CTL+FO Verification as Constraint Solving

Tewodros A. Beyene
Technische Universität
München, Germany

Marc Brockschmidt
Microsoft Research
Cambridge, UK

Andrey Rybalchenko
Microsoft Research
Cambridge, UK

ABSTRACT

Expressing program correctness often requires relating program data throughout (different branches of) an execution. Such properties can be represented using CTL+FO, a logic that allows mixing temporal and first-order quantification. Verifying that a program satisfies a CTL+FO property is a challenging problem that requires both temporal and data reasoning. Temporal quantifiers require discovery of invariants and ranking functions, while first-order quantifiers demand instantiation techniques. In this paper, we present a constraint-based method for proving CTL+FO properties automatically. Our method makes the interplay between the temporal and first-order quantification explicit in a constraint encoding that combines recursion and existential quantification. By integrating this constraint encoding with an off-the-shelf solver we obtain an automatic verifier for CTL+FO.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Theory

Keywords

Model Checking, Software Verification, Temporal Logic

1. Introduction

In specifying the correct behaviour of systems, relating data at various stages of a computation is often crucial. Examples include program termination [7] (where the value of a rank function should be decreasing over time), correctness of reactive systems [13] (where each incoming request should be handled in a certain timeframe), and information flow [11] (where for all possible secret input values, the output should be the same). The logic CTL+FO offers a natural specification mechanism for such properties, allowing to freely mix temporal and first-order quantification. First-order quantification makes it possible to specify variables dependent on the current system state, and temporal quantifiers allow to relate this data to system states reached at a later point.

While CTL+FO and similar logics have been identified as a specification language before, no fully automatic method to check CTL+FO properties on infinite-state systems was developed. Hence, the current state of the art is to either produce verification tools specific to small subclasses of properties, or using error-prone program modifications that explicitly introduce and initialize ghost variables, which are then used in (standard) CTL specifications.

In this paper, we present a fully automatic procedure to transform a CTL+FO verification problem into a system of existentially quantified recursive Horn clauses. Such systems can be solved by leveraging recent advances in constraint solving [3], allowing to blend first-order and temporal reasoning. Our method benefits from the simplicity of the proposed proof rule and the ability to leverage on-going advances in Horn constraint solving.

Related Work.

Verification of CTL+FO and its decidability and complexity have been studied (under various names) in the past. Bohn et al. [5] presented the first model-checking algorithm. Predicates partitioning a possibly infinite state space are deduced syntactically from the checked property, and represented symbolically by propositional variables. This allows to leverage the efficiency of standard BDD-based model checking techniques, but the algorithm fails when the needed partition of the state space is not syntactically derivable from the property.

Working on finite-state systems, Hallé et al. [10], Patthak et al. [15] and Rensink [16] discuss a number of different techniques for quantified CTL formulas. In these works, the finiteness of the data domain is exploited to instantiate quantified variables, thus reducing the model checking problem for quantified CTL to standard CTL model checking.

Hodkinson et al. [13] study the decidability of CTL+FO and some fragments on infinite state systems. They show the general undecidability of the problem, but also identify certain decidable fragments. Most notably, they show that by restricting first order quantifiers to state formulas and only applying temporal quantifiers to formulas with at most one free variable, a decidable fragment can be obtained. Finally, Da Costa et al. [8] study the complexity of checking properties over propositional Kripke structures, also providing

SPIN '14, July 21Ű23, 2014, San Jose, CA, USA

an overview of related decidability and complexity results. In temporal epistemic logic, Belardinelli et al. [1] show that checking FO-CTLK on a certain subclass of infinite systems can be reduced to finite systems. In contrast, our method directly deals with quantification over infinite domains.

2. Preliminaries

Programs.

We model programs as transition systems. A program P consists of a tuple of program variables v, an initial condition $init(v)$, and a transition relation $next(v, v')$. A state is a valuation of v. A computation π is a maximal sequence of states $s_1, s_2, \ldots$ such that $init(s_1)$ and for each pair of consecutive states (s, s') we have $next(s, s')$. The set of computations of P starting in s is denoted by $\Pi_P(s)$.

CTL+FO *Syntax and Semantics.*

The following definitions are standard, see e.g. [5, 14].

Let $\mathcal{T}$ be a first order theory and $\models_{\mathcal{T}}$ denote its satisfaction relation that we use to describe sets and relations over program states. Let c range over assertions in $\mathcal{T}$ and x range over variables. A CTL+FO formula φ is defined by the following grammar using the notion of a path formula ϕ.

$$\varphi ::= \forall x : \varphi \mid \exists x : \varphi \mid c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\,\phi \mid E\,\phi$$
$$\phi ::= X\varphi \mid G\varphi \mid \varphi U \varphi$$

As usual, we define $F\varphi = (true U \varphi)$. The satisfaction relation $P \models \varphi$ holds if and only if for each s such that $init(s)$ we have $P, s \models \varphi$. We define $P, s \models \varphi$ as follows using an auxiliary satisfaction relation $P, \pi \models \phi$. Note that d ranges over values from the corresponding domain.

$P, s \models \forall x : \varphi$	iff for all d holds $P, s \models \varphi[d/x]$
$P, s \models \exists x : \varphi$	iff exists d such that $P, s \models \varphi[d/x]$
$P, s \models c$	iff $s \models_{\mathcal{T}} c$
$P, s \models \varphi_1 \wedge \varphi_2$	iff $P, s \models \varphi_1$ and $P, s \models \varphi_2$
$P, s \models \varphi_1 \vee \varphi_2$	iff $P, s \models \varphi_1$ or $P, s \models \varphi_2$
$P, s \models A\,\phi$	iff for all $\pi \in \Pi_P(s)$ holds $P, \pi \models \phi$
$P, s \models E\,\phi$	iff exists $\pi \in \Pi_P(s)$ such that $P, \pi \models \phi$
$P, \pi \models X\varphi$	iff $\pi = s_1, s_2, \ldots$ and $P, s_2 \models \varphi$
$P, \pi \models G\varphi$	iff $\pi = s_1, s_2, \ldots$ for all $i \geq 1$ holds $P, s_i \models \varphi$
$P, \pi \models \varphi_1 U \varphi_2$	iff $\pi = s_1, s_2, \ldots$ and exists $j \geq 1$ such that $P, s_j \models \varphi_2$ and $P, s_i \models \varphi_1$ for $1 \leq i \leq j$

Quantified Horn Constraints.

Our method uses the EHSF [3] solver for forall-exists Horn constraints and well-foundedness. We omit the syntax and semantics of constraints solved by EHSF, see [3] for details. Instead, we consider an example:

$$x \geq 0 \rightarrow \exists y : x \geq y \wedge rank(x, y), \qquad wf(rank).$$

These constraints are an assertion over the interpretation of the "query symbol" *rank* (the predicate *wf* is not a query symbol, but requires well-foundedness). A solution maps query symbols into constraints. The example has a solution that maps $rank(x, y)$ to the constraint $(x \geq 0 \wedge y \leq x - 1)$.

EHSF resolves clauses like the above using a CEGAR scheme to discover witnesses for existentially quantified variables. The refinement loop collects a global constraint that declaratively determines which witnesses can be chosen. The chosen witnesses are used to replace existential quantification, and then the resulting universally quantified clauses are passed to a solver over decidable theories, e.g., HSF [9] or μZ [12]. Such a solver either finds a solution, i.e., a model for uninterpreted relations constrained by the clauses, or returns a counterexample, which is a resolution tree (or DAG) representing a contradiction. EHSF turns the counterexample into an additional constraint on the set of witness candidates, and continues with the next iteration of the refinement loop.

For the existential clause above, EHSF introduces a witness/Skolem relation *sk* over variables x and y, i.e., $x \geq 0 \wedge sk(x, y) \rightarrow x \geq y \wedge rank(x, y)$. In addition, since for each x such that $x \geq 0$ holds we need a value y, we require that such x is in the domain of the Skolem relation using an additional clause $x \geq 0 \rightarrow \exists y : sk(x, y)$. In the EHSF approach, the search space of a Skolem relation $sk(x, y)$ is restricted by a template function $\text{TEMPL}(sk)(x, y)$. To conclude this example, we note that one possible solution returned by EHSF is the Skolem relation $sk(x, y) = (y \leq x - 1)$.

3. Constraint Generation

In this section we present our algorithm GEN for generating constraints that characterize the satisfaction of a CTL+FO formula. We also consider its complexity and correctness and present an example.

See Figure 1. GEN performs a top-down, recursive descent through the syntax tree of the given CTL+FO formula. It introduces auxiliary predicates and generates a sequence of implication and well-foundedness constraints over these predicates. We use "," to represent the concatenation operator on sequences of constraints. At each level of recursion, GEN takes as input a CTL+FO formula φ_0, a tuple of variables v_0 that are considered to be in scope and define a state, assertions $init(v_0)$ and $next(v_0, v_0')$ that describe a set of states and a transition relation, respectively. We assume that variables bound by first-order quantifiers in φ_0 do not shadow other variables. To generate constraints for checking if $P = (v, init(v), next(v, v'))$ satisfies φ we execute $\text{GEN}(\varphi, v, init(v), next(v, v'))$.

Handling First-Order Quantification.

When φ_0 is obtained from some φ_1 by universally quantifying over x, we directly descend into φ_1 after adding x to the scope. Hence, the recursive call to GEN uses $v_1 = (v_0, x)$. Since $init(v_0)$ defines a set of states over v_1 in which x ranges over arbitrary values, the application $\text{GEN}(\varphi_1, v_1, init(v_0), \ldots)$ implicitly requires that φ_1 holds for arbitrary x. Since the value of x is arbitrary but fixed within φ_1, we require that the transition relation considered by the recursive calls does not modify x and thus extend *next* to $next(v_0, v_0') \wedge x' = x$ in the last argument.

When φ_0 is obtained from some φ_1 by existentially quantifying over x, we use an auxiliary predicate *aux* that implicitly serves as witness for x. A first constraint connects the set of states $init(v_0)$ on which φ_0 needs to hold with $aux(v_1)$, which describes the states on which φ_1 needs to hold. We require that for every state s allowed by $init(v_0)$, a choice of x exists such that the extension of s with x is allowed by $aux(v_1)$. Then, the recursive call $\text{GEN}(\varphi_1, v_1, aux(v_1), \ldots)$ generates constraints that keep track of satisfaction of φ_1 on arbitrary x allowed by $aux(v_1)$. Thus, $aux(v_1)$ serves as a restriction of the choices allowed for x. Again, we enforce rigidness of x by adding $x' = x$ to the *next* relation.

$\textsc{Gen}(\varphi_0, v_0, \mathit{init}(v_0), \mathit{next}(v_0, v_0')) =$
match φ_0 with
$\mid \forall x : \varphi_1 \Rightarrow$
let $v_1 = (v_0, x)$ in
$\textsc{Gen}(\varphi_1, v_1, \mathit{init}(v_0), \mathit{next}(v_0, v_0') \land x' = x)$
$\mid \exists x : \varphi_1 \Rightarrow$
let $v_1 = (v_0, x)$ in
let aux = fresh symbol of arity $|v_1|$ in
$\mathit{init}(v_0) \to \exists x : \mathit{aux}(v_1)$,
$\textsc{Gen}(\varphi_1, v_1, \mathit{aux}(v_1), \mathit{next}(v_0, v_0') \land x' = x)$
$\mid c \Rightarrow$
$\mathit{init}(v_0) \to c$
$\mid EF\varphi_1 \Rightarrow$
let $\mathit{inv}, \mathit{aux}$ = fresh symbols of arity $|v_0|$ in
let rank = fresh symbol of arity $|v_0| + |v_0|$ in
$\mathit{init}(v_0) \to \mathit{inv}(v_0)$,
$\mathit{inv}(v_0) \land \neg \mathit{aux}(v_0) \to \exists v_0' : \mathit{next}(v_0, v_0') \land \mathit{inv}(v_0') \land \mathit{rank}(v_0, v_0')$,
$\mathit{wf}(\mathit{rank})$,
$\textsc{Gen}(\varphi_1, v_0, \mathit{aux}(v_0), \mathit{next}(v_0, v_0'))$

Figure 1: Constraint generation rules for FO quantification, assertions, and existential/eventually temporal quantification.

Handling Temporal Quantification.

We use a deductive proof system for CTL [14] and consider its proof rules from the perspective of constraint generation.

When φ_0 is a background theory assertion, i.e., does not use path quantification, Gen produces a constraint that requires φ_0 to hold on every initial state.

When φ_0 requires that there is a path on which φ_1 eventually holds, then Gen uses an auxiliary predicate $\mathit{aux}(v_0)$ to describe those states in which φ_1 holds. Gen applies a combination of inductive reasoning together with well-foundedness to show that $\mathit{aux}(v_0)$ is eventually reached from the initial states. The induction hypothesis is represented as $\mathit{inv}(v_0)$ and is required to hold for every initial state and whenever $\mathit{aux}(v_0)$ is not reached yet. Then, the well-foundedness condition wf, which requires that it is not possible to come back into the induction hypothesis forever, ensures that eventually we reach a "base case" in which $\mathit{aux}(v_0)$ holds. Hence, eventually φ_1 holds on some computations.

Note that the induction hypothesis $\mathit{inv}(v_0)$, the well-founded relation $\mathit{rank}(v_0, v_0')$, and the predicate $\mathit{aux}(v_0)$ are left for the solver to be discovered.

See [2] for the remaining rules that describe the full set of CTL temporal quantifiers.

Complexity and Correctness.

Gen performs a single top-down descent through the syntax tree of the given CTL+FO formula φ. The running time and the size of the generated sequence of constraints is linear in the size of φ. Finding a solution for the generated constraints is undecidable in general. In practice however, the used solver often succeeds in finding a solution (cf. Sect. 4). We formalize the correctness of Gen in the following theorem.

Theorem 1. *For a given program P with $\mathit{init}(v)$ and $\mathit{next}(v, v')$ over v and a* CTL+FO *formula φ the application $\textsc{Gen}(\varphi, v, \mathit{init}(v), \mathit{next}(v, v'))$ computes a constraint that is satisfiable if and only if $P \models \varphi$.*

Proof. (sketch) We omit the full proof here for space reasons. We proceed by structural induction, as the constraint generation of the algorithm Gen. Formally, we prove that the constraints generated by $\textsc{Gen}(\varphi_0, v_0, \mathit{init}(v_0), \mathit{next}(v_0, v_0'))$ have a solution if and only if the program $P = (v_0, \mathit{init}(v_0), \mathit{next}(v_0, v_0'))$ satisfies φ_0. The base case, i.e., φ_0 is an assertion c from our background theory $\mathcal{T}$, is trivial.

As example for an induction step, we consider $\varphi_0 = \exists x : \varphi_1$. To prove soundness, we assume that the generated constraints have a solution. For the predicate aux, this solution is a relation S_{aux} that satisfies all constraints generated for aux. For each s with $\mathit{init}(s)$, we choose $\overline{x}_s$ such that $(s, \overline{x}_s) \in S_{aux}$. As we require $\mathit{init}(v_0) \to \exists x : \mathit{aux}(v_0, x)$, this element is well-defined. We now apply the induction hypothesis for $P' = ((v_0, x), \mathit{aux}(v_0, x), \mathit{next}(v_0, v_0') \land x' = x)$ and φ_1. Then for all s with $\mathit{init}(s)$, we have $P', (s, \overline{x}_s) \models \varphi_1$, and as P' is not changing x by construction, also $P', (s, \overline{x}_s) \models \varphi_1[\overline{x}_s/x]$. From this, $P, s \models \varphi_0$ directly follows.

For completeness, we proceed analogously. If $P, \varphi_0 \models$ holds, then a suitable instantiation $\overline{x}_s$ of x can be chosen for each s with $\mathit{init}(s)$, and thus we can construct a solution for $\mathit{aux}(v_0, x)$ from $\mathit{init}(v_0)$. $\square$

Example.

We illustrate Gen (see Figure 1) on a simple example. We consider a property that the value stored in a register v can grow without bound on some computation.

$$\forall x : v = x \to EF(v > x)$$

This property can be useful for providing evidence that a program is actually vulnerable to a denial of service attack. Let $\mathit{init}(v)$ and $\mathit{next}(v, v')$ describe a program over a single variable v.

We apply Gen on the property and the program and obtain the following application trace (here, we treat $\to$ as expected, as its left-hand side is a background theory atom).

$\textsc{Gen}(\forall x{:}v{=}x \to EF(v{>}x), v, \quad \mathit{init}(v), \quad \mathit{next}(v, v'))$
$\textsc{Gen}(v{=}x \to EF(v{>}x), \quad (v,x), \mathit{init}(v), \quad \mathit{next}(v, v') \land x'{=}x)$
$\textsc{Gen}(v{=}x \to \mathit{aux}(v,x), \quad (v,x), \mathit{init}(v), \quad \mathit{next}(v, v') \land x'{=}x)$
$\textsc{Gen}(EF(v{>}x), \quad (v,x), \mathit{aux}(v,x), \mathit{next}(v, v') \land x'{=}x)$

This trace yields the following constraints.

$\mathit{init}(v) \to (v = x \to \mathit{aux}(v))$
$\mathit{aux}(v) \to \mathit{inv}(v, x)$
$\mathit{inv}(v, x) \land \neg(v > x) \to \exists v', x' : \mathit{next}(v, x, v', x') \land x' = x \land \mathit{inv}(v', x') \land \mathit{rank}(v, x, v', x')$
$\mathit{wf}(\mathit{rank})$

Note that there exists an interpretation of aux, inv, and rank that satisfies these constraints if and only if the program satisfies the property.

4. Evaluation

In this section we present CTLFO, a CTL+FO verification engine. CTLFO implements the procedure Gen and applies Ehsf [3] to solve resulting clauses.

We run CTLFO on the examples `OS frag.1`, ..., `OS frag.4` from industrial code from [6, Figure 7]. Each example consists of a program and a CTL property to be

	Property ϕ	$\models_{CTL+FO}\ \phi$		$\models_{CTL+FO}\ \neg\phi$	
		Res.	Time	Res.	Time
P1	$\exists x : AG(a = x \rightarrow AF(r = 1))$	✓	1.0	×	0.1
	$AG(\exists x : a = x \rightarrow AF(r = 1))$	✓	0.9	×	0.1
P2	$\exists x : EF(a = x \wedge EG(r \neq 5))$	✓	0.9	×	0.2
	$EF(\exists x : a = x \wedge EG(r \neq 5))$	✓	0.6	×	0.2
P3	$\exists x : AG(a = x \rightarrow EF(r = 1))$	✓	1.1	×	0.1
	$AG(\exists x : a = x \rightarrow EF(r = 1))$	✓	1.0	×	0.1
P4	$\exists x : EF(a = x \wedge AG(r \neq 1))$	✓	1.8	×	0.4
	$EF(\exists x : a = x \wedge AG(r \neq 1))$	✓	0.9	×	0.4
P5	$\exists x : AG(s = x \rightarrow AF(u = x))$	✓	7.0	×	0.1
	$AG(\exists x : s = x \rightarrow AF(u = x))$	✓	7.2	×	0.1
P6	$\exists x : EF(s = x \wedge EG(u \neq x))$	✓	1.8	×	2.2
	$EF(\exists x : s = x \wedge EG(u \neq x))$	✓	1.1	×	2.1
P7	$\exists x : AG(s = x \rightarrow EF(u = x))$	✓	3.1	×	0.2
	$AG(\exists x : s = x \rightarrow EF(u = x))$	✓	6.5	×	0.1
P8	$\exists x : EF(s = x \wedge AG(u \neq x))$	✓	14.3	×	1.8
	$EF(\exists x : s = x \wedge AG(u \neq x))$	✓	13.9	×	1.8
P9	$\exists x : AG(a = x \rightarrow AF(r = 1))$	✓	118.7	×	17.3
	$AG(\exists x : a = x \rightarrow AF(r = 1))$	✓	82.3	×	1.4
P10	$\forall x : EF(a = x \wedge EG(r \neq 1))$	T/O	-	×	3.5
	$EF(\forall x : a = x \wedge EG(r \neq 1))$	T/O	-	×	3.5
P11	$\exists x : AG(a = x \rightarrow EF(r = 1))$	✓	126.8	×	3.6
	$AG(\exists x : a = x \rightarrow EF(r = 1))$	✓	140.3	×	0.2
P12	$\forall x : EF(a = x \wedge AG(r \neq 1))$	✓	146.7	×	3.2
	$EF(\forall x : a = x \wedge AG(r \neq 1))$	✓	161.7	×	0.2
P13	$\exists x : AF(io = x) \vee AF(ret = x)$	✓	576.8	×	0.3
P14	$\exists x : EG(io \neq x) \wedge EG(ret \neq x)$	✓	15.1	×	48.1
P15	$\exists x : EF(io = x) \wedge EF(ret = x)$	✓	166.4	×	1.9
P16	$\exists x : AG(io \neq x) \vee AG(ret \neq x)$	✓	3.4	T/O	-

Table 1: Evaluation of CTLFO on benchmarks from [6].

proven. We have modified the given properties to lift the CTL formula to CTL+FO. As example, consider the property $AG(a = 1 \rightarrow AF(r = 1))$. One modified property to check could be $\exists x : AG(a = x \rightarrow AF(r = 1))$, and another one is $AG(\exists x : (a = x \rightarrow AF(r = 1)))$. By doing similar satisfiability-preserving transformations of the properties for all the example programs, we get a set programs whose properties are specified in CTL+FO as shown in Table 1. For each pair of a program and CTL+FO property ϕ, we generated two verification tasks: proving ϕ and proving $\neg\phi$. While the existence of a proof for a property ϕ implies that $\neg\phi$ is violated by the same program, we consider both properties to show the correctness of our tool.

We report the results in Table 1. ✓ (resp. ×) marks the cases where CTLFO was able to prove (resp. disprove) a CTL+FO property. T/O marks the cases where CTLFO was not able to find either a solution or a counter-example in 600 seconds.

CTLFO is able to find proofs for all the correct programs except for P10 and counter-examples for all incorrect programs except for P16. Currently, CTLFO models the control flow symbolically using a program counter variable, which we believe is the most likely reason for the solving procedure to time out. Efficient treatment of control flow along the lines of explicit analysis as performed in the CPAchecker framework could lead to significant improvements for dealing with programs with large control-flow graphs [4]. An executable of CTLFO, together with a more verbose evaluation, can be found at `https://www7.in.tum.de/~beyene/ctlfo/`.

For cases where the property contains nested path quantifiers and the outer temporal quantifier is F or U, our implementation may generate non-Horn clauses following the proof system from [14]. While a general algorithm for solving non-Horn clauses is beyond the scope of this paper, we used a simple heuristic to seed solutions for queries appearing under the negation operator.

5. Conclusion

This paper presented an automated method for proving program properties written in the temporal logic CTL+FO, which combines universal and existential quantification over time and data. Our approach relies on a constraint generation algorithm that follows the formula structure to produce constraints in the form of Horn constraints with forall/exists quantifier alternation. The obtained constraints can be solved using an off-the-shelf constraint solver, thus resulting in an automatic verifier.

6. References

[1] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *KR*, 2012.

[2] T. Beyene, M. Brockschmidt, and A. Rybalchenko. CTL+FO verification as constraint solving. *CoRR*, abs/1406.3988, 2014.

[3] T. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, 2013.

[4] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *FASE*, 2013.

[5] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order-CTL model checking. In *FSTTCS*, 1998.

[6] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *PLDI*, 2013.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[8] A. Da Costa, F. Laroussinie, and N. Markey. Quantified CTL: expressiveness and model checking. In *CONCUR*, 2012.

[9] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[10] S. Hallé, R. Villemaire, O. Cherkaoui, and B. Ghandour. Model checking data-aware workflow properties with CTL-FO$^+$. In *EDOC*, 2007.

[11] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ASAC*, 2010.

[12] K. Hoder, N. Bjørner, and L. M. de Moura. μZ - an efficient engine for fixed points with constraints. In *CAV*, 2011.

[13] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *LICS*, 2002.

[14] Y. Kesten and A. Pnueli. A compositional approach to CTL* verification. *Theor. Comput. Sci.*, 331(2-3):397–428, 2005.

[15] A. C. Patthak, I. Bhattacharya, A. Dasgupta, P. Dasgupta, and P. Chakrabarti. Quantified computation tree logic. *Information processing letters*, 82(3):123–129, 2002.

[16] A. Rensink. Model checking quantified computation tree logic. In *CONCUR*, 2006.