Verifying CTL-Live Properties of Infinite State Models using an SMT Solver

Amirhossein Vakili and Nancy A. Day
Cheriton School of Computer Science
University of Waterloo
Ontario, Canada, N2L 3G1
{avakili,nday}@uwaterloo.ca

ABSTRACT

The ability to create and analyze abstract models is an important step in conquering software complexity. In this paper, we show that it is practical to verify dynamic properties of infinite state models expressed in a subset of CTL directly using an SMT solver without iteration, abstraction, or human intervention. We call this subset CTL-Live and it consists of the operators of CTL expressible using the least fixed point operator of the mu-calculus, which are commonly considered liveness properties (e.g., **AF**, **AU**). We show that using this method the verification of an infinite state model can sometimes complete more quickly than verifying a finite version of the model. We also examine modelling techniques to represent abstract models in first-order logic that facilitate this form of model checking.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal logic*

General Terms

Verification

Keywords

CTL-Live, First-order logic, Infinite state model, Model checking, SMT solver

1. INTRODUCTION

Abstraction is a key element to conquering complexity in the development of software [21,24]. We need tools that support reasoning about abstract models of systems in order to better understand our models and to detect errors earlier in the development process. Abstract models are often expressed using infinite or complex data structures. Temporal logic model checking [14] of the dynamic behaviour of models with infinite state spaces without the use of abstraction is usually considered beyond the realm of first-order logic (FOL) reasoners because of the iterative nature of the fixed point (or transitive closure) computation. However, with the recent advances in SMT (satisfiability modulo theories) solvers that have turned first-order reasoners into powerful, efficient verification tools, it is worth taking another look at the problem of how to express the temporal logic model checking problem in FOL. Some results use an SMT solver iteratively to analyze invariants of infinite state systems (e.g., [9,10]). These methods are guaranteed to terminate without approximation only if the property is not satisfied.

In recent work [30], we showed that the validity of properties within a subset of the temporal logic CTL (Computation Tree Logic) can be expressed in FOL directly without the use of iteration. We called this subset CTL-Live, and it consists of operators that are commonly used to describe liveness properties, i.e., those expressible using the least-fixed point operator of the mu-calculus (e.g., **AF**, **AU**). We also showed that CTL-Live is maximal with respect to FOL in the sense that CTL operators that are not within CTL-Live (e.g., invariants) are not expressible in FOL [31].

Our FOL theory for CTL-Live creates the possibility of the following practical use: model the system as a (potentially infinite) Kripke structure in FOL, 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).

We evaluate the practical application of this theory through a set of case studies. We address three open 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?

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635911

```
c : Integer
Initial condition:
        c := 0;
P1:                 P2:
    c := c + 2;         c := c + 3;
```

Figure 1: A simple counter

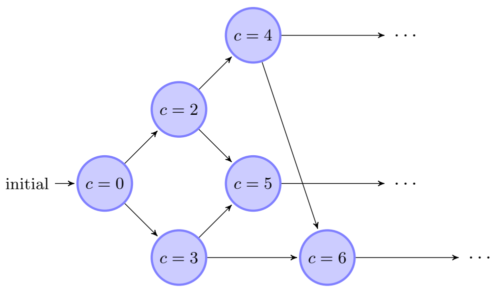

Figure 2: The Kripke structure represented by the system in Figure 1

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 [17] and CVC4 [5]. 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 Alloy [20] and Cadence SMV [26]. Throughout the paper, we consider questions regarding modelling techniques to facilitate this form of dynamic analysis.

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.

We believe our results regarding the model checking of infinite state systems automatically are an exciting step forward in the quest to provide automatic reasoning tools for abstract models of dynamic systems.

The rest of this paper is organized as follows: Section 2 provides the background material needed to understand our results. Section 3 describes the process and the chain of tools that we use to verify our case studies. The case studies are presented in Section 4. Section 5 discusses modelling choices that have an effect on the performance of the tools we use. Section 6 presents our theoretical result on the method of finding inductive invariants for model checking safety properties. Section 7 describes related work, and Section 8 concludes the paper.

2. BACKGROUND

In this section, we briefly present the background concepts needed to understand our results. A Kripke structure is a basic way of modelling the dynamic behaviour of a system. A *Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, \mathbb{P} \rangle$ is a four tuple, where $\mathcal{S}$ is a set of states, $\mathcal{S}_0$ is a non-empty subset of $\mathcal{S}$ called the initial states, $\mathcal{N}$ is a total binary relation over $\mathcal{S}$ called the transition relation, and $\mathbb{P}$ is a set of labelling predicates, where each labelling predicate is a subset of $\mathcal{S}$. A Kripke structure is infinite if and only if its set of states is infinite. Kripke structures are used to define the semantics of temporal logics.

Computation tree logic (CTL) is a temporal logic that allows us to describe properties over possible computation paths of a system [15]. CTL contains all the logical connectives of propositional logic and a set temporal connectives. Each temporal connective consists of two parts, a *path quantifier* and a *state quantifier*. The path quantifiers are **A** (for all) and **E** (exists). The state quantifiers are **X** (next state), **F** (eventually), **G** (globally), and **U** (strong until). The syntax of CTL is defined for a given set of labelling predicates $\mathbb{P}$:

$$\begin{aligned} \varphi \quad &::= \quad P \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \ \text{ where } P \in \mathbb{P} \\ &::= \quad \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{AG}\varphi \\ &::= \quad \varphi_1\mathbf{AU}\varphi_2 \mid \varphi_1\mathbf{EU}\varphi_2 \end{aligned} \tag{1}$$

A Kripke structure defines a set of computation paths, where each path represents a trace of execution. A computation path starting at state $s \in S$ is an infinite sequence of states, $s_0 \mapsto s_1 \mapsto \ldots$ such that $s = s_0$ and for every $i \geq 0$, $\mathcal{N}(s_i, s_{i+1})$. 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. The satisfiability relation for CTL, $\Vdash_c$, is defined by structural induction on φ:

$\mathcal{K}, s \Vdash_c P$	$\iff$	$P(s)$ holds, where $P \in \mathbb{P}$
$\mathcal{K}, s \Vdash_c \neg\varphi$	$\iff$	$\mathcal{K}, s \nVdash_c \varphi$
$\mathcal{K}, s \Vdash_c \varphi_1 \wedge \varphi_2$	$\iff$	$\mathcal{K}, s \Vdash_c \varphi_1$ and $\mathcal{K}, s \Vdash_c \varphi_2$
$\mathcal{K}, s \Vdash_c \mathbf{EX}\varphi$	$\iff$	$\exists s' \in S : \mathcal{N}(s, s') \wedge \mathcal{K}, s' \Vdash_c \varphi$
$\mathcal{K}, s \Vdash_c \mathbf{AF}\varphi$	$\iff$	for all paths $s_0 \mapsto s_1 \mapsto \ldots$ such that $s_0 = s$ there exists an i such that $\mathcal{K}, s_i \Vdash_c \varphi$.
$\mathcal{K}, s \Vdash_c \varphi_1\mathbf{EU}\varphi_2$	$\iff$	there exists a j and a path, $s_0 \mapsto s_1 \mapsto \ldots$, such that $s = s_0$, $\mathcal{K}, s_j \Vdash_c \varphi_2$, and for all $i < j$ $\mathcal{K}, s_i \Vdash_c \varphi_1$.

The connectives above form a complete fragment of CTL, e.g., $\mathbf{EG}\varphi$ is equivalent to $\neg\mathbf{AF}\neg\varphi$. A Kripke structure $\mathcal{K}$ satisfies the CTL formula φ, denoted by $\mathcal{K} \Vdash_c \varphi$, iff for all $s \in \mathcal{S}_0$ we have $\mathcal{K}, s \Vdash_c \varphi$.

EXAMPLE 1. Figure 1 represents a simple asynchronous counter system. There are two processes in this system, `P1` and `P2`. They both have access to a shared variable of type integer, `c`. At each moment in time, one of the processes changes the value of `c`: `P1` increments `c` by `2` units and `P2` by `3` units. In this asynchronous system, the order in which the processes are executed is not known; as a result, when `c=100`, in the next step `c=102` or `c=103` depending on which process has been executed. The initial value of `c` in this system is `0`. We want to use CTL to study the behaviour of this system. Figure 2 represents a part of the Kripke structure of the counter system in Figure 1. The set of states of this Kripke structure $\mathcal{K}$ is the set of all positive integers, $\mathcal{S} =$ `Integer`. This Kripke structure has only one

```
1)  (declare-fun Init (Int) Bool)
2)  (declare-fun Next (Int Int) Bool)
3)  (define-fun P1 ((c Int) (cn Int)) Bool (= cn (+ c 2)))
4)  (define-fun P2 ((c Int) (cn Int)) Bool (= cn (+ c 3)))
5)  (assert (forall ((c Int)) (= (Init c) (= c 0))))
6)  (assert (forall ((c Int) (cn Int)) (= (Next c cn) (or (P1 c cn) (P2 c cn)))))
```

Figure 3: SMT-LIB specification of the model in Figure 1

Table 1: Satisfiability Relations

Notation	Description
$\mathcal{K} \Vdash_c \varphi$	Kripke structure $\mathcal{K}$ satisfies CTL formula φ.
$\mathit{symbolic}(K) \models_c \varphi$	Symbolic Kripke structure $\mathit{symbolic}(K)$ satisfies CTL formula φ.
$\Gamma \models \Phi$	FOL formula Φ is valid with respect to the set Γ.

initial state, $\mathcal{S}_0 = \{0\}$, and the transition relation satisfies the following property:

$$(c, c') \in \mathcal{N} \quad \text{iff} \quad c' = c + 2 \;\vee\; c' = c + 3$$

This Kripke structure satisfies **EF** `c=5`, i.e., $\mathcal{K} \Vdash_c$ **EF** `c=5`, since there exists a path from the initial state to a state where `c` becomes `5`; on the other hand $\mathcal{K} \nVdash_c$ **AF** `c=5`, since not all paths eventually make `c` equal to `5`. If process `P1` is executed for the first 3 steps, `c` never becomes `5`. Since each process increases the value of `c`, we can see that $\mathcal{K} \Vdash_c$ **AF** `c>5` holds.

2.1 Kripke Structures in FOL and SMT Solvers

First-order logic (FOL) provides quantifiers along with propositional logic connectives to describe properties over *relational* and *functional symbols* [19]. An *interpretation* determines the content of each relational and functional symbol. A problem in FOL consists of a set of formulae. Every interpretation that satisfies all these formulae is called a satisfying interpretation. Using the concept of satisfiability, validity is defined as follows:

Definition 1. ***(Validity)*** *Suppose Γ is a set of FOL formulae and Φ is a FOL formula: Γ entails Φ, denoted by $\Gamma \models \Phi$, iff every interpretation that satisfies all the formulae in Γ also satisfies Φ.*

From this definition, we can prove that $\Gamma \models \Phi$ iff $\Gamma \bigcup \{\neg\Phi\}$ is unsatisfiable.

A satisfiability modulo theories (SMT) solver, for short SMT solver, is an automatic tool to check the satisfiability of a set of FOL formulae [3]. 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 [4]. 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[1], 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.

Example 2. Figure 3 presents a symbolic representation of the Kripke structure of Figure 2 as an SMT-LIB specification. This specification does not contain user-defined types. Lines 1 and 2 declare that `Init` and `Next` are relational symbols over `Int` and `Int` $\times$ `Int` respectively. To increase the readability of this specification, we have defined `P1` and `P2` in Lines 3 and 4. A definition is essentially a macro. In Lines 3-6, `c` represents the current state and `cn` the value of `c` in the next state. Line 5 is a constraint stating that the state `c` is an initial state iff it is equal to zero: `(= c 0)`. The constraint in Line 6 states that `cn` is the next value of `c` iff either `P1` holds between them or `P2`.

As this example suggests, to symbolically represent a Kripke structure in FOL, we need at least two relational symbols: `Init` representing the set of initial states and `Next` representing the transition relation. The types of `Init` and `Next` are `S -> Bool` and `S`$\times$`S -> Bool` respectively. In these declarations, `S` depends on the types of the variables used in the specification.

The symbolic representation of a Kripke structure $\mathcal{K}$, which we denote by $\mathit{symbolic}(K)$, is a set of FOL formulae that defines $\mathcal{K}$. The formula in Line 6 of Figure 3 in infix form using the classical symbols for FOL connectives is the following:

$$\forall \texttt{c,cn:Int}.\ \texttt{Next(c,cn)} \;\Leftrightarrow\; \texttt{P1(c,cn)} \vee \texttt{P2(c,cn)}$$

In this case, the transition relation is uniquely defined. However, in general, a symbolic Kripke structure can represent a set of Kripke structures, rather than a single one. Underspecification of the transition relation and the use of user-defined types and operations that are not fully interpreted are the main reasons that a symbolic Kripke structure can represent multiple Kripke structures. In all our case studies, we uniquely defined the transition relation.

We define $\mathit{symbolic}(K) \models_c \varphi$ to mean that every satisfying interpretation $\mathcal{K}$ of $\mathit{symbolic}(K)$ satisfies the CTL formula φ:

$$\mathit{symbolic}(K) \models_c \varphi$$
$$\text{iff}$$
$$\text{every } \mathcal{K} \text{ that satisfies } \mathit{symbolic}(K) \text{ also } \mathcal{K} \Vdash_c \varphi$$

If $\mathit{symbolic}(K)$ has only one satisfying interpretation up to isomorphism, namely $\mathcal{K}$, then $\mathit{symbolic}(K) \models_c \varphi$ is equivalent to $\mathcal{K} \Vdash_c \varphi$. Table 1 is a summary of the satisfiability notations used in this paper.

[1] A relational symbol is a functional symbol of type `Bool`.

`CTLL2FOL(`φ`):`
`case` φ `of`

1)	P	->	$\{\}$ where P is a labelling predicate
2)	$\neg\psi$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \neg\lceil\psi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ)
3)	$\psi_1 \vee \psi_2$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \lceil\psi_1\rceil(s) \vee \lceil\psi_2\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ_1) $\bigcup$ CTLL2FOL(ψ_2)
4)	$\psi_1 \wedge \psi_2$	->	$\{\ \forall s:\ \lceil\varphi\rceil(s) \leftrightarrow \lceil\psi_1\rceil(s) \wedge \lceil\psi_2\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ_1) $\bigcup$ CTLL2FOL(ψ_2)
5)	$\mathbf{EX}\psi$	->	$\{\ \forall s:\ \big(\exists s':\ N(s,s') \wedge \lceil\psi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ)
6)	$\mathbf{AX}\psi$	->	$\{\ \forall s:\ \big(\forall s':\ N(s,s') \rightarrow \lceil\psi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ)
7)	$\mathbf{EF}\psi$	->	$\{\ \lceil\psi\rceil \subseteq \lceil\varphi\rceil\ ,\quad \forall s: \big(\exists s': N(s,s') \wedge \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ)
8)	$\mathbf{AF}\psi$	->	$\{\ \lceil\psi\rceil \subseteq \lceil\varphi\rceil\ ,\quad \forall s: \big(\forall s': N(s,s') \rightarrow \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ)
9)	$\psi_1\mathbf{EU}\psi_2$	->	$\{\ \lceil\psi_2\rceil \subseteq \lceil\varphi\rceil\ ,\quad \forall s: \lceil\psi_1\rceil(s) \wedge \big(\exists s': N(s,s') \wedge \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ_1) $\bigcup$ CTLL2FOL(ψ_2)
10)	$\psi_1\mathbf{AU}\psi_2$	->	$\{\ \lceil\psi_2\rceil \subseteq \lceil\varphi\rceil\ ,\quad \forall s: \lceil\psi_1\rceil(s) \wedge \big(\forall s': N(s,s') \rightarrow \lceil\varphi\rceil(s')\big) \rightarrow \lceil\varphi\rceil(s)\ \} \bigcup$ CTLL2FOL(ψ_1) $\bigcup$ CTLL2FOL(ψ_2)

Figure 4: The definition of CTLL2FOL **(from [30]).** φ **is a CTL-Live formula.**

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 5: CTL-Live

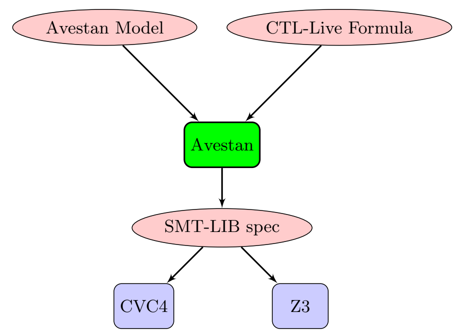

Figure 6: Overview of our method

2.2 CTL-Live Verification as a FOL Theory

In recent work [30,31], we presented a subset of CTL that we called CTL-Live and described how to represent the verification of a CTL-Live property as a validity problem in FOL. CTL-Live is presented in Figure 5. The grammar of CTL-Live does not allow a temporal connective to be within the scope of negation, e.g., the formula $\mathbf{AF}\neg\varphi$ is part of CTL-Live, but $\neg\mathbf{AF}\varphi$ is not.

CTL-Live includes the CTL connectives whose semantics in the mu-calculus are defined using the *least fixed-point* operator. The intuition behind reducing CTL-Live model checking to FOL validity checking is that model checking is about verifying whether the set of initial states is included in the set of states that satisfies a property. If the CTL property under study is expressible as the *smallest* set that satisfies some FOL formulae, then checking whether the set of initial states is a subset of the smallest one is equivalent to checking whether the set of initial states is a subset of *all* of them:

$$S_0 \subseteq \bigcap_{X \in \Theta} X \quad \text{iff} \quad S_0 \subseteq X \quad \text{for every} X \in \Theta$$

In this equation, Θ contains all the sets that satisfy some property and $\bigcap_{X \in \Theta} X$ is the smallest one. This property has a higher-order quantifier over sets, which is not available in FOL, but it is implicitly available in the quantification over interpretations in the definition of validity in FOL (Definition 1).

To model check a symbolic Kripke structure *symbolic*(K), and a CTL-Live formula φ, we use a function called CTLL2FOL, shown in Figure 4. The function CTLL2FOL recurses over the structure of φ and generates a set of FOL formulae. In Figure 4, $\lceil\varphi\rceil$ is a new relational symbol that is introduced by CTLL2FOL for the formula φ; for a labelling predicate P, $\lceil P\rceil$ is equal to P. The complexity of CTLL2FOL is linear with respect to the size of φ.

The following theorem allows us to reduce CTL-Live model checking to validity checking in FOL:

THEOREM 1. ***(Model checking CTL-Live)*** *Let* symbolic(K) *be a set of FOL formulae that specifies a Kripke structure(s), we have:*

$$\text{symbolic}(K) \models_c \varphi \iff \text{symbolic}(K) \bigcup \texttt{CTLL2FOL}(\varphi) \models S_0 \subseteq \lceil\varphi\rceil$$

where $\lceil\varphi\rceil$ *is a relational symbol generated by* CTLL2FOL *and* S_0 *is a predicate describing the initial set of states [30].*

3. METHOD

Our theoretical result regarding the ability to express the verification of CTL-Live properties in FOL makes it possible to turn a CTL-Live verification problem into a problem that can be directly solved by an SMT solver without iteration or human intervention. The approach that we use to implement our method is described in Figure 6.

A model is created in FOL using a tool that we call Avestan. Our current version of Avestan is a complete reengineering of our earlier tool [29] (also called Avestan), which was a language and tool to support the creation of models in

```
1)  (declare-fun af (Int) Bool)
2)  (assert (forall ((c Int)) (=> (> c 5) (af c))))
3)  (assert (forall ((c Int)) (=> (forall ((cn Int)) (=> (Next c cn) (af cn))) (af c))))
4)  (assert (not (forall ((c Int)) (=> (Init c) (af c)))))
```

Figure 7: Declarations and formulae that are added to Figure 3 to model check AF `c>5`

SMT-LIB. It was strongly based on Alloy [20], but the tool translated the model into an SMT-LIB specification. Our new tool is implemented in Python3 [1] 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 `CTLL2FOL` of Figure 4 in Avestan to create the constraints needed for the verification of a CTL-Live property. Using Avestan, 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). The following is an example that illustrates the method of applying the result of Theorem 1 to verify a declarative model using an SMT solver.

Example 3. Suppose we want to prove that in the Kripke structure of Figure 2, `c` eventually becomes larger than 5, by using an SMT solver. We need to prove that this Kripke structure satisfies **AF** `c>5`. **AF** is part of CTL-Live, therefore, we can use the result of Theorem 1. According to this theorem, we need to compute `CTLL2FOL(`**AF** `c>5)`. Following the definition of `CTLL2FOL` at Line 8, `CTLL2FOL(`**AF** `c>5)` is a set with two constraints:

1. $\forall s : \lceil\psi\rceil(s) \Rightarrow \lceil\varphi\rceil(s)$
2. $\forall s : \big(\forall s' : N(s,s') \Rightarrow \lceil\varphi\rceil(s')\big) \Rightarrow \lceil\varphi\rceil(s)$

where $\lceil\psi\rceil$ is `c>5`, N is `Next` (of Figure 3), and $\lceil\varphi\rceil$ is a new relational symbol `af` of type `Int ->Bool`. Since the state of the system is represented by an integer, the quantification over states becomes quantification over integers. Written in terms of the model and property, these constraints are:

1. $\forall$`c:Int. c>5  ⇒  af(c)`
2. $\forall$`c:Int.` $\big(\forall$`cn. Next(c, cn)  ⇒  af(cn)`$\big) \Rightarrow$ `af(c)`

Now, we need to check whether
$symbolic(K) \bigcup$ `CTLL2FOL(`**AF** `c>5)` entails the following:

$$\forall \texttt{c:Int. Init(c)} \;\Rightarrow\; \texttt{af(c)} \qquad (2)$$

We know that Γ entails Φ iff $\Gamma \bigcup \{\neg\Phi\}$ is unsatisfiable; therefore, we add the **negation** of the formula in Equation 2 to $symbolic(K) \bigcup$ `CTLL2FOL(`**AF** `c>5)` and run the SMT solver to check for the satisfiability: if it is unsatisfiable, then we can conclude that **AF** `c>5` holds. Figure 7 presents the declaration of `af` (Line 1), along with the three formulae (Lines 2-4) in SMT-LIB notation that need to be added to Figure 3 to model check **AF** `c>5`. The output of Z3 on this model is `unsat`, which tells us that the original model satisfies **AF** `c>5`.

4. CASE STUDIES

In this section, we present four case studies that test whether it is possible to use our theory and method to verify dynamic properties in CTL-Live of abstract models using an off-the-shelf SMT solver. Our models were chosen from a variety of sources and domains. As we present each case study, we discuss how it is modelled in FOL and, when possible, we compare to how it was modelled and verified previously.

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 paper are available on-line[2].

4.1 Case Study 1: Leader Election Protocol

The leader election model is a protocol to "elect" a process as the leader among a finite set of processes that form a ring [11]. A finite instance of it was previously verified by Jackson using the Alloy Analyzer [21]. 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 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' 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:

AF `(my_lead_i = lead_id)`

where `lead_id` is the largest ID among the current processes, `my_lead_i` the value of `my_lead` for the i^{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:

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

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

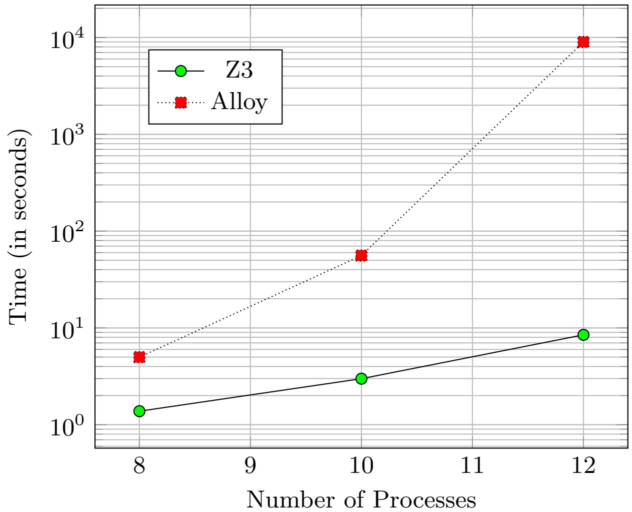

Figure 8: Leader election model: Z3 vs Alloy

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 [21]. 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 8, 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.

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) [7]. 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 guarantees that `Int` has an infinite set of elements, our technique does not produce spurious counterexamples.

4.2 Case Study 2: Bakery Algorithm

The bakery algorithm ensures mutual exclusion between two processes that run concurrently and asynchronously [9]. Bultan, Gerber, and Pugh verified that in this algorithm the two processes cannot get into their critical sections at the same time [9]. 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 `{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:

```
1)  (declare-sort ControlState 0)
2)  (declare-fun T () ControlState)
3)  (declare-fun W () ControlState)
4)  (declare-fun C () ControlState)
5)  (assert (not (= T W)))
6)  (assert (not (= T C)))
7)  (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:

$$\mathbf{AG}\left(\texttt{c1 = W} \Rightarrow \mathbf{AF}\,\texttt{c1 = C}\right) \tag{3}$$

This is an invariant property, therefore to verify this property, we needed to show that every reachable state satisfies `c1 = W` $\Rightarrow$ **AF** `c1 = C`. **AG** is not part of CTL-Live, therefore we cannot ask the SMT solver to prove this property directly. Instead, we created a more general property that implies the formula of Equation 3: we proved that the set of *all* states, which includes the reachable states, satisfies the following property:

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

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 paths of the model must be infinite and only those must be considered. Rather than making the transition relation total, we introduced the idea of a "dead-end" state, which is one 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.

Another algorithm studied by Bultan, Gerber, and Pugh is the ticket mutual exclusion algorithm [2, 9]. We tried to verify an invariant property similar to Equation 3 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

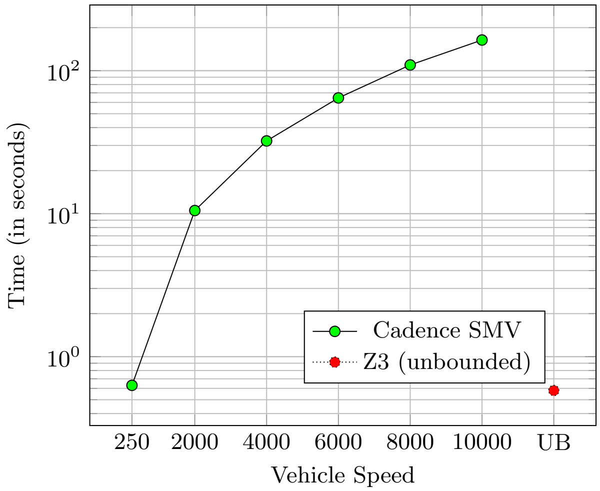

Figure 9: Collision Avoidance model in Cadence SMV (UB: UnBounded)

satisfied within the reachable set of states and therefore it does not hold for the entire set of states.

4.3 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 [18, 22]. 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 9 presents these results. As this figure shows, the performance of Cadence SMV degrades as the size of speed and threshold is increased.

4.4 Case Study 4: File System

Our last case study is a file system that was originally modelled in Z [32]. 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 a 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 [21], in our model, we explicitly introduced the state space as a new uninterpreted set `State` and declared `content` as follows:

$$\texttt{content:}\ \ \texttt{State}\ \times\ \texttt{Key -> Data}$$

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 "State" object 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:

$$\texttt{content(s, k)}\ \neq\ \texttt{NULL}\ \Rightarrow\ \exists s'\texttt{: delete(k, s, }s'\texttt{)} \tag{4}$$

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 a 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* [21]. 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 4 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 created two models with the same state space: one that includes all operations (model #1) and one that includes only `add` and `delete` (model #2). We assume that some state `s2` is the result of writing something to the file system at some state `s1`; then, we check in model #2 that `s2` is reachable from `s1`:

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

Z3 verified this property in 0.15 seconds and CVC4 in 0.69 seconds.

4.5 Conclusions

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

Table 2: 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

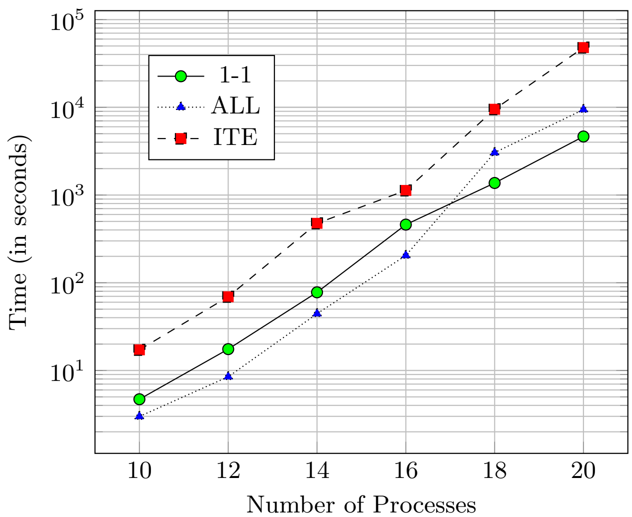

Figure 10: Z3 on different models for the leader election problem

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 2 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. MODELLING OPTIMIZATIONS

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.

The first approach adds $2 \times n$ 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 10 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. Plot ALL of Figure 10 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[3]. 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 `CTLL2FOL`, the eventuality property can be expressed using an existential quantifier as in:

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

Since Alloy's input language is as expressive as FOL, we can use our `CTLL2FOL` 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 11 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.

6. INDUCTIVE INVARIANTS

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 $\mathbf{AG}\ P$. $\mathbf{AG}$ is not part of CTL-Live and previously we proved that its model checking is not reducible to FOL entailment checking:

THEOREM 2. ***(Maximality of CTL-Live)*** *CTL-Live is the largest fragment of CTL that its model checking is reducible to entailment checking in FOL; in other words, the temporal part of CTL-Live cannot be extended with* **EG**, **AG**, *or* $\neg$ *for model checking a symbolic Kripke structure in*
FOL [31].

In our proof of this theorem, we showed that the complement of the halting problem on an empty tape for a deterministic Turing machine (DTM) is reducible to universal model

[3]Since the Alloy models were analyzed using the ALL approach, in Section 4.1, we have reported the results of the ALL approach using Z3 even though the 1-1 approach performs better.

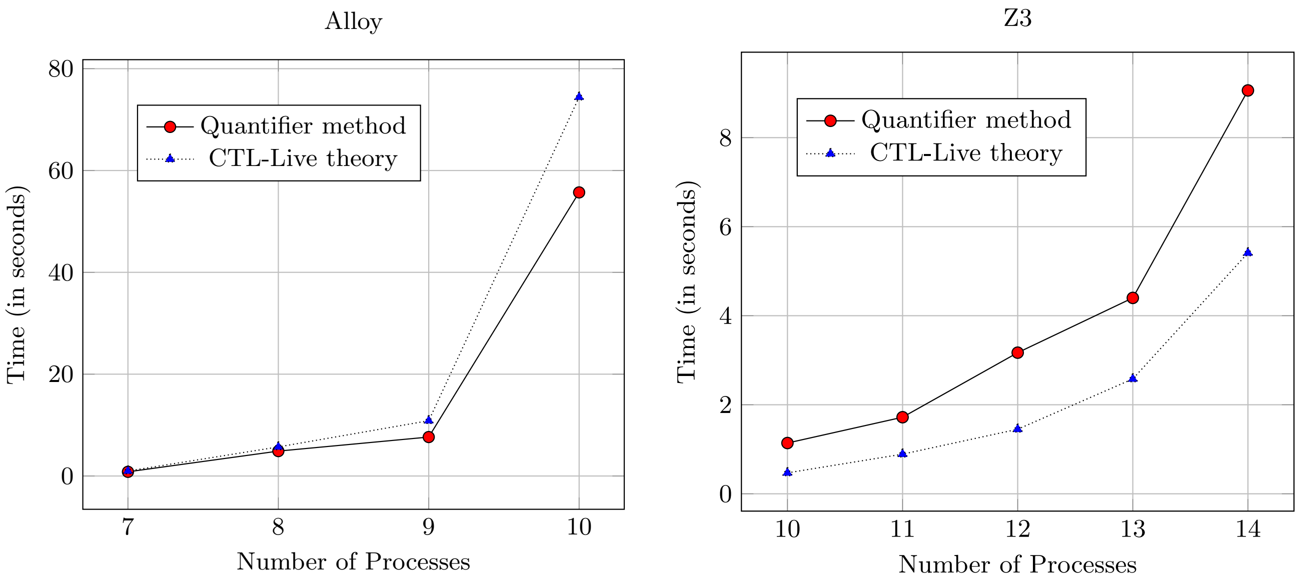

Figure 11: Alloy and Z3 with different approaches for the leader election case study

checking of **EG** (**AG**). The complement of the halting problem is not recursively enumerable, and as a result, it cannot be reduced to entailment 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 included in CTL-Live, $\neg$ cannot be added as well. Therefore, the verification of an invariant cannot be done by an SMT solver directly and alternative techniques are needed. A common approach to this problem is to find an inductive invariant. A property P is an *inductive invariant* for a Kripke structure $\mathcal{K}$ iff it satisfies the following two constraints:

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

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 2, 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 results have found inductive invariants by hand. The method of IC3 [8] is a way to find automatically inductive invariants for finite systems, and this approach has been generalized in nuXmv [10] in an incomplete approach to finding 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.

A 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 3. ***(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. *We have shown that proving a DTM does* not *halt on an empty tape is reducible to proving that a formula named $\neg$halt is an invariant [31]. 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.* □

7. RELATED WORK

SAT and SMT solvers have been used for bounded model checking [7, 27]. These methods use a reasoner directly for model checking by expanding the transition relation for a finite number of steps.

K-induction is a technique for unbounded model checking of safety properties [28]. This technique extends bounded model checking by proving that bounded model checking for the 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 [28], termination is guarantied due to the finiteness of the Kripke structures under study.

Bultan, Gerber, and Pugh use Presburger formulae to represent infinite sets of states symbolically [9]. Their model checking approach for invariants requires a fix-point calculation, and termination is achieved by using conservative approximation. This approach allows false negatives.

Recently, IC3 has been generalized using SMT solvers to verify safety properties of infinite systems iteratively [12,13]. These approaches also incorporate abstraction techniques to gain better performance.

Based on the deductive system of Kesten and Pnueli [23], Beyene, Popeea, and Rybalchenko encoded CTL model checking of infinite state systems into forall-exists quantified Horn clauses [6]. The contribution of [6] is to develop a solver for forall-exists quantified Horn clauses and demonstrate its use for model checking CTL properties. Their method requires the models and the model checking constraints to be expressed in forall-exists quantified Horn clauses 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.

In comparison to these approaches, our approach does not require using an SMT solver iteratively, but it is only applicable to a subset of CTL. Also, our approach is theoretically guaranteed to terminate when the property is valid, whereas the other approaches terminate when the property is not satisfied.

Compositional model checking [25] and abstraction [16] are techniques that can be applied to model check an infinite state system. Our approach could be used along with these techniques to verify safety and liveness properties of an infinite system.

8. CONCLUSION

In this paper, 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) Kripke structure in FOL, a theory of FOL constraints is automatically generated based on the property, and the problem is given to an SMT solver to solve by itself. The decidability of analysis is based on the subset of FOL used to express the model. 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. Finally, we proved that inductive invariants do not always exist for safety properties of infinite state systems.

In the future, we plan to investigate the scalability of our approach for larger models. However, even though the textual size of our case studies are all fairly small, the modelling efficiencies gained by representing systems abstractly offset somewhat concerns with respect to scalability. We are also exploring less primitive ways to write FOL models, the analysis of models with richer data types and quantifiers, and ways to more easily understand counterexamples produced by SMT solvers for temporal logic properties.

9. ACKNOWLEDGEMENTS

We thank the reviewers for their insightful and detailed comments, which have improved our paper.

10. REFERENCES

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

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

[3] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[4] C. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard Version 2.0 Reference Manual*, Jan. 2010.

[5] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302. Springer-Verlag, 2007.

[6] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. CAV, pages 869–882. Springer, 2013.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[8] A. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer Berlin Heidelberg, 2011.

[9] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In *CAV*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[10] R. Cavada, A. Cimatti, M. Dorigatti, A. Mariotti, A. Micheli, S. Mover, A. Griggio, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. Technical report, Feb. 2014.

[11] E. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processes. *CACM*, 22(5):281–283, 1979.

[12] A. Cimatti and A. Griggio. Software Model Checking via IC3. In *CAV*, volume 7358 of *LNCS*, pages 277–293. Springer Berlin Heidelberg, 2012.

[13] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. *CoRR*, abs/1310.6847, 2013.

[14] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS*, pages 244–263, 1986.

[16] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[17] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

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

[19] M. Huth and M. Ryan. *Logic in Computer Science, Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004.

[20] D. Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.

[21] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[22] A. L. Juarez Dominguez, N. A. Day, and R. T. Fanson. A Preliminary Report on Tool Support and Methodology for Feature Interaction Detection. Technical Report CS-2007-44, Cheriton School of Computer Science, University of Waterloo, 2007.

[23] Y. Kesten and A. Pnueli. A compositional approach to CTL^* verification. *Theoretical Computer Science*, pages 397 – 428, 2005.

[24] J. Kramer. Is Abstraction the Key to Computing? *CACM*, 50(4):36–42, Apr. 2007.

[25] K. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods*, LNCS, pages 219–237. Springer Berlin Heidelberg, 1999.

[26] K. L. McMillan. The SMV system, Nov. 06 1992.

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

[28] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD*, volume 1954 of *LNCS*, pages 127–144. Springer, 2000.

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

[30] A. Vakili and N. A. Day. Reducing CTL-live Model Checking to First-Order Logic Validity Checking. In *FMCAD'14*, Oct 2014. To appear as a short paper.

[31] A. Vakili and N. A. Day. Reducing CTL-live Model Checking to Semantic Entailment in First-Order Logic (Version 1). Technical Report CS-2014-05, Cheriton School of Computer Science, University of Waterloo, 2014.

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