LTL Model Checking using Coinductive Answer Set programming

Sarat Chandra Varanasi[1], Neda Saeedloei[2], Elmer Salazar[1], Joaquín Arias[3] and Gopal Gupta[1]

[1]The University of Texas at Dallas, Richardson, TX, USA
[2]Towson University, Towson, MD, USA
[3]CETINIA, Universidad Rey Juan Carlos, Madrid, Spain

Abstract

We present a model checker for Linear Temporal Logic using Goal-Directed Answer Set Programming under Costable model semantics (CoASP). Costable model semantics allows for positive loops to succeed unlike Stable model semantics where positive loops fail. Therefore, by using the Costable model semantics, LTL formulas involving the $\mathcal{G}$ and $\mathcal{R}$ operator can be proved coinductively.

Keywords

Goal-Directed Answer Set Programming, LTL Model Checking, Coinductive ASP

1. Introduction

We present an LTL model checker expressed in Coinductive Answer Set Programming. Coinductive Answer Set Programming is to be distinguished from Answer Set Programming by its underlying Costable model semantics where positive loops are allowed to succeed coinductively [1]. Allowing positive loops to succeed allows one to elegantly perform LTL model checking. Verifying LTL formulas involving the $\mathcal{G}$ operator can be mapped to looking for coinductive success in Coinductive Logic programming. Prior works exist in literature that provide an LTL interpreter using Coinductive Logic Programming [2]. In this work, we show how Coinductive ASP (CoASP) can be used to evaluate LTL formulas. Our work shows an important application of CoASP. Further, we use the implementation of CoASP incorporated in s(CASP) [3], a goal-directed Constraint ASP system, to evaluate LTL formulas. We assume familiarity with LTL syntax and semantics.

2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22), August 1, 2022
sxv153030@utdallas.edu (S. C. Varanasi); nsaeedloei@towson.edu (N. Saeedloei); elmer.salazar@utdallas.edu (E. Salazar); joaquin.arias@urjc.es (J. Arias); gupta@utdallas.edu (G. Gupta)
0000-0002-4620-4266 (S. C. Varanasi); 0000-0003-4148-311X (J. Arias); 0000-0001-9727-0362 (G. Gupta)
© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

2. Background

2.1. Coinductive Logic Programming and Goal-Directed ASP

Coinductive Logic Programming allows for circular terms to be constructed when performing unification [4]. This deviates from traditional least-fix point semantics-based logic programming where the *occurs check* prevents the construction of circular terms (which are equivalent to an infinite term). For performance reasons, Logic Programming systems do not perform an *occurs check* but fail to terminate when a circular term is to be printed. Making programs terminate when a circular term is constructed marked the birth of Coinductive Logic Programming (CoLP) [4]. A lot of work has been done in (constraint) CoLP towards verification of safety and liveness of cyber physical systems [5, 6, 2, 7, 8]. The development of Goal-Directed ASP was inspired from CoLP [6, 9]. The goal-directed ASP search succeeds coinductively when a goal unifies with itself in the call-stack through an even number of negations. This is similar to coinductive success in CoLP except that the negations are not counted. Goal-directed ASP was first developed for propositional answer set programs[10] and has been extended to predicate ASP programs [9, 11] and more recently to predicate ASP programs with constraints over reals [3]. The s(CASP) system implements goal-directed solving of predicate ASP programs with support for CLP(Q), with many applications in Knowledge Representation and Reasoning [12, 13, 14] to the modeling and verification of timed systems [15].

2.2. Costable Model semantics

The Costable model semantics was introduced in 2012 [1] and formalized by Elmer Salazar in the context of providing proof-theoretic foundations to *negation-as-failure* (NAF) in logic programming [16]. Costable model semantics allows positive loops to succeed. For example, a loop such as

```
p :- p.
```

is a positive loop. This loop does not succeed under the stable model semantics, because the last step of the Gelfond-Lifschitz transform [17] computes the least fixpoint (LFP) semantics of the residual program. Alternatively, we could compute the greatest fixpoint (GFP) semantics of the residual program in the last step of the transform. With the GFP semantics, the program

```
p :- p.
```

will produce an additional stable model, namely, {p}, in addition to the one corresponding to the LFP, namely, {}. The GFP-based stable model assumes that there is in infinite (coinductive) proof for p. The Costable model semantics precisely formulates this notion and finds GFP-based stable models for such positive loops. This property of the Costable model semantics lends itself well to LTL model checking. LTL formulas are defined over infinite sequences of states in a state transition system and verifying properties that are *globally true* can naturally be verified via a coinductive proof. More details about the costable model semantics can be found elsewhere [1, 16].

2.3. CoASP in s(CASP)

CoLP enables assumption-based reasoning. Given the rule

```
p :- p.
```

if we accept that p is entailed by this program, then essentially we are stating that p holds if we assume p to be true. There are instances where this may appear counter-intuitive as the following example shows. Consider the program for reachability of nodes in a graph:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- edge(X,Z), reach(Z,Y).
edge(1,2). edge(2,1). edge(3,3).
```

If we ask the query `?- reach(1,3).` then it will succeed under coLP and coASP, even though it is clear that node `3` cannot be reached from node `1`. If we look at the coinductive proof trace we see the following:

```
reach(1,3) → edge(1,Z),reach(Z,3) → reach(2,3) →
     edge(2,Z'),reach(Z',3) → reach(1,3) [coinductive success]
```

Not only `reach(1,3)` holds because we assume `reach(1,3)`, the subgoal `reach(2,3)` also holds because of this assumption. To make this assumption explicit, s(CASP) always prints all such assumptions upon termination of the query when it is called with coinductive execution option (with the `–co flag`).

3. LTL Model Checker in CoASP

The LTL model checker is succinctly described in CoASP as well as CoLP [5]. The transition system over which LTL propositions are checked is captured as a set of `trans/2` facts. For example, `trans(s1, s2)` means that state `s2` is the next state from state `s1`. For atomic propositions, `holds(s, p)` is used to denote that the atomic proposition `p` is true in state `s`. Compound propositions involve the standard LTL operators over atomic propositions. An interesting way to describe atomic propositions is by using NAF over non-atomic propositions as shown in Code 1.

```
not_atomic(and(_, _)). % and operator
not_atomic(or(_,_)).   % or operator
not_atomic(f(_)).      % future operator
not_atomic(g(_)).      % always operator
not_atomic(x(_)).      % next operator
atomic(X) :- not not_atomic(X).
```

Code 1: Atomic propositions as NAF over compound LTL formulas

The model checker itself is found in Code 2. The verification is achieved by `verify/3` predicate, where `verify(s, f, path)` means that the formula `f` is true on the (infinite) `path` starting

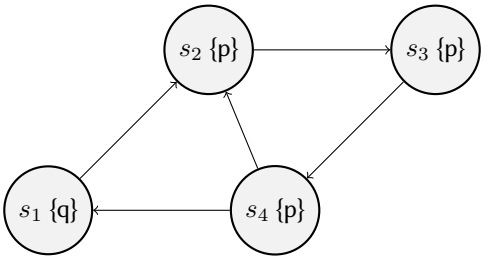

Figure 1: State transition system, q holds in s1, p holds in s2, s3, s4

at state `s`. The proposition `true` is unconditionally true in every state. An atomic proposition `p` is true at state `s`, if `holds(s,p)` is a fact. Negation of a formula is proved by using NAF. Other rules follow from the semantics of LTL formulas. When considering the `g(p)` for some formula `p`, an infinite path is first chosen using the `co_path` predicate. The predicate `co_path` non-deterministically returns paths that contain a cycle in the transition graph. The definitions of `co_path` are typical list processing rules over the `trans/2` terms and are not shown for brevity. Once the infinite path is selected, the proof of `g(p)` succeeds if and only if there is a coinductive success under the costable model semantics.
The interpreter can be similarly described in CoLP. In CoLP, the `co_path` can be removed as an infinite path is found automatically by finding a circular term involving states. Because, s(CASP) does not support circular terms, the `co_path` predicate is used.

4. An example

Consider the following transition system as shown in Figure 1. The atomic proposition `q` is true in state `s1`, and the proposition `p` holds in states `s2`, `s3` and `s4`. By observing the transition system it is easy to follow that `x(p)` holds in state `s1`, similarly `f(g(p))` holds in `s1`. These formulas can be checked by asking s(CASP) the query `?- verify(s1, x(p))`. Before querying s(CASP), it should be launched using the `-co` option to execute the queries in CoASP mode. If the formula `x(p)` is true in `s1`, the query would succeed and the path along which the formula holds would be returned in the s(CASP) model. Alternatively, an infinite path can be chosen and then a formula can be checked as follows: `?- co_path(s1, Path), verify(s1, x(p), Path)`. Due to s(CASP)'s support to print the justification tree for a query, one can alternatively explore why a certain formula holds at a certain state. A fragment of the justification tree of the previous query is shown in Code 3. Similarly to `f(g(p))` being true at `s1`, the formula `f(g(neg(q)))` succeeds at `s1`. Note that we use NAF to indicate that a certain atomic proposition does not hold in a state. Since, `q` is not mentioned in states `s2`, `s3` and `s4`, it is always true in the infinite sequence `s2`, `s3`, `s4`, `s2`, `s3`, `s4`.... The source files are available in this github repo [18].
Consider another query that uses the `G` operator. It is clear that `g(f(p))` is true in state `s1`. This is because `f(p)` is true in the infinite path `s1`, `s2`, `s3`, `s4`, `s2`, `s3`, `s4`.... In this case, the proof has to succeed by coinduction and s(CASP) prints a warning to the user that a proof by coinduction using a positive loop was performed. s(CASP) flags this as a warning as

```
verify(S, P) :- co_path(S, Path), verify(S, P, Path).

verify(_, true, _). % true holds unconditionally

verify(S, P, _) :- atomic(P), holds(S, P). % atomic propositions

verify(S, neg(P), Path) :- not verify(S, P, Path).% negation uses NAF
verify(S, or(P, Q), Path) :- verify(S, P, Path). % disjunction
verify(S, or(P, Q), Path) :- verify(S, Q, Path). % disjunction

verify(S, x(P), Path) :- % next operator
  member(trans(S, S1), Path), verify(S1, P, Path).

verify(S, f(P), Path) :- verify(S, P, Path). % future operator
verify(S, f(P), Path) :-
  member(trans(S, S1), Path), verify(S1, P, Path).

verify(S, g(P), Path) :- coverify(S, g(P), Path).
coverify(S, g(P), Path) :-  % always operator: relies on coinduction
  verify(S, and(P, x(g(P))), Path).

verify(S, and(P, Q), Path) :- % conjunction
verify(S, P, Path), verify(S, Q, Path).

% co_path and its auxiliary definitions
co_path(S, P) :-
    co_path(S, [], P).

co_path(S, V, [trans(S, S1)|V]) :- trans(S, S1), member_state(S1, V).

co_path(S, V, P) :-
      trans(S, S1),
      has_trans(S1),
      not member_state(S1, V),
      co_path(S1, [trans(S, S1)|V], P).

member_state(S,[trans(S, _)|_]).
member_state(S,[trans(_, S)|_]).
member_state(S,[trans(S1, S2)|T]) :-
     S \= S1, S \= S2,member_state(S,T).

has_trans(S) :- trans(S, _).
```

Code 2: LTL Model Checker in CoASP, rules directly capture LTL semantics

```
verify(s1,x(p),[trans(s1,s2),trans(s2,s3),trans(s3,s4),trans(s4,s2)]) :-
     member(trans(s1,s2),[trans(s1,s2),trans(s2,s3),trans(s3,s4),trans(s4,s2)]),
    verify(s2,p,[trans(s1,s2),trans(s2,s3),trans(s3,s4),trans(s4,s2)]) :-
        atomic(p) :-
            not not_atomic(p).
        holds(s2,p).
```

Code 3: Proof tree for why x(p) is true in state s1 as printed by s(CASP)

```
WARNING: Coinductive success due to a positive loop
verify(s2,g(f(p)),
[trans(s4,s2),trans(s3,s4),trans(s2,s3),trans(s1,s2)])
verify(s2,g(f(p)),
[trans(s4,s2),trans(s3,s4),trans(s2,s3),trans(s1,s2)])
```

Code 4: A message printed to user upon success due to costable models

the primary use of s(CASP) is geared towards Stable model semantics and the application of Co-stable model semantics is novel and recent. Because the infinite path is due to the repeating states: `s2, s3, s4...`, `verify/2` succeeds by Costable model semantics starting at state `s2`, as shown in 4.

5. Conclusion and Future Work

This work reports a novel application of CoASP in LTL model checking. The rules currently are quite inefficient. Formulas involving the and operator generate a lot of unnecessary paths before proving that both the sub-formulas contained within the and operator are satisfied on the same path. This can be improved by implementing the rules in a fail early manner, where the CoASP execution fails and backtracks upon finding the first state where both sub-formulas within a conjunction fail to hold. Due to s(CASP)'s support for CLP(R), the LTL model checker can be extended to support LTL formulas with real-time constraints (RT-LTL). Developing the RT-LTL checker in CoASP is part of future work.

References

[1] K. M. Gopal Gupta, et al., Coinductive answer set programming *or* consistency-based computing, in: Proc. ICLP'12 Workshop on Coinductive Logic Programming, 2012.

[2] N. Saeedloei, Verification of Complex Real-time Systems., Ph.D. thesis, Department of Computer Science, The University of Texas at Dallas, 2011.

[3] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, Theory and Practice of Logic Programming 18 (2018) 337–354.

[4] L. E. Simon, Extending logic programming with coinduction, Ph.D. thesis, Department of Computer Science, The University of Texas at Dallas, 2006.

[5] G. Gupta, N. Saeedloei, B. W. DeVries, R. Min, K. Marple, F. Kluzniak, Infinite computation, co-induction and computational logic, in: Proc. Algebra and Coalgebra in Computer Science - CALCO 2011, volume 6859 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 40–54.
[6] A. Bansal, Towards next Generation Logic Programming Systems, Ph.D. thesis, Department of Computer Science, The University of Texas at Dallas, 2007.
[7] N. Saeedloei, G. Gupta, Timed definite clause ω-grammars, in: M. V. Hermenegildo, T. Schaub (Eds.), Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK, volume 7 of *LIPIcs*, 2010, pp. 212–221.
[8] N. Saeedloei, G. Gupta, A logic-based modeling and verification of CPS, SIGBED Rev. 8 (2011) 31–34. doi:10.1145/2000367.2000374.
[9] R. K. Min, Predicate answer set programming with coinduction, The University of Texas at Dallas, 2009.
[10] K. Marple, G. Gupta, Galliwasp: A goal-directed answer set solver, in: International Symposium on Logic-Based Program Synthesis and Transformation, Springer, 2012, pp. 122–136.
[11] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, preprint arXiv:1709.00501 (2017).
[12] Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil, A physician advisory system for chronic heart failure management based on knowledge patterns, Theory and Practice of Logic Programming 16 (2016) 604–618.
[13] K. Basu, S. Varanasi, F. Shakerin, J. Arias, G. Gupta, Knowledge-driven natural language understanding of english text and its applications, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, 2021, pp. 12554–12563.
[14] K. Basu, S. C. Varanasi, F. Shakerin, G. Gupta, Square: Semantics-based question answering and reasoning engine, arXiv preprint arXiv:2009.10239 (2020).
[15] S. C. Varanasi, J. Arias, E. Salazar, F. Li, K. Basu, G. Gupta, Modeling and verification of real-time systems with the event calculus and s(CASP), in: International Symposium on Practical Aspects of Declarative Languages, Springer, 2022, pp. 181–190.
[16] E. E. Salazar, G. Gupta, Proof-theoretic foundations of normal logic programs, 2019. Technical Report, University of Texas at Dallas.
[17] M. Gelfond, Y. Kahl, Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach, Cambridge University Press, 2014.
[18] S. C. Varanasi, Ltl in coasp, https://github.com/sarat-chandra-varanasi/LTL-CoASP, 2022.