Abstraction-Based Model Checking of Linear Temporal Properties

Milán Mondok, András Vörös
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Email: mondokm@edu.bme.hu, vori@mit.bme.hu

***Abstract*—Even though the expressiveness of linear temporal logic (LTL) supports engineering application, model checking of such properties is a computationally complex task and state space explosion often hinders successful verification. LTL model checking consists of constructing automata from the property and the system, generating the synchronous product of the two automata and checking its language emptiness. We propose a novel LTL model checking algorithm that uses abstraction to tackle the challenge of state space explosion. This algorithm combines the advantages of two commonly used model checking approaches, counterexample-guided abstraction refinement and automata theoretic LTL model checking. The main challenge in combining these is the refinement of "lasso"-shaped counterexamples, for which task we propose a novel refinement strategy based on interpolation.**

I. Introduction

Linear temporal logic (LTL) specifications are particularly expressive and thus easy to use for engineers, but LTL model checking is a computationally expensive task. An efficient and commonly used linear temporal logic verification algorithm is based on automata theory. It consists of constructing automata from the property and the system, generating the synchronous product of the two automata and checking its language emptiness. This reduces the LTL model checking task to product calculation and language emptiness checking, which can be efficiently computed on Büchi automata, but the problem of state space explosion still hinders verification.

As the number of state variables in a system increases, the system's state space grows at least exponentially, which makes the exploration resource-intensive. Several approaches were developed to tackle the challenge of state space explosion. Counterexample-guided abstraction refinement checks a simplified model instead of the original problem, iteratively adding more detail until the verification task can be decided. Abstraction-based solutions proved efficient in reachability analysis, but have not been elaborated in the domain of LTL model checking yet.

We propose a novel LTL model checking algorithm that performs automata theoretic model checking on an iteratively refined abstract model. The abstraction is refined using a novel algorithm based on interpolation.

Our approach is similar to the one described by Zhao Duan et al. in [4]. As an optimization, they limit the scope of the verification to terminable programs and define an alternate version of LTL that is interpreted over finite paths. These alternate LTL formulas can be expressed using deterministic finite automata, which makes their verification computationally less demanding than regular LTL model checking.

II. Background

We use the following notation [6] from first-order logic (FOL) throughout our paper. Given a set of variables $V = \{v_1, v_2, ...\}$ let $V' = \{v'_1, v'_2, ...\}$ and $V^{\langle i \rangle} = \{v_1^{\langle i \rangle}, v_2^{\langle i \rangle}, ...\}$ represent the primed and indexed version of the variables. We use V' to refer to successor states and $V^{\langle i \rangle}$ for paths. Given an expression φ over $V \cup V'$, let $\varphi^{\langle i \rangle}$ denote the indexed expression obtained by replacing V and V' with $V^{\langle i \rangle}$ and $V^{\langle i+1 \rangle}$ respectively in φ.

A. Control flow automata

In our work we describe programs using *Control flow automata* (CFA) [6]. We define a *Control flow automaton* as a 4-tuple $\langle V, L, l_0, E \rangle$, where:

- $V = \{v_1, v_2, ..., v_k\}$ is the set of *variables*. Each variable v_i has an associated domain D_{v_i};
- L is the set of *control locations*, which model the program counter;
- $l_0 \in L$ is the initial location;
- $E \subseteq L \times Ops \times L$, where $op \in Ops$ are FOL formulas over V and V', is a set of directed edges representing the operations that are executed when control flows from the source location to the target.

A *concrete state* (l, c) is a pair of a location $l \in L$ and an interpretation $c \in D_{v_0} \times ... \times D_{v_n}$ that assigns a value $c(v) = d \in D_v$ to each variable $v \in V$ of its domain D_v. The set of initial states is $\{(l, c) | l = l_0\}$ and a transition exists between states (l, c) and (l', c') if an edge $(l, op, l') \in E$ exists with $(c, c') \models \mathrm{op}$. A *concrete path* is a finite, alternating sequence of concrete states and operations $\sigma = ((l_1, c_1), \mathrm{op}_1, ..., \mathrm{op}_{n-1}, (l_n, c_n))$ if $(l_i, \mathrm{op}_i, l_{i+1}) \in E$ for every $1 \leq i < n$ and $(c_1^{\langle 1 \rangle}, c_2^{\langle 2 \rangle}, ..., c_n^{\langle n \rangle}) \models \bigwedge_{1 \leq i < n} op_i^{\langle i \rangle}$, i.e., there is a sequence of edges starting from the initial location and the interpretations satisfy the semantics of the operations.

B. Counterexample-guided abstraction refinement

Counterexample-guided abstraction refinement (CEGAR) [2] [6] aims to tackle the problem of state space explosion by performing the verification task on a simpler, abstract model. The abstract model is an overapproximation of the concrete model: it contains all behaviours of the concrete model, but can contain additional behaviour as well. Such models are sufficient to prove the absence of counterexamples but can contain false positives, meaning that the counterexamples in the abstract models have to undergo further analysis.

The core of the CEGAR algorithm is the CEGAR-loop, which consists of two components, the *abstractor* and the *refiner*. The task of the abstractor is to calculate the abstract state space based on the current precision and to search for counterexamples, while the task of the refiner is to verify the concretizability of the abstract counterexample and refine the precision accordingly. The loop can only be left in two scenarios, either if the abstractor finds no counterexamples, or if the refiner finds that an abstract counterexample is feasible.

In *Boolean predicate abstraction* [6], an abstract state $s \in S$ in the set of abstract states is a Boolean combination of FOL predicates. A precision $\pi \in \Pi$ is a set of FOL predicates that are currently tracked by the algorithm. For example, if the current precision π contains two predicates, $(x < 0)$ and $(x < 1)$, then $true$, $x < 0$ or $!(x < 0) \land x < 1$ are examples of possible abstract states.

The result of the transfer function [6] $T(s, op, \pi)$ is the strongest Boolean combination of predicates in the precision that is entailed by the source state s and the operation op. This can be calculated by assigning a fresh propositional variable v_i to each predicate $p_i \in \pi$ and enumerating all satisfying assignments of the variables v_i in the formula $s \land op \land \bigwedge_{p_i \in \pi}(v_i \leftrightarrow p'_i)$. For each assignment, a conjunction of predicates is formed by taking predicates with positive variables and the negations of predicates with negative variables. The disjunction of all such conjunctions is the successor state s'.

Locations of the CFA are tracked explicitly. Abstract states $S_L = L \times S$ are pairs of a location $l \in L$ and a state $s \in S$. The transfer function extended with locations is $T_L((l, s), \pi) = \{(l', s') | (l, op, l') \in E, s' \in T(s, op, \pi)\}$, i.e., (l', s') is a successor of (l, s) if there is an edge between l and l' with op and s' is a successor of s with respect to the inner transfer function T.

An *abstract path* $\sigma = ((l_1, s_1), op_1, ..., op_{n-1}, (l_n, s_n))$ is an alternating sequence of abstract states and operations. An abstract path is *feasible* if a corresponding concrete path $((l_1, c_1), op_1, ..., op_{n-1}, (l_n, c_n))$ exists, where each c_i is mapped to s_i.

The abstractor explores the abstract state space using a search strategy (such as DFS of BFS) looking for counterexamples, i.e., abstract paths that start in the initial state and end in an error state. The exploration starts in the abstract state $(l_0, true)$. When visiting a state, all of its unvisited successors with respect to the transfer function T_L are visited by the search. The search can be optimized by not visiting *covered* successors, i.e. abstract states (l_c, s_c), for which an already visited (l_v, s_v) exists such that $l_c = l_v$ and $(s_c \Rightarrow s_v)$. If all reachable states were visited and no counterexample was found, then the model is *safe*, however, if a counterexample was found the refiner needs to check its validity.

The refinement [6] happens as follows. The input is a path $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, ..., op_{n-1}, (l_n, s_n))$ and the current precision π. First, the feasibility of the path is decided by querying an SMT solver with the formula $s_1^{\langle 1 \rangle} \land op_1^{\langle 1 \rangle} \land s_2^{\langle 2 \rangle} \land op_2^{\langle 2 \rangle} \land ... \land op_{n-1}^{\langle n-1 \rangle} \land s_n^{\langle n \rangle}$. If this formula is satisfiable, then the model is *unsafe* and a satisfying assignment to this formula is returned as the counterexample. Otherwise, an interpolant is calculated from the infeasible path σ that holds information for the further steps of refinement.

A Craig interpolant [7] for a mutually inconsistent pair of formulas (A,B) is a formula that is (1) implied by A, (2) inconsistent with B, and (3) expressed over the common variables of A and B.

A binary interpolant for an infeasible path σ can be calculated by defining $A \equiv s_1^{\langle 1 \rangle} \land op_1^{\langle 1 \rangle} \land ... \land op_{i-1}^{\langle i-1 \rangle} \land s_i^{\langle i \rangle}$ and $B \equiv op_i^{\langle i \rangle} \land s_{i+1}^{\langle i+1 \rangle}$, where i corresponds to the longest prefix of σ that is still feasible. The refined precision returned is the union of π and the new predicate that is obtained by replacing the variables $V^{\langle i \rangle}$ with V in this interpolant.

C. Automata theoretic LTL model checking

Kripke structures, LTL expressions and Büchi automata can all be used to characterize ω-regular languages [10]. As LTL expressions can only characterize a strict subset of ω-regular languages, while every ω-regular language can be recognized by a Büchi automaton, all LTL-expressions can be transformed to equivalent Büchi automata, for example using the algorithm of Gerth et al [5].

We regard the state space of the model as a Kripke structure M. Given an LTL-formula φ let $L(M)$ and $L(\varphi)$ denote the language that the Kripke structure can produce and the language that the LTL-formula specifies. The LTL model checking problem [3] can now be restated as follows: is the set of provided behaviours a subset of the valid behaviours, i.e., does $L(M) \subseteq L(\varphi)$ hold?

An equivalent formalization is $L(M) \cap \overline{L(\varphi)} \overset{?}{=} \emptyset$, where $\overline{L(\varphi)}$ is the complement of the language $L(\varphi)$. Complementation is computationally hard, but it can avoided in case of LTL model checking by utilizing that the complement of the language of an LTL-formula is the language of the negated formula: $\overline{L(\varphi)} \equiv L(\neg\varphi)$. This allows the model checking problem to be reduced to language intersection and language emptiness, both of which can be efficiently computed on Büchi automata.

A possible way of checking the language emptiness of a Büchi automaton is checking whether at least one strongly connected component (SCC) that contains an accepting state is reachable from the initial state. If such an SCC is reachable, then the Büchi automaton contains at least one run that contains an accepting state infinitely many times, fulfilling the acceptance condition of Büchi automata. Tarjan's algorithm [9] identifies SCCs using a single depth-first search (DFS) and clever indexing. Algorithms based on Nested DFS [8] offer a different approach. These algorithms usually conduct two depth-first searches, the former one to find and sort accepting states, and the latter one to find cycles that contain accepting states.

III. Overview of the Approach

LTL model checkers have always struggled with performance. We propose to use counterexample-guided abstraction refinement in LTL model checking. The key idea of our approach (Fig. 1) is that we conduct the automata theoretic LTL model checking on abstract models that we iteratively refine to the required precision using the the CEGAR algorithm.

The algorithm can work with various abstract domains, such as explicit value abstraction [1], predicate abstraction, or even a mix of the two. The appropriate abstraction method can only be selected based on the desired application domain. In this paper, we present the algorithm using predicate abstraction, a variant more suited for reactive systems as variables in such systems usually only get assigned a relatively small subset of their domains as values.

The algorithm has the following steps:

1) The requirement specification is given in the form of an LTL-formula φ. Negate this formula and transform it to an equivalent Büchi automaton S;
2) Apply abstraction to the concrete model with the current precision, calculate the abstract state space and represent it with an automaton M;
3) Calculate the synchronous product of the two automata $S \times M$. During each step of the product the model automaton steps first, then the specification automaton steps based on the target state of the model automaton.
4) Check the language emptiness of the product automaton $S \times M$;
    - If the language of the product is empty, then the model meets the correctness specification as no counterexamples were found;
    - If a counterexample is found in the abstract state space, then verify whether it is feasible in the concrete state space as well;
        - A feasible counterexample means that the model does not meet the correctness specification (i.e. is *unsafe*), as we found a contradicting trace;
        - If the counterexample isn't feasible in the concrete system (i.e. *spurious*), then refine the precision and jump to step 2.

When using a suitable language emptiness checking algorithm such as Nested DFS [8], the tasks of state space generation, calculation of the product automaton and language emptiness checking can be conducted together, which can result in a significant increase in performance. If these three tasks are carried out at the same time, then the model checking is said to happen "on-the-fly".

IV. Refinement

In this section we present a novel refinement method for predicate abstraction. The algorithm searches for counterexamples that have a "lasso"-like form. The first part of the counterexample is a path leading to an accepting state and the second part is a cycle which starts and ends in said accepting state. If such a counterexample is found, then an accepting run is possible, because by repeatedly traversing the cycle, an accepting state can be explored infinitely many times, fulfilling the Büchi acceptance condition.

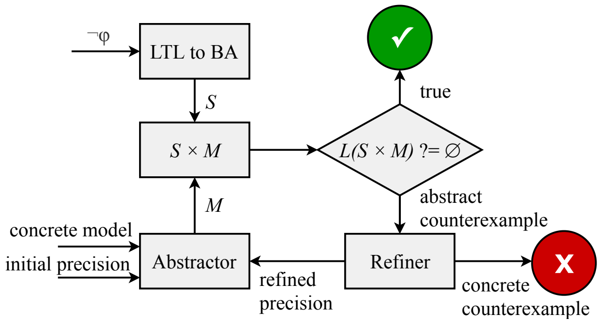

Fig. 1. Overview of CEGAR-based LTL model checking.

The CEGAR algorithm is usually used for reachability checking, where counterexamples are abstract paths leading from the initial state to an error state. When verifying these counterexamples the only thing that needs to be checked is whether such a path exists in the concrete model, whose states and transitions all correspond to the states and transitions of the abstract path. However, the fulfillment of this condition is required, but not enough, when analysing a cycle. A path that is not a cycle in the concrete model might appear as one in the abstract model.

We developed a novel counterexample refinement strategy that is capable of handling "lasso"-like counterexamples. The input is an abstract path $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, ..., op_{n-1}, (l_n, s_n))$ and an integer $1 \leq cycle \leq n$ that is the index of the initial state of the cycle, i.e. the recurrent accepting state ($s_{cycle} = s_n$). The path is first fed to the traditional CEGAR refinement algorithm presented in Section II-B. Based on the result of this algorithm, we have two options. If the algorithm finds that the path isn't traversable and returns a refined precision, then we simply return this refined precision. However, if the algorithm finds that the path is traversable, then we conduct further analysis to decide whether it is traversable in such a way that the initial and the end state of the cycle are the same concrete states.

Control locations are tracked explicitly during state space exploration, thus deciding whether two concrete states that belong to the same abstract state are identical can be done by comparing their data values (i.e. the values assigned to the variables in them). We construct a constraint $B \equiv \bigwedge_{v \in V} v^{\langle cycle \rangle} = v^{\langle n \rangle}$, which expresses that each variable has the same value in the initial and end state of the cycle, i.e. they are the same concrete states. We also construct the same formula that the refinement algorithm in II-B used to verify traversability, $A \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge ... \wedge s_{cycle}^{\langle cycle \rangle} \wedge op_{cycle}^{\langle cycle \rangle} \wedge ... \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$. By querying an SMT solver with the conjuntion of these two formulas, i.e., $A \wedge B$, we

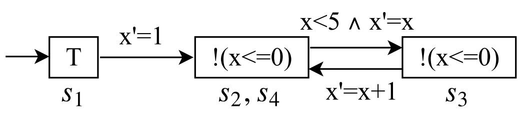

Fig. 2. Example of an abstract counterexample.

verify whether the counterexample is feasible. If this formula is satisfiable then the model does not meet the requirement specification (i.e. is *unsafe*) and a satisfying assignment is returned as counterexample. If the formula isn't satisfiable (i.e. is *spurious*), then we refine the precision by calculating an interpolant based on this formula.

Algorithm 1 Lasso refinement

```
    Input: $\sigma$: abstract path, $cycle$: initial index of cycle
    Output: ($unsafe$ or $spurious$, $\pi'$)
 1: procedure $lasso\_refine(\sigma, cycle)$
 2:     $res := refine(\sigma)$
 3:     if $res$ is $spurious$ then return $res$
 4:     else
 5:         $A \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge ... \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$
 6:         $B \equiv \bigwedge_{v \in V} v^{\langle cycle \rangle} = v^{\langle n \rangle}$
 7:         if $A \wedge B$ is feasible then return $(unsafe, \pi)$
 8:         else
 9:             $I \leftarrow$ get interpolant for ($A$, $B$)
10:             $\eta \leftarrow$ get satisfying assignment for $A$
11:             $\pi' \leftarrow$ create predicate from $I$:
12:                 replace all $v^{\langle n \rangle} \in I$ with $v$
13:                 replace all $v^{\langle cycle \rangle} \in I$ with $\eta(v^{\langle cycle \rangle})$
14:             return ($spurious$, $\pi \cup \pi'$)
```

To refine the precision we obtain an interpolant I for A and B. This interpolant is interpreted over $V^{\langle cycle \rangle}$ and $V^{\langle n \rangle}$, let's denote this with $I(v_1^{\langle cycle \rangle}, ..., v_k^{\langle cycle \rangle}, v_1^{\langle n \rangle}, ..., v_k^{\langle n \rangle})$. We also query the SMT solver for a satisfying assignment η to the formula A, which describes a concrete path $\sigma = ((l_1, c_1), \mathrm{op}_1, ..., \mathrm{op}_{n-1}, (l_n, c_n))$, where $c_{cycle} \neq c_n$. To ensure that the spurious counterexample described by η isn't found again during later explorations of the abstract state space, we need to extend our precision with a new predicate $\pi'(v_1, ..., v_k)$ that evaluates to $false$ in c_{cycle} and to $true$ in c_n (or vice versa), so that c_{cycle} and c_n get mapped to different abstract states. Formally, $\pi'(\eta(v_1^{\langle cycle \rangle}), ..., \eta(v_k^{\langle cycle \rangle})) = false$ and $\pi'(\eta(v_1^{\langle n \rangle}), ..., \eta(v_k^{\langle n \rangle})) = true$. To construct the predicate π' from the interpolant I, we replace the variables $V^{\langle n \rangle}$ with V, and $V^{\langle cycle \rangle}$ with values that are assigned to them by η. Formally, $\pi'(v_1, ..., v_k) := I(\eta(v_1^{\langle cycle \rangle}), ..., \eta(v_k^{\langle cycle \rangle}), v_1, ..., v_k)$.

If we evaluate $\pi'(\eta(v_1^{\langle n \rangle}), ..., \eta(v_k^{\langle n \rangle}))$, i.e. π' in c_n, we get $I(\eta(v_1^{\langle cycle \rangle}), ..., \eta(v_k^{\langle cycle \rangle}), \eta(v_1^{\langle n \rangle}), ..., \eta(v_k^{\langle n \rangle}))$, which is $true$, because of the first property of Craig interpolants ($A \rightarrow I$), from which it follows that if an assignment η satisfies A, then it also satisfies I.

Evaluating π' in c_{cycle} however, results in $I(\eta(v_1^{\langle cycle \rangle}), ..., \eta(v_k^{\langle cycle \rangle}), \eta(v_1^{\langle cycle \rangle}), ..., \eta(v_k^{\langle cycle \rangle}))$, which is $false$. In this case the variables $V^{\langle cycle \rangle}$ are assigned the same values as their counterparts $V^{\langle n \rangle}$, which means that B is $true$. It follows that I in this case is $false$, because of the second property of Craig interpolants ($I \wedge B$ is unsatisfiable),

We demonstrate the refinement process on the abstract counterexample in Fig. 2. The white rectangles represent abstract states with the applying predicates displayed inside them, the arrows represent transitions, the precision only contains one predicate, $(x \leq 0)$. The value of cycle and n is 2 and 4, respectively. We construct the following formulas based on this path:

$$A \equiv true \;\wedge\; x^{\langle 2 \rangle}{=}1 \wedge !(x^{\langle 2 \rangle}{\leq}0) \;\wedge\; x^{\langle 2 \rangle}{<}5 \wedge x^{\langle 3 \rangle}{=}x^{\langle 2 \rangle} \;\wedge\; !(x^{\langle 3 \rangle}{\leq}0) \;\wedge\; x^{\langle 4 \rangle}{=}x^{\langle 3 \rangle}{+}1 \wedge !(x^{\langle 4 \rangle}{\leq}0)$$
$$B \equiv x^{\langle 2 \rangle}{=}x^{\langle 4 \rangle}$$

By querying an SMT solver with the formula $A \wedge B$ we find that the counterexample in spurious, as $A \wedge B$ isn't satisfiable. The solver returns the interpolant $I \equiv x^{\langle 2 \rangle} < x^{\langle 4 \rangle}$ (note that this is only one of the possible interpolants). We request a satisfying assignment for A from the solver, and construct a predicate from I by replacing $x^{\langle 4 \rangle}$ with x and $x^{\langle 2 \rangle}$ with 1 (the value that is assigned to it in the satisfying assignment). Finally, we return that the counterexample is spurious, accompanied by the refined precision $(x \leq 0), (1 < x)$.

V. Conclusion

In our paper we examined LTL model checking and proposed a novel algorithm, which combines the advantages of counterexample-guided abstraction refinement and automata theoretic LTL model checking. We also proposed a novel refinement method for predicate abstraction. We implemented our algorithm in the Theta framework [11], but chose to omit experimental evaluation from this paper due to the lack of space.

References

[1] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. 2013.
[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. 2003.
[3] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
[4] Zhao Duan, Cong Tian, and Zhenhua Duan. Verifying temporal properties of c programs via lazy abstraction. 2017.
[5] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. 1996.
[6] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. 2019.
[7] Kenneth McMillan. Applications of craig interpolation to model checking. 2005.
[8] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. 2005.
[9] Robert Tarjan. Depth first search and linear graph algorithms. 1972.
[10] Wolfgang Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on Infinite Objects. 1990.
[11] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. 2017.