Learn Your Program

Hadar Frenkel\textsuperscript{1} *, Orna Grumberg\textsuperscript{1} and Sarai Sheinvald\textsuperscript{2}
\textsuperscript{1} Department of Computer Science, The Technion, Israel
\textsuperscript{2} Department of Software Engineering, ORT Braude College, Israel

Abstract—We introduce a synthesis framework for quantifier-free first-order LTL formulas over program variables. Such formulas are natural for specifying security requirements. Our framework uses automata learning to construct the program, as well as Hoare logic in order to reason about the program semantics and abstract the program states. Hoare logic also supply us with a proof of correctness for the program. We aim to learn not only the program, that is, the program states and transitions, but also to infer the program statements using abduction. Since this is an ongoing work, We discuss problems of termination, inference of Hoare predicates, and inference of new program statements. Our algorithm will allow automatically elimination of security vulnerabilities by constructing safe programs.

I. INTRODUCTION

In program verification, we verify that a program satisfies a given specification. We then say that the program is correct. In program synthesis, instead of trying to prove correctness of a given program, we wish to build a correct program, with respect to the specification. In this work we consider quantifier-free first-order LTL (FO-LTL) specifications, which, instead of atomic propositions, contain predicates over the program variables. FO-LTL is a natural logic for expressing security vulnerabilities within a program, such as array out of bound, or memory leaks. We then use program synthesis to construct safe programs, that contain no vulnerabilities of the specified kind. Moreover, we can use synthesis algorithms in order to eliminate security vulnerabilities detected using other methods (such as model checking). Once a vulnerability is uncovered, a synthesis algorithm is required to determine how to fix it.

An example for a specification in FO-LTL is \( \varphi_{GF} := G ((x = 0) \rightarrow F (x > 0)) \) over the program variable \( x \). We then wish to build a program \( P \), which all of its computations satisfy \( \varphi_{GF} \). We use \( \varphi_{GF} \) throughout the paper to demonstrate our approach.

In order to construct such a program, we use automata learning methods, derived from \( L^* \) algorithm for learning regular languages [1]. The resulted program is then in the shape of a finite automaton, that can be viewed as the control-flow graph of the program. From now on we refer to the program as an automaton, over the alphabet \( \Sigma \) of program statements. We later discuss the question of how we infer program statements. For now, we assume such an alphabet is given.

Unlike standard automata learning, the alphabet here is symbolic. Thus, syntactic checks for membership and equivalence queries, needed in the automata learning process, are no longer enough. We suggest a novel approach that combines words over program statements, together with predicates over the variables. The predicates behave as abstractions of the automaton states. To do so, our words are sequences of Hoare triplets [7] over the program statements. We thus learn an annotated program, where the annotations of the states provide us with a proof of correctness. Moreover, Hoare triplets allow us to adjust learning to semantic queries. They also allow us a succinct construction of the program and a more efficient teacher for the learning process.

II. LEARNING AUTOMATA OVER PROGRAM STATEMENTS

Let \( \varphi \) be a FO-LTL specification. The language of \( \varphi \), denoted by \( \mathcal{L}(\varphi) \), is the set of all computations satisfying \( \varphi \). Note that this definition does not restrict us to a certain alphabet. In the synthesis problem, we wish to learn a program \( P \), such that \( \mathcal{L}(P) \subseteq \mathcal{L}(\varphi) \). Recall that \( P \) is an automaton. Then, the above means that every computation of the program satisfies the specification. In case we have such a program \( P \), we say that \( P \) is correct with respect to \( \varphi \).

Given a specification \( \varphi \), we use a variation on \( L^* \) algorithm for automata learning [1] in order to learn a correct program \( P \) with respect to \( \varphi \). Our algorithm, as \( L^* \), consists of two entities: a learner, who preforms queries in order to construct an automaton for the language; and a teacher, who answers the queries. There are two types of queries; membership queries, in which the learner asks whether a given word is in \( \mathcal{L}(P) \); and equivalence queries, in which the learner presents a candidate automaton \( A \) and asks if the automaton is correct, i.e., if \( \mathcal{L}(A) \subseteq \mathcal{L}(\varphi) \). In case the automaton is not correct, the teacher returns a counterexample \( c \in \mathcal{L}(\varphi) \Delta \mathcal{L}(A) \) where \( \Delta \) is the symmetric difference operator. The learner then returns to the membership queries phase with an answer on the word \( c \), and tries to construct a new candidate automaton.

In the case of regular languages over finite alphabets, \( L^* \) algorithm is guaranteed to terminate. One question we are yet to resolve is termination in case of semantically irregular requirements. Note that every FO-LTL specification is syntactically regular, since we can build an automaton describing it, relating to the predicates in the specification as atomic propositions and ignoring their semantics, similar to [3].

Unlike the standard \( L^* \), which handles finite alphabets, here we use symbolic program statements. Therefore, we can no
longer ask syntactic questions. In order to allow symbolic construction, we use Hoare sequences. A Hoare sequence is infeasible, and we can choose to not query it. Thus, \( \langle \alpha_1, \ldots, \alpha_n \rangle \) is a valid Hoare triplet, and \( \forall i : \alpha_i \in \Sigma \), that is, \( \alpha_i \) is a program statement.

Before we discuss the nature of the membership and equivalence queries, we first introduce the teacher.

A. The Teacher: Modeling the Specification

In order to answer membership and equivalence queries, we build a tableau \( T_\varphi \) [2] out of the specification \( \varphi \). In the construction, we refer to the predicates in \( \varphi \) as atomic propositions. Nevertheless, we use the semantics of the predicates in order to find a more succinct tableau; we can remove states that are labeled with (at least) two predicates \( p_1 \) and \( p_2 \) such that \( p_1 \land p_2 \rightarrow \text{false} \).

Infinite Words vs. Finite Computations: Usually, LTL formulas describe ongoing computations, whereas the programs we construct consist of finite computations. For that, we can use semantics of LTL over truncated paths, as proposed for example in [5] or in [6].

B. Membership Queries

Given a Hoare sequence \( s = \langle p_1, \alpha_2 \rangle \cdot \langle p_2, \alpha_3 \rangle \cdots \langle p_n, \alpha_{n-1} \rangle \cdot \langle p_n \rangle \) where each \( \langle p_i, \alpha_i(p_{i+1}) \rangle \) is a valid Hoare triplet, and \( \forall i : \alpha_i \in \Sigma \), that is, \( \alpha_i \) is a program statement.

Lemma 1. \( s \) is semantically contained in \( T_\varphi \) iff \( s \) satisfies \( \varphi \).

Example 1. Consider \( \varphi \in L \) from Section I and the Hoare sequence \( s = \langle \text{true}, x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x > 0 \rangle \). Since the sequence \( \langle \text{true}, x = 0 \rangle \cdot \langle x > 0 \rangle \) satisfies \( \varphi \), it holds that \( s \in L(T_\varphi) \).

C. Constructing a Candidate Program

Every predicate used in the Hoare sequences is in fact an abstraction of a state in the program automaton. Now, in order to construct the automaton, we need to know the transition relation. Usually, for each state there must be exactly one transition with each alphabet letter, to make the automaton deterministic and complete. Here, we look for semantic determinism. We use the Hoare triplets to construct transitions for every possible value at this abstract state; we can ignore values that result in infeasible words. Hoare triplets allow us then to eliminate some transitions, without asking the teacher, since they are infeasible. Since every state is annotated with a predicate over the program variables, we can know, for example, that the word \( \langle \text{true}, x < 0 \rangle \cdot x > 0 \cdot \text{false} \) is infeasible, and we can choose to not query it. Thus, to construct the transition relation, for every existing state represented by a Hoare sequence \( s \), we query about \( s \cdot \alpha(p) \), where \( s \cdot \alpha(p) \) is feasible. We then follow \( L^* \) to construct a complete transition relation and present a candidate automaton.

D. Equivalence Queries

Once a candidate automaton is constructed, the learner asks the teacher whether it is correct. For a specification \( \varphi \) and a candidate \( A \), we can check if \( L(A) \subseteq L(\varphi) \) by constructing a tableau for \( \neg \varphi \) and check if \( A \cap T_{\neg \varphi} = \emptyset \).

E. Termination

We are yet to study the termination of the process. In particular, we study the termination of each candidate generation phase. Once a candidate \( A \) is presented, we can artificially create a correct program by taking \( P = A \cap T_\varphi \). Thus, we can manually terminate the process given a candidate program, in case the whole process does not converge.

III. Program Alphabet

As an initial alphabet for the first iteration of membership and equivalence queries, we aim to infer program statements from the specification. For example, from the specification \( \varphi \in L \) we can infer the program statements \{ if \( (x \leq 0) \), if \( (x > 0) \), if \( (x = 0) \), if \( (x < 0) \), if \( (x = 1) \) \}. The assignment \( x := 1 \) is of course a very simplistic solution for \( x > 0 \) appears in the specification. We work towards obtaining more general statements in the process of learning. In addition, we look for examples for which such predefined alphabet is not enough. In such cases, we wish to use abduction [4] in order to learn new alphabet letters for the program. We look for characterizations of failed membership or equivalence queries, for which we can say that in order to satisfy the specification we need new program statements.

IV. Summary

We suggest a new synthesis framework for FO-LTL specifications, using automata learning and Hoare logic. We use Hoare triplets to reason about the semantics of the program and suggest a succinct program construction. We aim to learn program statements when needed, using abductive inference.

We can then use FO-LTL to describe previously detected security vulnerabilities, and apply our algorithm to construct programs free from these vulnerabilities.

A. Open Questions

Termination of Learning: we study the terms for termination of the learning process, since there might be cases for which the membership queries phase does not converge into a candidate automaton.

Alphabet Inference: we are yet to determine how to infer the initial alphabet out of the specification, and to decide when to use abduction in order to infer new alphabet letters.

Use of Hoare Triplets: throughout the paper we assumed the predicates for Hoare triplets are known, but it is not the case. We study the inference of predicates for Hoare triplets. This is crucial for our algorithm since these predicates behave as states abstraction and proof of correctness to the program.
REFERENCES