1 Variable

// Existential model checking is used to produce a term
// that has a normal form, but not all reduction paths
// result in term that is in normal form.
// (EF normal) and (EG ~normal) is used to express this property.
// One of the output is (\x.(\x.xx))((\x.xx)(\x.xx))

pred OrderInReducing[]{
        infinite in Term - (*BetaR).(normal.True)
        all t:infinite.(*BetaR)| some t.BetaR

        let R = bound[BetaR,normal.False] |
                infinite’ in ((*BetaR).(normal.True)) & ((*R).(loop[R]))
}

run OrderInReducing for 6 but 1 Variable
//run OrderInReducing for 10 but 3 Variable
```

References

[1] Fourth Hardware Model Checking Competition. http://fmv.jku.at/hwmcc11/hwmcc11.pdf.

[2] Python. http://www.python.org.

[3] Yael Abarbanel-Vinov, Neta Aizenbud-Reshef, Ilan Beer, Cindy Eisner, Daniel Geist, Tamir Heyman, Iris Reuveni, Eran Rippel, Irit Shitsevalov, Yaron Wolfsthal, and Tali Yatzkar-Haham. On the Effective Deployment of Functional Formal Verification. *Formal Methods in System Design*, page 3544, 2001.

[4] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, New York, 1996.

[5] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings Publishing Co., Inc., 1991.

[6] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, New York, 2010.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press; Cambridge, MA, 2008.

[8] Clark Barrett, ChristopherL. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer-Aided Verification*, Lecture Notes in Computer Science, pages 171–177. Springer, 2011.

[9] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[11] Shoham Ben-David, Richard Trefler, and Grant Weddell. Model Checking Using Description Logic. *Journal of Logic and Computation*, pages 111–131, 2010.

[12] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving Existentially Quantified Horn Clauses. Computer-Aided Verification, pages 869–882. Springer, 2013.

[13] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness Checking as Safety Checking. In *FMICS02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier Science Publishers, B.V.; Amsterdam, 2002.

[14] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[15] Egon Börger. The ASM Method for System Design and Analysis. A Tutorial Introduction. In *Frontiers of Combining Systems*, Lecture Notes in Computer Science, pages 264–283. Springer, 2005.

[16] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[17] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 1997.

[18] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, pages 428–439, Jun 1990.

[19] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1990.

[20] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Computer-Aided Verification*, pages 334–342. 2014.

[21] Ernest Chang and Rosemary Roberts. An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processes. *Communications of the ACM*, pages 281–283, 1979.

[22] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic Model Checking of Declarative Relational Models. In *International Conference on Software Engineering*, pages 312–320, 2006.

[23] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, pages 345–363, 1936.

[24] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer-Aided Verification*, Lecture Notes in Computer Science, pages 241–268. Springer, 2002.

[25] Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press; Cambridge, MA, 1999.

[26] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer, 1982.

[27] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, page 217232, 1995.

[28] Martin Davis, Ron Sigal, and Elaine J Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science.* Morgan Kaufmann, 2 edition, 1994.

[29] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[30] Giuseppe Del Castillo and Kirsten Winter. Model Checking Support for the ASM High-Level Language. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 331–346. Springer, 2000.

[31] A. Dold. A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, 1998.

[32] Alma L. Juarez Dominguez. *Detection of Feature Interactions in Automotive Active Safety Features*. PhD thesis, Cheriton School of Computer Science, University of Waterloo, May 2012.

[33] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 333–336. Springer, 2004.

[34] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, pages 85–107, 1997.

[35] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with Actions. In *International Conference on Software Engineering*, pages 442–451, 2005.

[36] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[37] Curt Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, 1929. Proof of completenss theorem for FOL.

[38] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, 2009.

[39] Gerard J Holzmann. The Theory and Practice of A Formal Method: NewCoRe. In *IFIP Congress (1)*, pages 35–44. 1994.

[40] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, may 1997.

[41] Michael Huth and Mark Ryan. *Logic in Computer Science, Modelling and Reasoning about Systems*. Cambridge University Press, New York, second edition, 2004.

[42] Neil Immerman and Moshe Vardi. Model Checking and Transitive-Closure Logic. In *Computer-Aided Verification*, Lecture Notes in Computer Science, pages 291–302. Springer, 1997.

[43] Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. 11(2):256–290, 2002.

[44] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis.* The MIT Press; Cambridge, MA, 2006.

[45] Yonit Kesten and Amir Pnueli. A compositional approach to CTL* verification. *Journal of Theoretical Computer Science*, pages 397 – 428, 2005.

[46] R. P. Kurshan and K. McMillan. A Structural Induction Theorem for Processes. pages 239–247. ACM, 1989.

[47] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, Lecture Notes in Computer Science, page 855874. Springer, 2003.

[48] Michael Leuschel and Michael Butler. ProB : an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, pages 185–203, 2008.

[49] K. L. McMillan. The SMV system, November 06 1992.

[50] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[51] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4.5):567–576, July 1997.

[52] Tobias Schüle and Klaus Schneider. Bounded model checking of infinite state systems. *Formal Methods in System Design*, pages 51–81, 2007.

[53] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2000.

[54] Amirhossein Vakili and Nancy A. Day. Avestan: A declarative modeling language based on SMT-LIB. In *ICSE Workshop on Modeling in Software Engineering (MISE)*, pages 36–42. June 2012.

[55] Amirhossein Vakili and Nancy A. Day. Temporal Logic Model Checking in Alloy. In *ASM, Alloy, B, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2012.

[56] Amirhossein Vakili and Nancy A. Day. Reducing CTL-live Model Checking to First-Order Logic Validity Checking. In *Formal Methods in Computer-Aided Design*, pages 34:215–34:218. FMCAD Inc., 2014.

[57] Amirhossein Vakili and Nancy A. Day. Reducing CTL-live Model Checking to Semantic Entailment in First-Order Logic (Version 1). Technical Report CS-2014-05, Cheriton School of Comp. Sci., University of Waterloo, 2014.

[58] Amirhossein Vakili and Nancy A. Day. Verifying CTL-Live Properties of Infinite State Models Using an SMT Solver. In *Foundations of Software Engineering*, FSE 2014, pages 213–223. 2014.

[59] Moshe Y. Vardi and Pierre Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115:1–37, 1994.

[60] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, pages 68–80. Springer, 1990.

[61] Pierre Wolper, M.Y. Vardi, and A.Prasad Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 24th Annual Symposium on*, pages 185–194, Nov 1983.

[62] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., 1996.