# Symbolic Grey-Box Learning of Input-Output Relations

*Domagoj Babic*
*Matko Botincan*
*Dawn Song*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-59
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-59.html

May 3, 2012

# Symbolic Grey-Box Learning of Input-Output Relations

Domagoj Babić[1] Matko Botinčan[2] Dawn Song[1]

[1]UC Berkeley [2]University of Cambridge

## Abstract

Learning of stateful models has been extensively used in verification. Some applications include inference of interface invariants, learning-guided concolic execution, compositional verification, and regular model checking. Learning shows a great promise for verification, but suffers from two fundamental limitations. First, learning stateful models over concrete alphabets does not scale in practice, as alphabets can be large or even infinite in size. Second, learning techniques produce conjectures, which might be neither over- nor under-approximations, but rather some mix of the two. The new technique we propose — Sigma$^*$ — overcomes these problems by combining black- and white-box analysis techniques: learning and abstraction. Such grey-box setting allows inspection of the internal symbolic state of the program, allowing us to learn symbolic transducers with input and output alphabets ranging over finite sets of symbolic terms. The technique alternates between symbolic conjectures and sound over-approximations of the program. As such, the technique presents a novel twist to the more standard alternation among under- and over-approximations often used in verification. Sigma$^*$ is parameterized by an abstraction function and a class of symbolic transducers. In this paper, we develop Sigma$^*$ parameterized by a variant of predicate abstraction, and $k$-lookback symbolic transducers — a new variant of symbolic transducers, for which we present learning and separation sequence computation algorithms. Verification of such transducers is, for instance, important for security of web applications and might find its applications in other areas of verification. The main technical result we present is that Sigma$^*$ is complete relative to abstraction function.

## 1. Introduction

Grammatical inference techniques for learning various types of automata from input-output sequences have been applied in a wide variety of verification settings: inference of interface invariants [3], learning-guided concolic execution [15], compositional verification [17], and regular model checking [25]. In those settings, learning is performed in a black-box manner, even though source code is usually available. As a consequence, the classical Angluin's $L^*$ algorithm [4] used for learning works over concrete alphabets, as it cannot inspect the internal symbolic state of the system whose input-output relation is to be learned. The concrete treatment of the alphabet results in a state explosion problem, because alphabets are typically large or even infinite in practice. Since the complexity of $L^*$ is quadratic in the size of the alphabet and the number of states, such approaches scale poorly.

By allowing $L^*$ to inspect the internal symbolic state of the program for which we wish to learn an input-output relation, we obtain a more powerful symbolic version of $L^*$. Generating concrete input sequences, analyzing how they are processed, and observing the symbolic responses, our approach learns symbolic input-output relations. As a consequence, the approach is applicable even when the input alphabets are infinite, and promises better scalability in general.

The classical $L^*$ merely conjectures input-output relations, without giving any guarantees that conjectures are either under- or over-approximations of the input-output relation implemented by the program. To check conjectures, $L^*$ relies upon an equivalence checking oracle. Such oracles are often either unavailable in practice, computationally intractable, or very difficult to implement as the input-output relation being learned and the program are often very different formal objects. We show how such oracles could be implemented for a class of programs by equivalence checking of deterministic approximations computed by an $L^*$-like algorithm and sound possibly non-deterministic over-approximations of the program being learned. We refer to such a learning framework parameterized by a class of symbolic transducers and an abstraction function as $\Sigma^*$. Notably, $\Sigma^*$ is the first learning framework working fully at the symbolic level.

In this paper, we develop $\Sigma^*$ parameterized by $k$-lookback symbolic transducers ($\overleftarrow{k}$-SLT) and a variant of predicate abstraction. We introduce $\overleftarrow{k}$-SLT as a variant of symbolic finite-state transducers (SFT) with registers and simple updates (SFTR) [11, 12]. In our variant, all the states are final. Without that requirement, it is unclear how to automatically learn an SFT(R), as they allow inherent ambiguity in where the output is produced. Indeed, existing concrete finite-state transducer learning algorithms [34, 36, 39] require all states to be final. Also, we impose a restriction on how registers are used to simplify learning and exposition, but we believe that restriction is not fundamental. The restriction allows us to completely abstract away the registers. We show that with $\overleftarrow{k}$-SLTs we can model faithfully 67% of sanitizers from the Google AutoEscape framework, and 92% with a simple generalization.

We also present a learning algorithm for $\overleftarrow{k}$-SLTs. To check the learned conjectures, we exploit the fact that our conjectures are deterministic. While equivalence checking is undecidable even for simple concrete non-deterministic $\varepsilon$-free generalized sequential machines [30], equivalence checking of deterministic and non-deterministic finite-state transducers is decidable [18], and we present a generalization of Demers et al.'s [18] algorithm for our symbolic setting.

The role of the abstraction in $\Sigma^*$ is to synthesize an over-approximation of the program whose input-output relation we wish to learn. In the particular instantiation of $\Sigma^*$ we develop in this paper, we use a variant of the standard well-known predicate abstraction [6, 23]. Predicate abstraction serves the purpose of abstracting away the internal computation of a program, while we leave the input-output data flow intact. The resulting (possibly non-deterministic) abstraction is still unsuitable for comparison with conjectures, which are in the form of $\overleftarrow{k}$-SLTs. We show how to synthesize non-deterministic $\overleftarrow{k}$-SLTs from the computed abstractions.

The most important technical result of this paper is that we show that $\Sigma^*$, parameterized by $\overleftarrow{k}$-SLTs and predicate abstraction, is relatively complete for a a family of infinite-state programs, which we call $\overleftarrow{k}$-SLT programs. The framework is complete relative to the chosen abstraction function; we show that $\Sigma^*$ is complete if the abstraction function eventually terminates producing the strongest inductive invariant of program's transition relation (see [8, 31]). Different classes of programs could be learned by parameterizing $\Sigma^*$ in a different way. For instance, subsequential symbolic transducers could be learned with a symbolic variant of Vilar's [39] algorithm and by adjusting the abstraction accordingly. For an even broader class of transductive programs that read as inputs and produce as outputs sequences of symbols, $\Sigma^*$ might not terminate. However, the learned incomplete symbolic conjectures might still be useful for various applications, like MACE [15].

The main motivation for $\Sigma^*$ came from the recent work of Bjørner et al. [11, 12]. They develop the theory of symbolic transducers and show how such transducers can be used for verification of web sanitizers, by checking their equivalence, idempotence, and commutativity. In addition, they also discuss a number of other applications: detection of malicious programs, image blurring, and location privacy. To construct formal objects amenable to verification, they had to invest several hours of a human expert's time into manual synthesis of each SFT from code. In contrast, $\Sigma^*$ can completely automatically infer 10 out of 14 publicly available examples used to evaluate Bek [27]. We also draw inspiration from the recent work on MACE [15]. MACE uses an optimized version of $L^*$ to learn an approximation of a program's search space. That approximation guides concolic execution, which in turn gets deeper into the search space, discovering new input and output symbols used for refining the approximation. While showing strong experimental results, MACE offers absolutely no guarantees on completeness and is computationally very expensive, because it operates on large concrete transducers. $\Sigma^*$ could be used to realize a symbolic variant of MACE, offering relative completeness for certain classes of programs, and potentially better scalability in general. $\Sigma^*$ might also find its applications in compositional verification, inference of interface invariants, and regular model checking.

We claim the following contributions:

**Parameterized Learning Framework $\Sigma^*$.** We introduce an automated learning framework $\Sigma^*$ for learning symbolic input-output relations of infinite-state programs. The key insight behind $\Sigma^*$ is that we can use an over-approximation to detect convergence. Another novel insight we put forward is that by combining black-box learning with white-box abstraction, we reap the benefits of both.

***k*-Lookback Symbolic Transducers.** When developing a specialization of $\Sigma^*$, we introduce a variant of SFTRs [11, 12] that is amenable to learning. Additionally, we restrict how the registers are used to simplify learning and exposition. We present a learning algorithm for $\overleftarrow{k}$-SLTs, which is, to the best of our knowledge, the first learning algorithm operating fully at the symbolic level.

**Synthesis of Program Abstractions into $\overleftarrow{k}$-SLTs.** We show how to translate predicate abstractions of programs being learned to $\overleftarrow{k}$-SLTs and how to refine the abstraction in our setting, when the abstraction is too coarse. The key novelty in this synthesis is the computation of a finite representation of a potentially infinite input-output relation implemented by the program.

**Relative Completeness Results.** The main technical result of this paper is that $\Sigma^*$ converges to a complete and sound model for certain classes of infinite-state programs, as long as the abstraction function eventually terminates producing the strongest inductive invariant of program's transition relation. Effectively, this results shows that the completeness of learning can be reduced to the completeness of abstraction.

## 2. Related Work

In this section, we position our work with respect to the most relevant existing literature. After discussing symbolic transducers and their learning, we end with a discussion of connections with predicate abstraction, which we use in our specialization of $\Sigma^*$.

### 2.1 Symbolic Transducers

Symbolic finite-state transducers have become a popular topic lately [2, 11, 12, 27]. Van Noord and Gerdemann [38] introduced the first simple symbolic transducers. The expressivity of those transducers is severely limited as they are not able to establish functional dependencies from input to output. In a series of papers [11, 12, 27], a group of authors proposed symbolic finite-state transducers (SFTs) with registers (SFTRs), where input (resp. output) symbols range over predicates (resp. terms). Bjørner and Veanes [11] show an effective composition construction and prove that equivalence is decidable for single-valued SFTRs with simple updates, as long as the underlying theory is decidable. Unfortunately, their proof is not constructive and does not reveal how to efficiently compute a separating sequence. $\overleftarrow{k}$-SLTs are less expressive than SFTRs. We carefully limited the expressivity of SFTRs (to obtain $\overleftarrow{k}$-SLTs) so as to be able to represent interesting real-world examples. able to learn and constructively equivalence check learned $\overleftarrow{k}$-SLTs.

One feature of SFTRs makes them difficult to learn — the set of final states of an SFTR can be a strict subset of all states. Transducers with some non-final states are partial functions that accumulate the output and yield it only when the last reached state is final. Furthermore, the existence of non-final states allows ambiguity of where the output is created. Thus, it is not surprising that the existing algorithms for learning concrete transducers [34, 36, 39] require all states to be final. Similarly, we require all the states of our symbolic transducers to be final. We also impose a restriction on how registers are used. In particular, we require that registers can contain input symbols read at most $k$ transitions before the last transition. That restriction allows us to abstract away registers, while still allowing complex bounded functional dependencies among inputs and outputs. We do not believe that restriction to be fundamental. As the set of predicates used as transition guards in SFTRs and $\overleftarrow{k}$-SLTs is finite, predicates partition register valuations into a finite set of equivalence classes, which could be learned at the cost of having a more complex learning algorithm. Similar partitioning arguments were used in the proofs of decidability of equivalence of single-valued SFTRs [11] and streaming transducers [2]. In streaming transducers [2], there are two type of registers — one for symbols and the other for strings. At every step, a streaming transducer can make decisions based on the current state, the tag (ranging over a finite set of values) of the next input symbol, and the ordering between next input symbol and symbols stored in data registers. Each value stored in string registers can be either copied or concatenated with a symbol, but has to be used only once. They show an intricate proof of decidability of equivalence of streaming transducers. Inference of both SFT(R)s [11, 12] and streaming transducers [2] has been left as an open question.

### 2.2 Learning of Symbolic Transducers

Little work has been done on learning of symbolic transducers. The work we are aware of [1, 10, 29] focuses on a limited form of symbolic transducers that have simple guards with a fixed number of parameters and can either update registers or output a sequence of concrete values in every transition. Such transducers have a Nerode relation that partitions their states into a finite number of equivalence classes, which makes their learning possible. All of the above approaches learn over concrete alphabets, translating the obtained concrete transducer to a symbolic one, and assume existence of a suitable equivalence checking oracle. Such approaches are exponential in *both* the length of counterexamples and the number of parameters the guard predicates can have, as well as in the number of registers. We take a different approach, and by allowing $\Sigma^*$ to inspect the internal symbolic state of the program, we can handle arbitrarily complex guard and output expressions, as long as (1) the program's input-output relation can be described using finitely many such expressions, and (2) there exists a solver capable of checking satisfiability of guards and (in)equality of output expressions. There is another important point of difference. Instead of just assuming existence of a suitable equivalence checking oracle, like the prior work [1, 10, 29], we propose an efficient algorithm for checking the learned conjectures against a synthesized abstraction. Most importantly, the alternation between learning and abstraction allows us to detect convergence, assuming an abstraction that can infer the strongest invariant of program's transition relation.

Next, we will clarify the connection of $\Sigma^*$ to a few other lines of work. Elkind et al. [19] propose a model checking approach for systems in which some components are completely black- and some completely white-box. They call the resulting approach grey-box checking. We use the phrase "grey-box" differently to refer to a combination of a learning algorithm that is inherently black-box and the white-box analysis required to answer the membership queries and to compute an abstraction. Alur and Černý [2] propose an algorithm for inferring automata describing safe behaviors of interfaces. They use the standard $L^*$ algorithm for inferring conjectures and check that the conjectures are subsumed by the predicate abstraction of the code implementing the interface, i.e., they check that the conjecture is safe. The safety check can be done efficiently, while finding the most permissive (complete) model is NP-hard in their setting. In contrast, our focus is on the input-output behavior of programs and inference of symbolic transducers describing that behavior. Since we want to find the exact transducers modeling the program, we find discrepancies among conjectures and abstractions by computing a separating sequence between the two.

### 2.3 Predicate Abstraction

The specialization of $\Sigma^*$ we develop in this paper uses predicate abstraction [7, 23] to generate a finite-state model of the unbounded internal state space of the program. Predicate abstraction has been successfully applied in counterexample-guided abstraction refinement schemes for checking safety properties. Tools like SLAM [6], BLAST [26], IMPACT [33], and SATABS [16] have demonstrated that predicate abstraction scales well to real-world programs with intricate control-flow patterns. Compared to the regular predicate abstraction, our goal is different—we infer the exact input-output relation of a program so that we can check non-safety properties such as equivalence, idempotence and commutativity [12].

Collecting path conditions as in concolic testing [21] has been previously combined with predicate abstraction in the SYNERGY algorithm [24] and its descendants [9, 22]. $\Sigma^*$ combines *must* (learning) and *may* (abstraction) analyses to learn a more compact input-output relation, while SYNERGY uses *may-must* combination to check safety properties faster. Similarly as safety properties could be checked with MAY analysis only, input-output relations could be learned with our predicate abstraction (Section 7.2) only, by refining abstractions until the relation becomes single-valued. However, such relations can be unboundedly larger than the ones learned by $\Sigma^*$.

The completeness of predicate abstraction depends on the choice of predicates for abstracting the state space. Completeness of predicate abstraction is an active research area [8, 31] and a number of heuristics exist that are often complete in practice [5, 20].

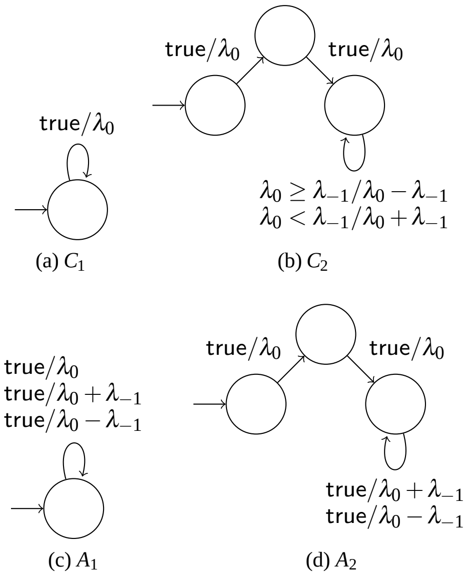

true/$\lambda_0$
(a) $C_1$
true/$\lambda_0$
true/$\lambda_0$
$\lambda_0 \geq \lambda_{-1}/\lambda_0 - \lambda_{-1}$
$\lambda_0 < \lambda_{-1}/\lambda_0 + \lambda_{-1}$
(b) $C_2$
true/$\lambda_0$
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$
(c) $A_1$
true/$\lambda_0$
true/$\lambda_0$
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$
(d) $A_2$

Figure 1: Examples of Learning Conjectures and Abstractions.

Assuming a predicate selection oracle that eventually yields enough predicates for constructing the strongest invariant of the program's transition relation, one can show relative completeness of predicate abstraction [8, 31]. We found that for the examples used in this paper, a simple oracle mining predicates from sequences of predicates and output terms obtained from the concolic execution of the program [13, 21, 35] is sufficient. More generally, we prove that the specialization of $\Sigma^*$ developed in this paper is relatively complete to predicate abstraction.

## 3. A Motivating Example

We begin by showing how $\Sigma^*$ works on the following example.

```
prev = 0; cur = 0; i = 0;
while (true) {
   cur = in();
   if (i < 2)           out(cur);
   else
      if (cur < prev) out(cur + prev);
      else            out(cur − prev);
   prev = cur;
   i++;
}
```

The program reads input in an infinite loop using command **in**(), which reads the next input symbol, or halts if there is no more input to read. In the first two iterations, the program prints the last read symbol (using command **out**()), and in every subsequent iteration it outputs the sum or difference of the last two symbols, depending on their relative ordering. Assuming symbols from an infinite alphabet, the program is infinite-state.

$\Sigma^*$, parameterized by $\overleftarrow{k}$-SLT and predicate abstraction, alternates between predicate abstraction and learning. The details of the learning algorithm are intricate, and will be presented later. Familiarity with the literature on $L^*$ [4, 36] and concolic execution [13, 21, 35] would be helpful for developing intuition on how such an algorithm might work.

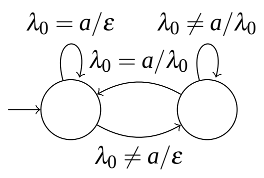
We represent learned conjectures and synthesized abstractions in the example as $\overleftarrow{1}$-SLTs. Each transition is labeled by a predicate-term pair. The transition is taken when the predicate evaluates to true for a given sequence of concrete input symbols. (We define symbolic evaluation formally later.) The second element of the pair describes symbolically the computed output. Predicates and terms are expressions over input symbols and constants. Since $\overleftarrow{k}$-SLTs are allowed to read only the current and the last $k$ symbols, we can relativize all references to input symbols with respect to the current position of the input tape head. We use $\lambda_0$ to denote the current symbol and $\lambda_{-i}$ to denote symbols read $i$ transitions before. With symbolic sequences like $\vec{p} = \lambda_0 \cdot (\lambda_0 > \lambda_{-1}) \cdot (\lambda_0 > \lambda_{-1})$, we represent symbolic path conditions, i.e., an equivalence class of a potentially infinite number of paths. Before solving such sequences with a solver, we need to derelativize them and compute a suitable formula passed to the solver. In the case above, the solver might return a concrete sequence $0 \cdot 1 \cdot 2$ satisfying $\vec{p}$.

$\Sigma^*$ begins processing the example by learning the first conjecture $C_1$ (Figure 1a), which is an under-approximation, and compares it against predicate abstraction $A_1$ of the program with respect to an empty set of predicates (Figure 1c). Note that, unlike in the standard predicate abstraction, we leave the input-output behavior intact. Equivalence checking of $C_1$ and $A_1$ produces the first symbolic counterexample, say a predicate $\vec{p}_1 = \mathsf{true}$ and output $\vec{\mathbf{t}}_1 = (\lambda_0 + \lambda_{-1})$, which is a behavior captured by the abstraction, but not the conjecture. Any concrete sequence satisfying those two formulas is a concrete counterexample. For example, take input $\lambda_{-1} = 1, \lambda_0 = 1$. The program will produce output $\lambda_0$ for that input without reading $\lambda_{-1}$, showing that $\vec{p}_1$ is a spurious symbolic counterexample, because symbolic outputs do not match. We refine the abstraction with respect to a set of predicates, obtained from some predicate selection oracle (e.g, [20, 37]). Suppose the oracle picks $\{i = 0, i < 2, i \geq 2\}$. The refined abstraction $A_2$ is shown in Figure 1d. Equivalence checking of $A_2$ and $C_1$ produces the second symbolic counterexample, say $\vec{p}_2 = \mathsf{true} \cdot \mathsf{true} \cdot \mathsf{true}$ and $\vec{\mathbf{t}}_2 = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_{-1})$. Any sequence of three symbols will satisfy $\vec{p}_2$ and it turns out that counterexample is not spurious. Thus, we have to refine the conjecture.

After processing the counterexample, $\Sigma^*$ learns conjecture $C_2$ shown in Figure 1b. $C_2$ is correct, but $\Sigma^*$ does not know that yet, because equivalence checking of $C_2$ and $A_2$ returns the third counterexample, say $\vec{p}_3 = \mathsf{true} \cdot \mathsf{true} \cdot (\lambda_0 < \lambda_{-1})$ and $\vec{\mathbf{t}}_3 = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 + \lambda_{-1})$. The third counterexample is not spurious either. Thus, once again, we need to refine the abstraction. Suppose the oracle extends the set of predicates with $\lambda_0 < \lambda_{-1}$. Predicate abstraction computes $A_3$, which is equivalent to $C_2$, and the process terminates correctly inferring program's input-output relation.

For the following example with nested loops, $\Sigma^*$ terminates inferring a $\overleftarrow{0}$-SLT shown below.

```
while (true) {
    x = in();
    while (x != 'a') {
        x = in();
        out(x);
    }
}
```


$\lambda_0 = a/\varepsilon$
$\lambda_0 \neq a/\lambda_0$
$\lambda_0 = a/\lambda_0$
$\lambda_0 \neq a/\varepsilon$

## 4. Notation and Terminology

In this section, we introduce the notation used throughout the paper. Let $\mathbb{Z}$ be the set of integers, $\mathbb{N} = \{i \in \mathbb{Z} \mid i > 0\}$ the set of naturals, and $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$ the set of booleans. Let $D$ be a set; by $D^*$, we denote the free monoid generated by $D$ with concatenation as the operation and the empty word $\varepsilon$ as identity. We often refer to words as sequences. For sequences of predicates (resp. terms), we write $\vec{p} = p_1 \dots p_m$ (resp. $\mathbf{t} = t_1 \dots t_n$). During learning, we treat a sequence of terms ($\mathbf{t}$) as a single object, to simplify exposition and the algorithms. For sequences of sequences of terms, we write $\vec{\mathbf{t}}$. We denote the length of a sequence as usual ($|\vec{p}|$). We refer to the $i$-th element of a sequence as $\vec{p}|_i = p_i$ and to a subsequence $\vec{p}|_{[i,i+k]} = p_i \dots p_{i+k}$. For function $f \colon D \to C$ and $D' \subseteq D$, we write $f|_{D'}$ to denote the restriction of $f$ on $D'$. For function $f$ and sequences $\vec{a}$ and $\vec{b}$ such that $n = |\vec{a}| = |\vec{b}|$, we write $f[\vec{a} \leftarrow \vec{b}]$ to denote the function $f'$ such that for all $i \in \{1, \dots, n\}$, $f'(\vec{a}|_i) = \vec{b}|_i$ and for all $x \notin \{\vec{a}|_1, \dots, \vec{a}|_n\}$, $f'(x) = f(x)$. We use the notation $\mathbf{t}[x \mapsto y]$ to denote the term obtained from $\mathbf{t}$ by simultaneously replacing all free occurrences of $x$ with $y$. A transduction from an input alphabet $\Gamma_1$ to an output alphabet $\Gamma_2$ is a partial function $\Gamma_1^* \rightharpoonup \Gamma_2^*$. Finally, we use $\_a$ to denote implicit existential quantification when we care about the named variable and "_" when we do not.

## 5. Transductive Programs

We begin the technical part of this paper by formalizing the class of programs that transform an input stream into an output stream of symbols from a possibly infinite-sized alphabet. Many types of programs behave in such a way, e.g., sanitizers, client and server programs exchanging messages over the network, etc.

### 5.1 Program Representation

We now formally define our syntactic representation of programs. Let $\mathsf{V} = \mathsf{V_E} \cup \mathsf{V_D}$ be the set of program variables divided into the sets of internal and data variables. Internal variables are used in internal computations of the program, while data variables store the data read from the input and can be used for constructing the output. We build internal expressions $Exp[\mathsf{V_E}]$, data expressions $DExp[\mathsf{V_D}]$, boolean expressions $BExp[\mathsf{V}]$, and atomic commands $Cmd[\mathsf{V}]$ with the following grammar:

$$\begin{array}{rcll}
e & ::= & \mathsf{k} \mid v \mid e + e \mid e - e \mid \dots & \in Exp[\mathsf{V_E}] \\
ds & ::= & \mathsf{k} \mid x & \\
d & ::= & ds \mid d + d \mid d - d \mid \dots & \in DExp[\mathsf{V_D}] \\
b & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg b \mid b \wedge b \mid b \vee b \mid & \\
 & & d = d \mid d \neq d \mid d < d \mid d \leq d \mid \dots \mid & \\
 & & e = e \mid e \neq e \mid e < e \mid e \leq e \mid \dots & \in BExp[\mathsf{V}] \\
c & ::= & \mathsf{assume}(b) \mid v := e \mid x := ds \mid & \\
 & & x := \mathsf{in}() \mid \mathsf{out}(d) & \in Cmd[\mathsf{V}]
\end{array}$$

where $\mathsf{k} \in \mathbb{Z}$, $v \in \mathsf{V_E}$ and $x \in \mathsf{V_D}$. As long as the quantifier-free theory of $DExp[\mathsf{V_D}] \cup Exp[\mathsf{V_E}]$ with equality and satisfiability of $BExp[\mathsf{V}]$ are decidable, the actual operators and predicates in the grammar are irrelevant, hence the elisions. The command $x := \mathsf{in}()$ reads an integer from the input stream and stores it to $x$ if the input is available, otherwise it causes the program to halt. The command $\mathsf{out}(d)$ writes the value of the expression $d$ to the output stream.

We represent programs using control-flow automata [26] over the language of atomic commands (rule $c$ in the above grammar). The control-flow automaton is determined by a set of control nodes $\mathsf{L}$ containing a distinguished node $\mathsf{s_L} \in \mathsf{L}$ representing the starting point of the program and a function $\mathsf{succ} \colon \mathsf{L} \times Cmd[\mathsf{V}] \rightharpoonup \mathsf{L}$ representing labeled edges. All nodes either have a single successor or have all outgoing edges labeled with a label of the form $\mathsf{assume}(b)$. In the latter case, we assume that all corresponding predicates $b$ are mutually exclusive and that their disjunction is a tautology.

### 5.2 Symbolic Semantics

Given a program $\mathscr{P}$ we now define its symbolic semantics. Symbolic semantics represent executions of $\mathscr{P}$ on all inputs. We formalize the contents of the input and the output stream by using sets of ghost variables $\mathsf{V_I} \triangleq \{in_1, in_2, \dots\}$ ($in$-variables) and $\mathsf{V_O} \triangleq \{out_1, out_2, \dots\}$ ($out$-variables), respectively. On a particular run of $\mathscr{P}$ all except finitely many of these variables are unde-

$$
\begin{array}{rcl}
[\![\mathsf{k}]\!]_\sigma & = & \mathsf{k} \\
[\![x]\!]_\sigma & \doteq & \begin{cases} \sigma(x) & \text{if } x \in \mathsf{V}, \\ x & \text{if } x \in \mathsf{V_I}, \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![e \pm f]\!]_\sigma & \doteq & [\![e]\!]_\sigma \pm [\![f]\!]_\sigma
\end{array}
$$

Figure 2: Symbolic Evaluation of Expressions. Symbol $\sigma$ denotes a map of type $\mathsf{Mem}$.

$$
\begin{array}{lcl}
(n,\sigma,\rho,i,j) & \overset{\mathsf{assume}(b)}{\rightsquigarrow} & (n',\sigma,\rho,i,j) \\
(n,\sigma,\rho,i,j) & \overset{x:=e}{\rightsquigarrow} & (n',\sigma[x \leftarrow [\![e]\!]_\sigma],\rho,i,j) \\
(n,\sigma,\rho,i,j) & \overset{x:=\mathsf{in}()}{\rightsquigarrow} & (n',\sigma[x \leftarrow [\![in_i]\!]_\sigma],\rho,i+1,j) \\
(n,\sigma,\rho,i,j) & \overset{\mathsf{out}(d)}{\rightsquigarrow} & (n',\sigma,\rho[out_j \leftarrow [\![e]\!]_\sigma],i,j+1)
\end{array}
$$

Figure 3: Symbolic Semantics of Commands.

fined, and for those which are defined, $in_i$ corresponds to the $i$-th element of the input stream, and $out_j$ to the $j$-th element of the output stream. By $\vec{in}$ we refer to the sequence of *in*-variables $in_1 in_2 \ldots$ and we use $\ell_{\mathsf{in}}$ to represent the length of a sequence of symbols in the input stream.

We evaluate program variables on a memory defined as a map $\mathsf{Mem} \triangleq (\mathsf{V_E} \rightharpoonup \mathbb{Z}) \cup (\mathsf{V_D} \rightharpoonup DExp[\mathsf{V_I}])$ and we execute commands symbolically on states of the form $\mathsf{State} \triangleq \mathsf{L} \times \mathsf{Mem} \times \mathsf{OutStr} \times \mathbb{N} \times \mathbb{N}$. The first component is the set of control nodes. The third component represents the symbols in the output stream as $\mathsf{OutStr} \triangleq \mathsf{V_O} \rightharpoonup DExp[\mathsf{V_I}]$ (although technically a part of the environment, we keep the output stream component in the state to simplify the exposition). The last two components of the state, the *in*-index and the *out*-index, store indices of the next element in the input stream and the output stream, respectively. We denote the components of a state $s \in \mathsf{State}$ by $s.n$, $s.\sigma$, $s.\rho$, $s.i$, and $s.j$, respectively. The initial state is $s_0 = (\mathsf{s_L}, \emptyset, \emptyset, 1, 1)$. Figure 2 shows the semantics of expressions.

Figure 3 shows symbolic semantic rules $\rightsquigarrow \subseteq \mathsf{State} \times C \times \mathsf{State}$ that define the effect of each command on a state. In each rule we have $n' = \mathsf{succ}(n,c)$. We write $s \overset{c}{\rightsquigarrow} s'$ to denote the transition $(s,c,s') \in \rightsquigarrow$ and say that $s'$ is a $\overset{c}{\rightsquigarrow}$-successor of $s$. Applying symbolic semantic rules to all states gives rise to a labeled transition system (LTS) $\mathscr{T} = (\mathsf{State}, C, \rightsquigarrow)$ with $\mathsf{State}$ as the set of states, $C \subseteq Cmd[\mathsf{V}]$ as the finite set of labels, and $\rightsquigarrow$ as the labeled transition relation.

### 5.3 Traces and Path Predicates

The LTS $\mathscr{T}$ representing symbolic semantics is infinite in general as it encodes all possible executions of $\mathscr{P}$ with respect to any input. Given a concrete input $\vec{a} \in \mathbb{Z}^*$, we can construct a sequence of states from $\mathscr{T}$ representing the symbolic trace of $\mathscr{P}$ on that input. We start with the initial state $s_0$ and at each step we follow the transition $s_k \overset{c}{\rightsquigarrow} s_{k+1}$, such that if $c$ is of the form $\mathsf{assume}(b)$ then $b$ is true in $s_k.\sigma[\vec{in} \leftarrow \vec{a}]$. We stop if we ever reach a state $s_f = (n_f, \sigma_f, \rho_f, i_f, j_f)$, which we call ending, such that the *in*-index $i_f$ equals $\ell_{\mathsf{in}} + 1$. We define the symbolic trace of $\mathscr{P}$ on input $\vec{a}$, denoted $\tau_{\mathscr{P}}(\vec{a})$, as the finite sequence $s_0 s_1 \ldots s_f$ if an ending state is reached, or as the infinite sequence $s_0 s_1 \ldots$ otherwise.

**Definition 1.** *We say that $\mathscr{P}$ is* transductive *if for every input $\vec{a}$, $\tau_{\mathscr{P}}(\vec{a})$ is finite and no transition with a label $\mathsf{out}(\_)$ occurs before the first transition with a label $\_ := \mathsf{in}()$.*

Informally, a program is transductive if it terminates on all inputs (having read all symbols from the input stream) and never produces an output before reading the first input symbol.

We refer to a state $s_i$ as *in-state* when the next transition from $s_i$ reads an input symbol, i.e., $s_i \overset{\_:=\mathsf{in}()}{\rightsquigarrow} s_{i+1}$. Let $\tau^{\mathsf{in}}_{\mathscr{P}}(\vec{a}) \triangleq \tilde{s}_1 \ldots \tilde{s}_n$ be a sequence of *in*-states from $\tau_{\mathscr{P}}(\vec{a})$. For $1 \leq i < n$, let $p_i$ be the conjunction of predicates $b$ of the form $\mathsf{assume}(b)$ encountered on the transitions between $\tilde{s}_i$ and $\tilde{s}_{i+1}$, and let $p_n$ be such a conjunction of predicates between $\tilde{s}_n$ and $s_f$. In case no such transitions exist between the two states we set $p_i$ to $\mathsf{true}$. Furthermore, for $1 \leq i < n$, let $\mathbf{t}_i$ be the sequence of symbolic outputs (as written to *out*-variables by $\mathsf{out}(\_)$ transitions) between $\tilde{s}_i$ and $\tilde{s}_{i+1}$, and let $\mathbf{t}_n$ be such sequence of outputs between $\tilde{s}_n$ and $s_f$. If no output is generated we set $\mathbf{t}_i$ to $\varepsilon$. We define *big-step transition* as a transition between two successive *in*-states and write $\tilde{s}_i \overset{p_i/\mathbf{t}_i}{\rightsquigarrow}_{\vec{a}} \tilde{s}_{i+1}$.

We define *path predicates* of $\mathscr{P}$ on input $\vec{a}$ as the sequence $\pi_{\mathscr{P}}(\vec{a}) \triangleq p_1 \ldots p_n$. Analogously, we define the *symbolic output* of $\mathscr{P}$ on $\vec{a}$ by $\theta_{\mathscr{P}}(\vec{a}) \triangleq \mathbf{t}_1 \ldots \mathbf{t}_n$. The *concrete output* of $\mathscr{P}$ on $\vec{a}$ is defined by $\gamma_{\mathscr{P}}(\vec{a}) \triangleq \theta_{\mathscr{P}}(\vec{a})[\vec{in} \mapsto \vec{a}]$.

## 6. Symbolic Transducers With Look-back

In the last section, we introduced transductive programs and formalized their semantics. In this section, we introduce a transducer representation of transductive programs. We parameterize the introduced transducers by $k$, which limits the set of input variables that can be used in computation of path predicates and symbolic output. The $k$ factor can be seen as a size of the sliding window, from which a transducer can read inputs. Given an unbounded window, the symbolic transducers we introduce are equivalent to transductive programs from the last section. In our presentation, the windows begin at the current input tape head position and cover $k$ last symbols, because we observed that such transducers are most useful for the examples we encountered in practice. However, there is no fundamental reason why such windows could not also cover symbols ahead of the current head position.

### 6.1 Definitions

We now formally define symbolic $\varepsilon$-input-free finite-state transducers with $k$-lookback — $\overleftarrow{k}$-SLTs. We specialize our exposition for the case when the alphabet of input and output symbols is $\mathbb{Z}$, but generalization is easy. We omit $k$, when it is not important.

Instead of reading a single symbol from the input tape at a time, the tape head of a $\overleftarrow{k}$-SLT is effectively a window of size $k+1$, reading the current and the last $k$ symbols. Equivalently, such a transducer could be seen as a transducer with $k$ registers updated in a FIFO manner on each transition — the newly read symbol is inserted, while the oldest is removed from the queue. Rather than using registers in the further exposition, we use a set of input variables $\mathsf{V}^k_{\mathsf{T}} := \{\lambda_0, \lambda_{-1}, \ldots, \lambda_{-k}\}$, where $\lambda_0$ is the current symbol and $\lambda_{-i}$ is the symbol $i$ positions back. Abstracting away the registers simplifies the exposition and the proofs.

**Definition 2.** *A symbolic finite transducer with lookback k ($\overleftarrow{k}$-SLT) is a tuple $\mathscr{A} = (Q, q_0, \Delta)$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq Q \times BExp[\mathsf{V}^k_{\mathsf{T}}] \times Q \times DExp[\mathsf{V}^k_{\mathsf{T}}]^*$ is a finite transition relation.*

Informally, SLT is a variant of a symbolic sequential $\varepsilon$-input-free (i.e., real-time) transducer having only final states, and in general can be non-deterministic and does not have to be bounded-valued. Next, we define deterministic SLTs.

**Definition 3.** *We say that SLT $\mathscr{A}$ is* deterministic *if for every two transitions $q \xrightarrow{\varphi/\mathbf{t}} r$ and $q \xrightarrow{\varphi'/\mathbf{t}'} r'$ if $\varphi \wedge \varphi'$ is satisfiable then $r = r'$ and $(\varphi \wedge \varphi') \Rightarrow \mathbf{t} = \mathbf{t}'$ is valid. We say that SLT $\mathscr{A}$ is* transition-complete *if for every $q$ the disjunction of guards from $q$ is valid.*

SLTs that that are inferred by $\Sigma^*$ are deterministic and transition-complete, and even more, transitions from every state have mutually disjoint guards.

Before defining a run, we introduce some convenience notation. For brevity, we refer to the sequence $\lambda_{-k} \dots \lambda_0$ as $\overleftarrow{\lambda}$. To process the input, $\overleftarrow{k}$-SLT prepends it with $k$ dummy symbols $\perp \notin \mathbb{Z}$. Any operation with $\perp$ yields $\perp$ and every comparison with $\perp$ (except $\perp = \perp$) is false. For a sequence $\vec{a}$, let us denote $\vec{a}^{\perp} \triangleq \perp^k \cdot \vec{a}$. The run of a $\overleftarrow{k}$-SLT $\mathscr{A}$ is defined as follows.

**Definition 4.** *A* run *of $\overleftarrow{k}$-SLT $\mathscr{A} = (Q, q_0, \Delta)$ on $\vec{a} \in \mathbb{Z}^n$ is a finite sequence $q_0 \dots q_n, q_i \in Q$ such that there exists a sequence of transitions*

$$q_0 \xrightarrow{\varphi_1/\mathbf{t}_1} q_1 \xrightarrow{\varphi_2/\mathbf{t}_2} q_2 \ \cdots \ q_{n-1} \xrightarrow{\varphi_n/\mathbf{t}_n} q_n,$$

*where $\varphi_1, \dots, \varphi_n \in BExp[\mathsf{V}_\mathsf{T}^k]$ and $\mathbf{t}_1, \dots, \mathbf{t}_n \in Exp[\mathsf{V}_\mathsf{T}^k]^*$ such that for all $1 \le i \le n$, $\vec{a}^{\perp}|_{[i,i+k]}$ satisfies $\varphi_i$. We say that $\mathscr{A}$ on the input $\vec{a}$* produces the output *$\vec{\mathbf{o}} \in (\mathbb{Z}^*)^*$ and write $\vec{a} \twoheadrightarrow_{\mathscr{A}} \vec{\mathbf{o}}$ if for all $1 \le i \le n$ $\mathbf{o}_i = \mathbf{t}_i \left[\overleftarrow{\lambda} \mapsto \vec{a}^{\perp}|_{[i,i+k]}\right]$.*

If $\mathscr{A}$ is deterministic, the run is uniquely determined by the input sequence. For a deterministic $\mathscr{A}$ and $\vec{a} \in \mathbb{Z}^*$ let us denote by $\pi_{\mathscr{A}}(\vec{a})$, $\theta_{\mathscr{A}}(\vec{a})$ and $\gamma_{\mathscr{A}}(\vec{a})$ the corresponding sequences $\varphi_1 \dots \varphi_n$, $\mathbf{t_1} \dots \mathbf{t_n}$, and $\mathbf{o}_1 \dots \mathbf{o}_n$, respectively.

### 6.2 Equivalence Checking

To refine deterministic learned conjectures, we need to check whether they are equivalent to possibly non-deterministic abstractions. While equivalence checking is undecidable even for simple concrete non-deterministic $\varepsilon$-free generalized sequential machines [30], the equivalence of non-deterministic and deterministic sequential machines can be checked efficiently with an algorithm from Demers et al. [18]; we present a symbolic variant of their algorithm. First, we check whether the program's abstraction $\mathscr{A}_\Phi$ is single-valued, i.e., whether $\forall \vec{a} \ . \ |\mathscr{A}_\Phi(\vec{a})| = 1$. If not, it cannot be equivalent to the deterministic conjecture $\mathscr{A}_C$, because deterministic transducers are single-valued. If the abstraction is single-valued, we check whether $\mathscr{A}_\Phi = \mathscr{A}_C$, which is true iff $\mathscr{A}_\Phi \cup \mathscr{A}_C$ is single-valued. Checking whether a non-deterministic $\overleftarrow{k}$-SLT $\mathscr{A} = (Q, q_0, \Delta)$ is single-valued is efficiently decidable in $\mathscr{O}(|Q|^2)$ time [18] by checking whether a linear grammar generated from $\mathscr{A}$ generates a language of palindromes [28].

Let $(N, T, P, S)$ be a linear context-free grammar, with a finite set of non-terminals (resp. terminals) $N$ (resp. $T$), a finite set of productions $P$ of the form $N ::= TNT \mid \varepsilon$, and the start symbol $S \in N$. From a $\overleftarrow{k}$-SLT $\mathscr{A}$, generate a grammar $\mathscr{G} = (Q \times Q, BExp[\mathsf{V}_\mathsf{T}^k] \times DExp[\mathsf{V}_\mathsf{T}^k]^*, P, [q_0, q_0])$, where $P$ is defined as $[s_1, s_2] ::= (\varphi_1, \mathbf{t}_1)[s_1', s_2'](\varphi_2, \mathbf{t}_2)$, such that $(s_i, \varphi_i, \mathbf{t}_i, s_i') \in \Delta$, $\bigwedge_i \varphi_i$ is satisfiable, and $\mathbf{t}_i \neq \perp$. By $\perp$, our learning algorithm denotes outputs on transitions that are either (1) infeasible because of the constraints on the path condition, or (2) subsumed by other predicates. $\mathscr{A}$ is single-valued iff $\mathscr{G}$ generates a set of palindromes. Checking whether the outputs match under the guards reduces to checking the validity of formula $(\bigwedge_i \varphi_i) \Rightarrow \mathbf{t}_1 = \mathbf{t}_2$, which can be done with $\mathscr{O}(|Q|^2)$ calls to the prover. If $\mathscr{A}_\Phi \neq \mathscr{A}_C$, one can construct a witness called *separating sequence* by finding the shortest path from the start symbol to the first reachable rule that does not generate a palindrome.

## 7. Synthesizing Over-Approximations

$\Sigma^*$ works by iteratively learning increasingly more precise conjectures and comparing them with over-approximations of the program until the two become equivalent. We now show how to construct a sound approximation of a transductive program in the form of a non-deterministic SLT in two steps. In the first step, we construct an over-approximation of the program that is based on predicate abstraction [7, 23]. In the second step, we transform the obtained abstraction to an equivalent non-deterministic SLT. When such an SLT does not capture the behavior of the program precisely, we refine it by augmenting the set of predicates used for the predicate abstraction.

### 7.1 Abstraction of Transductive Programs

Let us consider the LTS $\mathscr{T} = (\mathsf{State}, C, \rightsquigarrow)$ of a transductive program $\mathscr{P}$ as defined in Section 5. We build our abstraction of $\mathscr{T}$ by performing predicate abstraction of the control-flow component and treating the data component explicitly. Predicate abstraction is known to be effective for control-flow dominated properties, motivating our choice of abstraction.

We parameterize our abstraction by a set of predicates $\Phi$ over variables from $\mathsf{V_E}$, interpreted over $|\mathsf{V_E}| \rightharpoonup \mathbb{Z}$. Let us denote by $\mathsf{Pred}(\Phi)$ the set of boolean combinations over predicates from $\Phi$ (i.e., all minterms). We define the abstraction of $\mathscr{T}$ as the LTS $\mathscr{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$. The set of abstract states $\mathsf{State}^\sharp$ is given by

$$\mathsf{State}^\sharp \triangleq \mathsf{L} \times \mathsf{Pred}(\Phi) \times (\mathsf{V_D} \rightharpoonup DExp[\mathsf{V_I}]) \times \mathsf{OutStr} \times \mathbb{N} \times \mathbb{N}$$

in which the valuations of internal variables are mapped to predicates in $\mathsf{Pred}(\Phi)$ satisfied by the valuation, while other components of the state are kept intact. Using the approximate post operator on $\mathsf{Pred}(\Phi)$ computed with predicate abstraction we obtain the transition relation $\rightsquigarrow^\sharp$ on abstract states. We rely on the soundness of the predicate abstraction to obtain the following.

**Proposition 5.** *For every input $\vec{a}$, if $\tau_{\mathscr{P}}(\vec{a}) = s_0 \dots s_n$ is a trace in $\mathscr{T}$ on $\vec{a}$, then there is a trace $\tau^\sharp_{\mathscr{P}}(\vec{a}) = s_0' \dots s_n'$ in $\mathscr{T}^\sharp$ such that for every $0 \le i \le n$, if $s_i = (n, \sigma|_{V_E} \cup \sigma|_{V_D}, \rho, i, j)$ and $s_i' = (n', \varphi', \sigma', \rho', i', j')$ then $n = n'$, $\rho = \rho'$, $i = i'$, $j = j'$, $\sigma|_{V_D} = \sigma'$ and $\sigma|_{V_E}$ satisfies $\varphi'$. Consequently, the output in the abstraction, $\gamma^\sharp_{\mathscr{P}}(\vec{a})$, is equal to $\gamma_{\mathscr{P}}(\vec{a})$.*

### 7.2 Translation to SLT

We now translate the abstract LTS $\mathscr{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$ into an equivalent (possibly non-deterministic) SLT. First, we focus on the penultimate *i*-component of the abstract state tuple, which represents the offset of the input tape head from the beginning of the tape. We can abstract away that offset by *relativizing* the states so that offset is relative to the current position of the input tape head, rather than the beginning of the tape. Let $\mathsf{V_T} \triangleq \{\lambda_0, \lambda_{-1}, \dots\}$ be the infinite set of all $\lambda$-variables. Let $\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}$ denote a variable substitution that maps each *in*-variable $in_j$ to $\lambda$-variable $\lambda_{j-i}$.

We define input-relativisation of a state $s = (n, \varphi, \sigma, \rho, i, j) \in \mathsf{State}^\sharp$ by $\Lambda(s) \triangleq (n, \varphi[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}], \sigma[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}], \rho[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}], j)$. Intuitively, $\Lambda(s)$ relativizes symbolic expressions in $s$ with respect to the current *in*-index (i.e., the current position of the input tape head). In general, expressions in $\sigma$- and $\rho$-components of $\Lambda(s)$ may use unboundedly many $\lambda$-variables as the expressions can refer to inputs from arbitrary far in the past. To allow translation to a SLT, we focus on a class transductive programs that, in addition to being input-output equivalent to a SLT, use only finitely many $\lambda$-variables.

**Definition 6.** *We say that a transductive program $\mathscr{P}$ is $\overleftarrow{k}$–SLT if (1) there exist a $\overleftarrow{k}$–SLT $\mathscr{A}$ such that for all $\vec{a}$, $\gamma_{\mathscr{P}}(\vec{a}) = \gamma_{\mathscr{A}}(\vec{a})$, and (2) for all $s$, $\Lambda(s)$ uses only $\lambda$-variables in $\mathsf{V}_\mathsf{T}^k$.*

A consequence of the first property in the definition is that $\overleftarrow{k}$-SLT programs will have a finite number of *in*-states and big-step transitions (Section 5.3). The second property ensures that program consumes input using a bounded sliding window. In practice, instead of postulating this property, we could replace it with additional safety checks that would enforce avoiding infeasible paths with look-back greater than $k$ when constructing the abstraction.

Now we define an equivalence relation $\sim$ on $\mathsf{State}^\sharp$ as follows. For $s, s \in \mathsf{State}^\sharp$, we let $s \sim s'$ iff for $\Lambda(s) = (n, \varphi_\lambda, \sigma_\lambda, \rho_\lambda, j)$ and $\Lambda(s') = (n', \varphi'_\lambda, \sigma'_\lambda, \rho'_\lambda, j')$ we have $n = n'$, $\varphi_\lambda \Leftrightarrow \varphi'_\lambda$ and $\sigma_\lambda = \sigma'_\lambda$.

**Lemma 7.** *If $\mathscr{P}$ is an SLT then $\sim$ is of finite index.*

Let us define $\mathsf{paths}^\sharp_{\mathsf{in}}(s,t)$ as the set of all $\rightsquigarrow^\sharp$-sequences of states between $s$ and $t$ such that there is a single input transition between states on the path from $s$ to $t$. For $\xi \in \mathsf{paths}^\sharp_{\mathsf{in}}(s,t)$, let us denote by $\pi^\sharp(\xi)$ the conjunction of assumed predicates on transitions in $\xi$ and let $\theta^\sharp(\xi)$ be the produced symbolic output. We need the following lemma for our translation to SLT to be well-defined.

**Lemma 8.** *If $s \sim s'$ and $t \sim t'$ then for every path $\xi \in \mathsf{paths}^\sharp_{\mathsf{in}}(s,t)$, there exists a unique path $\xi' \in \mathsf{paths}^\sharp_{\mathsf{in}}(s',t')$ such that $\pi^\sharp(\xi) = \pi^\sharp(\xi')$ and $\theta^\sharp(\xi) = \theta^\sharp(\xi')$.*

We define $\mathscr{A}_\Phi$ to be an SLT $(Q, q_0, \Delta)$ with

$$Q \triangleq \left\{ [s]_\sim \mid \exists s' \in \mathsf{State}^\sharp \,.\, s \overset{-:=\mathsf{in}()^\sharp}{\rightsquigarrow} s' \right\}$$

as the set of states, $q_0 \triangleq [s_0]_\sim$ as the initial state[1] and $\Delta$ as the transition relation such that $[s] \xrightarrow{\varphi/\mathbf{t}} [s'] \in \Delta$ iff there exist $\xi \in \mathsf{paths}^\sharp_{\mathsf{in}}(s,s')$, such that $\pi^\sharp(\xi) = \varphi$ and $\theta^\sharp(\xi) = \mathbf{t}$. Intuitively, $\mathscr{A}_\Phi$ represents all isomorphism classes of big-step transitions between the abstracted *in*-states.

**Lemma 9.** *If $\mathscr{P}$ is $\overleftarrow{k}$-SLT then $\mathscr{A}_\Phi$ is $\overleftarrow{k}$-SLT.*

We can now show that $\mathscr{A}_\Phi$ captures exactly the behavior of $\mathscr{T}^\sharp$ thus $\mathscr{A}_\Phi$ soundly over-approximates the behavior of $\mathscr{P}$.

**Proposition 10.** *For all $\vec{a}$, $\gamma^\sharp_{\mathscr{P}}(\vec{a}) = \vec{\mathbf{o}}$ iff $\vec{a}^\perp \twoheadrightarrow_{\mathscr{A}_\Phi} \vec{\mathbf{o}}$.*

**Corollary 11.** *For all $\vec{a}$, if $\gamma_{\mathscr{P}}(\vec{a}) = \vec{\mathbf{o}}$ then $\vec{a}^\perp \twoheadrightarrow_{\mathscr{A}_\Phi} \vec{\mathbf{o}}$.*

### 7.3 Refinement

Suppose the learned conjecture and the abstraction $\mathscr{A}_\Phi$ differ and that $\vec{a}$ is a separating sequence. If the following conditions are all true: $\gamma_{\mathscr{P}}(\vec{a}) = \vec{\mathbf{o}}$, $\vec{a} \twoheadrightarrow_{\mathscr{A}_\Phi} \vec{\mathbf{o}}'$, and $\vec{\mathbf{o}} \neq \vec{\mathbf{o}}'$, then we need to refine the abstraction $\mathscr{A}_\Phi$. We want to add enough predicates to $\Phi$ to evidence infeasibility of the spurious run in $\mathscr{A}_\Phi$ that generates $\vec{\mathbf{o}}'$.

The existence of a counterexample means that $\mathscr{A}_\Phi$ is not precise enough, i.e., that it strictly over-approximates $\mathscr{P}$. Since our abstraction is based on predicate abstraction and fully precise isomorphic representation of other components of the state, $\mathscr{A}_\Phi$ in fact defines the strongest inductive invariant $\psi$ of transition relation of $\mathscr{P}$ that is expressible as a Boolean combination of the given set of predicates, while the input-output relation is preserved exactly in the relativized form. Assuming a complete decision procedure for the underlying theory and a predicate selection method that would eventually build $\Phi$, by the relative completeness of predicate abstraction [8, 31], we could generate an invariant as strong as $\psi$.

[1] Without loss of generality, we can assume that the initial state is an *in*-state, because $\mathscr{P}$ is transductive (Definition 1) and therefore cannot produce output before reading some input. Thus, the states before the first *in*-state are uninteresting, and can be merged into the first *in*-state.

It is not clear whether $\psi$ can always be constructed, and if it can, is it independent of the number of states (or some other intrinsic property) of the $\overleftarrow{k}$-SLT $\mathscr{A}_{\mathscr{P}}$ that is behaviorally equivalent to $\mathscr{P}$. It is, on the other hand, possible to construct such an invariant if the number of states $n$ is known a priori, by explicitly encoding a checking sequence [32] that distinguishes $\mathscr{A}_{\mathscr{P}}$ from all other transducers up to $n$ states. Therefore, to abstract away the complexity of such construction, we assume existence of a predicate selection oracle that eventually yields a set $\Phi$ resulting in an abstraction $\mathscr{A}_\Phi$ equivalent to $\mathscr{P}$. We say that a predicate selection oracle is *complete* if it is guaranteed to eventually generate a sufficient set of predicates to construct $\mathscr{A}_\Phi$ equivalent to $\mathscr{P}$. Our main result expresses completeness of our learning algorithm relative to existence of such a complete predicate selection method.

Further on, we assume a suitable refine procedure that given a set of predicates $\Phi$ and an input $\vec{a}$ producing a spurious run in $\mathscr{A}_\Phi$ returns an augmented set $\Phi'$, for which the spurious run is avoided in $\mathscr{A}_{\Phi'}$. Although generally it may be hard if not impossible to construct a predicate selection method that would always yield the right predicates for constructing $\mathscr{A}_\Phi$, our empirical evaluation shows that standard heuristics work well in practice for the examples we analyzed.

## 8. Learning

In this section, we describe the $\Sigma^*$ algorithm. The algorithm constructs a table, called *observation table*, similarly as $L^*$, but table entries are path predicates and symbolic output (see Section 5.3), rather than concrete sequences. The finished table can be easily translated into a $\overleftarrow{k}$-SLT, representing a conjecture. The conjecture is always deterministic. By checking equivalence of the conjecture and a potentially non-deterministic $\overleftarrow{k}$-SLT abstraction of program $\mathscr{P}$ (Section 6.2), we either generate a concrete counterexample showing how the conjecture and abstraction differ, or prove they are equivalent. If the counterexample is spurious (the program does not produce the same sequence of outputs as the abstraction), we refine the abstraction. Otherwise, the counterexample represents a behavior that the conjecture failed to capture, and we refine the conjecture. We proceed by describing some notational conveniences used in this section, followed by the detailed presentation of the algorithm and its properties.

For brevity, all sets of data and boolean expressions are over $\overleftarrow{k}$-SLT input variables $\mathsf{V}^k_{\mathsf{T}}$, defined in Section 6.1. Let $\mathsf{V}_{\mathsf{T}}$ be the infinite set of all $\lambda$-variables, defined in Section 7.2. We define a sequence-relativisation function $\Lambda(s_1 \ldots s_n) \triangleq s_1[\vec{in} \overset{1}{\mapsto} \overleftarrow{\lambda}] \ldots s_n[\vec{in} \overset{n}{\mapsto} \overleftarrow{\lambda}]$, where $\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}$ is the variable substitution defined in Section 7.2 and $\vec{s}$ is either a path predicate or symbolic output. If $\vec{s}$ is a symbolic output $\vec{\mathbf{t}}$, the same relativisation function is applied to each individual term in the subsequence, i.e., if $\mathbf{t} = t_1 \ldots t_m$ then $\mathbf{t}[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] = t_1[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] \ldots t_m[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}]$. We define SOLVE as a function that takes a sequence of predicates, derelativizes them by applying the inverse of the $\Lambda$ substitution, computes a conjunction of derelativized predicates $\bigwedge_{1 \leq i \leq |\vec{p}|} \left(\Lambda^{-1}(\vec{p})\right)|_i$, passes the conjunction to a solver, and returns a concrete sequence $\mathbb{Z}^*$ of input symbols $\vec{a}$ satisfying the conjunction such that $|\vec{a}| = |\vec{p}|$, or $\perp$ if the conjunction is infeasible. Finally, we point out that all equality (resp. inequality) checks $=$ (resp. $\neq$) over predicates and terms in this section are syntactic equality (resp. inequality) checks.[2]

[2] At the cost of more complex exposition, we could use semantic equality and check that output terms are equal under the guard restrictions. Such an approach might allow us to learn more compact $\overleftarrow{k}$-SLTs.

## 8.1 Definitions

$\Sigma^*$ constructs a symbolic observation table, defined as follows:

**Definition 12.** Symbolic observation table *is a quadruple* $(R, S, E, T) \subset (BExp^*, BExp^*, BExp^*, BExp^* \times BExp^* \to (DExp \cup \{\perp, \varepsilon\})^*)$, *where* $R \subseteq S$ *represents a set of identified states,* $S$ *(resp.* $E$*) is a prefix- (resp. suffix-) closed set of relativized path predicates,* $T$ *is a table indexed by* $\vec{p}_{\mathsf{p}} \in S, \vec{p}_{\mathsf{s}} \in E$ *containing a relativized symbolic output. The entry at* $T[\vec{p}_{\mathsf{p}}, \vec{p}_{\mathsf{s}}]$ *is only the suffix of symbolic output generated by* $\mathscr{P}$ *when processing* $\vec{p}_{\mathsf{s}}$*, or more formally:* $\forall \vec{a} = \textsc{Solve}(\vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}})\ .\ \vec{\mathbf{t}} = \theta_{\mathscr{P}}(\vec{a}) \wedge \vec{\mathbf{t}} = \vec{\mathbf{t}}_p \cdot \vec{\mathbf{t}}_s \wedge |\vec{\mathbf{t}}_s| = |\vec{p}_{\mathsf{s}}| \wedge T[\vec{p}_{\mathsf{p}}, \vec{p}_{\mathsf{s}}] = \vec{\mathbf{t}}_s$*. For some* $\vec{p} \in S$*, we define a* $\vec{p}$-row *in the observation table as an* $E$*-indexed set, denoted* $\vec{p}$*-row. We denote outputs generated by infeasible and redundant transitions in the table by* $\perp$*.*

Intuitively, set $R$ represents a set of shortest paths leading to discovered states, set $S \supseteq R$ contains exactly path predicates from $R$ and additionally all the sequences that extend sequences from $R$ by exactly one big-step transition. The role of $S$ is to exercise all the transitions in the inferred transducer. Finally, set $E$ is the set of distinguishing tests that distinguish different states.

The standard $L^*$ makes a conjecture when the table is *closed*, which means that every sequence in $S$ has a representative in $R$, or more formally $\forall \vec{p} \in S\ .\ \exists \vec{r} \in R\ .\ \vec{p}\text{-}row = \vec{r}\text{-}row$. We define closedness in the same way as $L^*$. From a closed table, one can construct a complete (for all states and input symbols, all transitions are defined) transducer using standard techniques (e.g., see [4, 36]).[3]

## 8.2 Algorithm

We begin by describing the FillRows algorithm that computes the undefined entries of the table, continue with the ExtendTable algorithm that explores the successor states of all states discovered at certain step, and end with the $\Sigma^*$ algorithm.

Algorithm 1 computes the missing entries in the table. If the entry is missing for some prefix $\vec{p}_{\mathsf{p}} \in S$ and suffix $\vec{p}_{\mathsf{s}} \in E$, we first try to compute a concrete witness $\vec{a}$ by splicing together the prefix and the suffix. While the $S$ (and $R$) sets contain path predicates that are collected along prefixes of some feasible paths in $\mathscr{P}$ starting from the initial state, the $E$ set contains suffixes of feasible paths. Naturally, when we arbitrarily splice prefixes and suffixes of different paths, the resulting formula might be infeasible. If feasible, we execute $\vec{a}$ on $\mathscr{P}$ using concolic execution [13, 21, 35] (Line 4) and collect the predicates ($\vec{r}$) and output terms ($\vec{\mathbf{t}}$) from big-step transitions. Note that the collected predicates might differ from $\vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}}$, but at least the prefix (corresponding to $\vec{p}_{\mathsf{p}}$) will always match. The outputs corresponding to mismatched predicates and infeasible path conditions are marked $\perp$. Lines 6–11 replace the output terms at positions where the $\vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}}$ and $\vec{r}$ sequences differ syntactically. Finally, lines 16–17 close the table.

Algorithm 2 takes a state representative $\vec{r}$, i.e., a path predicate that holds on the shortest path to the identified state, and finds all the outgoing big-step transitions from that state, adding the predicates from those transitions to $E$ and the entire sequence ($\vec{r}$ extended by one transition) to $S$. The only interesting part of the algorithm is the discovery of new transitions and the corresponding predicates. In the first iteration, Line 6 extends the representative sequence $\vec{r}$ with predicate `true`, effectively allowing the solver to produce an arbitrary value for the last element of the concrete input sequence.

[3] In the $L^*$ setting, one also defines the *consistency* property, which roughly says that if two sequences $\vec{p}_1, \vec{p}_2$ from $R$ are equivalent, then both states reached by $\gamma_{\mathscr{P}}(\textsc{Solve}(\vec{p}_1))$ and $\gamma_{\mathscr{P}}(\textsc{Solve}(\vec{p}_2))$ must produce the same output in the next big-step transition given the same input symbol. We maintain the consistency of our symbolic observation table by always assuring that each state has only one representative in the $R$ set.

```
input and output : Observation table OT
1  forall the $\vec{p}_{\mathsf{p}} \in S, \vec{p}_{\mathsf{s}} \in E$ such that $T[\vec{p}_{\mathsf{p}}, \vec{p}_{\mathsf{s}}]$ is undefined do
2      $\vec{a} := \textsc{Solve}(\vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}})$
3      if $\vec{a} \neq \perp$ then
4          $(\vec{r}, \vec{\mathbf{t}}) := (\Lambda(\pi_{\mathscr{P}}(\vec{a})), \Lambda(\theta_{\mathscr{P}}(\vec{a})))$
           // assert($|\vec{r}| = |\vec{\mathbf{t}}|$)
5          $\vec{\mathbf{t}}_s := \varepsilon$
6          forall the $1 \leq i \leq |\vec{\mathbf{t}}|$ do
7              if $i > |\vec{p}_{\mathsf{p}}|$ then
8                  if $(\vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}})|_i = \vec{r}|_i$ then
9                      $\vec{\mathbf{t}}_s := \vec{\mathbf{t}}_s \cdot \vec{\mathbf{t}}|_i$
10                 else
11                     $\vec{\mathbf{t}}_s := \vec{\mathbf{t}}_s \cdot \perp$
12             else
                   // assert($\vec{p}_{\mathsf{p}}|_i = \vec{r}|_i$)
13         $T[\vec{p}_{\mathsf{p}}, \vec{p}_{\mathsf{s}}] := \vec{\mathbf{t}}_s$
14     else
15         $T[\vec{p}_{\mathsf{p}}, \vec{p}_{\mathsf{s}}] := \perp$
   // Now close the table
16 forall the $\vec{p} \in S$ s.t. $\neg\exists \vec{r} \in R\ .\ \vec{p}\text{-}row = \vec{r}\text{-}row$ do
17     $R := R \cup \vec{p}$                                   // New state
18 Return OT
```

**Algorithm 1:** The FillRows Algorithm.

Executing the obtained concrete sequence and collecting predicates along the path, we identify the first big-step transition guard predicate ($r_s$). In every following iteration, we negate a disjunction of the predicates discovered so far, until the disjunction becomes valid (test at Line 4). All infeasible traces lead to a ghost state that has one self-loop transition labeled `true`$/\perp$. The algorithm creates such a state automatically, if needed, by filling the corresponding row with $\perp$, as even the prefix to the ghost state is infeasible.

```
input and output : Observation table OT
1  forall the $\vec{r} \in R$ do           // Extend all sequences from R
2      if $\neg\exists \vec{p}_{\mathsf{s}} \in E\ .\ |\vec{p}_{\mathsf{s}}| = 1 \wedge \vec{r} \cdot \vec{p}_{\mathsf{s}} \in S$ then
3          $s :=$ false
4          while $s \not\Leftrightarrow$ true do
5              $r_s := \varepsilon$
6              $\vec{a} := \textsc{Solve}(\vec{r} \cdot \neg s)$
7              if $a = \perp$ then
8                  $r_s := \neg s$
9                  $s :=$ true
10             else
11                 $\vec{p} := \Lambda(\pi_{\mathscr{P}}(\vec{a}))$
                   // assert($|\vec{p}| = |\vec{r}| + 1$)
12                 $r_s := \vec{p}|_{|\vec{r}|+1}$
13                 $s := s \vee r_s$
14             $E := E \cup \{r_s\}$
15             $S := S \cup \{\vec{r} \cdot r_s\}$
16             $OT :=$ FillRows($OT$)
17 Return OT
```

**Algorithm 2:** The ExtendTable Algorithm.

Finally, Algorithm 3 infers a symbolic transducer. Lines 1–9 discover all the big-step transitions from the initial state and the corresponding predicates. All the discovered predicates are added to the $E$ set. In the next three lines, we extend and close the table, producing the first $\mathscr{A}_C$ conjecture and the first abstraction $\mathscr{A}_\Phi$ of $\mathscr{P}$. The loop beginning on Line 14 checks the equivalence between the conjecture and abstraction. If they are equivalent, the algorithm terminates returning the exact transducer implemented by $\mathscr{P}$. Otherwise, the counterexample is checked against $\mathscr{P}$. If it is spurious, we refine the abstraction, otherwise, we refine the conjecture. We use Shahbaz and Groz's [36] technique for processing the counterexamples adapted for our symbolic setting. First, we collect the predicates from $\mathscr{P}$ along the path determined by the counterexample and discard the longest prefix that is already in $S$. We denote the remaining suffix by $\vec{p}_\mathsf{s}$. We add all suffixes of $\vec{p}_\mathsf{s}$ to $E$ (Line 24), to assure that $E$ remains suffix closed.

```
init   : $R = \{\varepsilon\}, S = \{\varepsilon\}, E = \emptyset, T = \emptyset, i = 0, s = \mathsf{false}$
result : $\overleftarrow{k}$-SLT $\mathscr{A}_C$
repeat // Fill 1st row
    if $s \Leftrightarrow \mathsf{false}$ then
        $a :=$ randomly generated array of type $\mathbb{Z}[1]$
    else
        $a :=$ SOLVE$(\neg s)$
    $p := \Lambda((\pi_{\mathscr{P}}(a))|_1)$ // 1st pred. from path predicate
    $E := E \cup p$
    $T[\varepsilon, p] := \Lambda((\theta_{\mathscr{P}}(a))_1)$
    $s := s \vee p$
until $s \Leftrightarrow \mathsf{true}$
$(R,S,E,T) :=$ EXTENDTABLE(FILLROWS$(R,S,E,T)$)
Compute $\mathscr{A}_C$ from $(R,S,E,T)$
Compute initial $\mathscr{A}_\Phi$ of $\mathscr{P}$
while true do
    Let $(\vec{a}, \vec{\mathbf{o}})$ be a separating sequence between $\mathscr{A}_C$ and $\mathscr{A}_\Phi$
    if $\vec{a} = \varepsilon$ then
        Return $\mathscr{A}_C$
    if $\gamma_{\mathscr{P}}(\vec{a}) \neq \vec{\mathbf{o}}$ then // Spurious counterexample?
        Refine $\mathscr{A}_\Phi$ of $\mathscr{P}$ on $(\vec{a}, \vec{\mathbf{o}})$
    else
        $\vec{p} := \Lambda(\pi_{\mathscr{P}}(\vec{a}))$
        Let $\vec{p}_\mathsf{p}$ be the longest prefix of $\vec{p}$ s.t. $\vec{p}_\mathsf{p} \in S$
        Let $\vec{p}_\mathsf{s}$ be s.t. $\vec{p} = \vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s}$
        $E := E \cup Suffix(\vec{p}_\mathsf{s})$
        $(R,S,E,T) :=$ EXTENDTABLE(FILLROWS$(R,S,E,T)$)
        Compute $\mathscr{A}_C$ from $(R,S,E,T)$
```

**Algorithm 3:** The $\Sigma^*$ Algorithm.

### 8.3 Properties

First, we state the main properties of $\Sigma^*$, and then proceed with the proof of relative completeness and a discussion of computational complexity.

**Lemma 13.** *Let $T = (R,S,E,T)$ be a symbolic observation table. $\Sigma^*$ preserves the following invariants:*

1. *R and S (resp. E) are always prefix- (resp. suffix-) closed.*
2. *For every $\vec{p} \in S$ there is a unique $\vec{r} \in R$ such that $\vec{p}$-row $= \vec{r}$-row.*
3. *For every $\vec{r} \in R$, there are $r_\mathsf{s}^1, \ldots, r_\mathsf{s}^n$ such that $\bigvee_{i=1}^n r_\mathsf{s}^i \Leftrightarrow \mathsf{true}$ and for all i, it holds that $\vec{r} \cdot r_\mathsf{s}^i \in S$.*
4. *The conjecture $\mathscr{A}_C$ is closed at the end of each step.*

***Correctness.*** $R$ represents the part of the all-path symbolic execution tree of $\mathscr{P}$ on which the conjecture faithfully represents the behaviour of $\mathscr{P}$, which is stated with the following proposition.

**Proposition 14** (Bounded correctness)**.** *After each step, for all $\vec{r} \in R$ and $\vec{a}$ such that $\vec{a} =$ SOLVE$(\vec{r})$, $\gamma_{\mathscr{P}}(\vec{a}) = \gamma_{\mathscr{A}_C}(\vec{a})$ holds.*

***Completeness.*** The following lemma ensures that a progress is made after each conjecture refinement.

**Lemma 15.** *If at some step of $\Sigma^*$, $(\vec{a}, \vec{\mathbf{o}})$ is a separating sequence such that $\gamma_{\mathscr{P}}(\vec{a}) = \vec{\mathbf{o}}$, then at the end of the step, for all $\vec{a}'$ such that $\vec{a}' =$ SOLVE$(\pi_{\mathscr{P}}(\vec{a}))$, $\gamma_{\mathscr{A}_C}(\vec{a}') = \gamma_{\mathscr{P}}(\vec{a}')$ holds.*

We state our completeness result relative to the completeness of the predicate selection oracle.

**Theorem 16** (Relative completeness)**.** *If $\mathscr{P}$ is an SLT and the predicate selection method for refinement is complete, then $\Sigma^*$ terminates with $\mathscr{A}_C$ being behaviorally equivalent to $\mathscr{P}$.*

*Proof.* First note that when $\mathscr{A}_\Phi$ is single-valued, then by the soundness of abstraction, $\mathscr{A}_\Phi$ is in fact behaviorally equivalent to $\mathscr{P}$. As the predicate selection method is assumed to be complete, we will eventually obtain a single-valued $\mathscr{A}_\Phi$.

The equivalence check of $\mathscr{A}_\Phi$ and $\mathscr{A}_C$ always returns the shortest separating sequence (if one exists). There can be only finitely many shortest separating sequences of a given length, and at each step such a sequence is used either to refine the conjecture or to refine the abstraction. Therefore, the number of conjecture refinements must also be finite. $\square$

***Computational complexity.*** Next, we analyze the complexity of $\Sigma^*$. Let $n$ be the number of states of the inferred $\overleftarrow{k}$-SLT, $k$ the maximal number of outgoing big-step transitions from any state, $m$ the maximal length of any counterexample, and $c$ the number of counterexamples. There can be at most $n-1$ counterexamples, as each counterexample distinguishes at least one state. Since we initialize $E$ with $k$ predicates, $|E|$ can grow to at most $k + m(n-1)$. The size of $S$ is at most $n \cdot k$. Thus, the table can contain at most $n \cdot k \cdot (k + m \cdot n - m) = \mathscr{O}(n \cdot k^2 + m \cdot n^2 k)$ entries. The $k$ factor is likely to be small in practice, and our equivalence checking algorithm finds the minimal counterexample. The solver is called once per each state (i.e., representative in $R$) and for each outgoing transition. For each equivalence check, we might need to call the solver $n^2$ times. Thus, the total worst-case number of calls to the solver is $\mathscr{O}(n \cdot k + n^2)$, not including the number of calls required for abstraction refinement, which depends on the abstraction technique used.

## 9. Experimental Evaluation

We have implemented the learning part of $\Sigma^*$ (algorithms in Section 8) and evaluated it on the Google AutoEscape (GA) framework sanitizers, the same examples used recently by Hooimeijer et al. [27]. We instrumented them with a single command to denote the line corresponding to the *in*-state. We dropped the ValidateUrl sanitizer, as it is effectively just a wrapper for calling other sanitizers. Programs `EncodeHtml` and `GetTags` have been obtained from the corresponding C# examples with the same name from [27] and [12], respectively.

We patched Klee [14] with around 300 lines of code to generate symbolic relativized traces, wrote an implementation of Algorithms 1–3. $\Sigma^*$ learned 10 out of 14 examples, and the first conjecture was correct on all but one benchmark (GetTags required two conjectures). To check the conjectures, we manually constructed predicate abstractions of the examples using syntactic predicates from the code. Such a simple heuristic was sufficient to construct a complete predicate abstraction of programs' control flow. The predicate abstraction checking took us on the order of 30 minutes per example, but it is well known it could be automated.

We ran the experiments under Cygwin on Windows 7 64bit, running on Intel 2.8GHz Core Duo CPU with 4GB RAM. The path exploration with Klee took under 10 minutes on all examples, and the

| Benchmark | Learned | Control Bits | Data Bits |
|---|---|---|---|
| CleanseCss | $+$ | 32 | 8 |
| CleanseAttribute | $-(*)$ | 32 | 8 |
| CssUrlEscape | $+$ | 32 | 8 |
| HtmlEscape | $+$ | 0 | 8 |
| JsonEscape | $+$ | 0 | 8 |
| PreEscape | $+$ | 0 | 8 |
| UrlQueryEscape | $+$ | 0 | 8 |
| XMLEscape | $+$ | 0 | 8 |
| JavascriptEscape | $+$ | 0 | 16 |
| JavascriptNumber | $-(*)$ | 33 | 8 |
| PrefixLine | $-$ | 0 | 8 |
| SnippetEscape | $-(*)$ | 0 | 8 |
| EncodeHtml | $+$ | 0 | 8 |
| GetTags | $+$ | 32 | 16 |

Table 1: Experimental Results. The benchmarks that $\Sigma^*$ successfully learned are denoted by $+$. Benchmarks marked with $(*)$ could be learned if $\Sigma^*$ were extended to handle subsequential transducers. The last two columns show the size of the internal control and data state. We counted $\lambda_0$ as 8 bits of data state. We counted boolean variables as a single bit (e.g., for the JavascriptNumber benchmark).

learning itself took under a second. In comparison, human expert required several hours of analysis to extract symbolic transducers manually [27].

The learned $\overleftarrow{k}$-SLTs all have the lookback of zero or one, and between one and three states, which attests to the compactness of $\overleftarrow{k}$-SLTs as a symbolic representation. Concrete versions of these sanitizers are far less compact, as the alphabets are either 8- or 16-bit, and some benchmarks have internal state of up to 48 bits (e.g., GetTags).

PrefixLine is effectively a $\overleftarrow{0}$-SLT, but it is implemented using the `memchr` function, and the results of these function calls are used in guards. The remaining three sanitizers, CleanseAttribute, JavascriptNumber and SnippetEscape, are subsequential: CleanseAttribute detects the end of the input and handles the end differently, JavascriptNumber checks validity of the input stream before deciding the output, and SnippetEscape outputs unclosed tags from a predefined set at the end of the input. These could be learned if $\Sigma^*$ were extended to subsequential transducers by generalizing Vilar's algorithm [39] to the symbolic setting.

Although small in numbers of lines of code, sanitizers are often very difficult to implement correctly. Being able to automatically infer a formal model of these functions, allows us to automatically check their properties such as idempotence, commutativity and reversibility [27] that are important for establishing security of web applications.

## 10. Limitations and Future Work

In this section, we discuss the main theoretical and practical limitations of $\Sigma^*$. The main theoretical limitation is the relative completeness to the abstraction method chosen for over-approximating the program. While the existing predicate selection heuristics worked well for our examples, and were indeed capable of constructing the strongest invariant of program's transition relation, more research is needed in this direction. We hope that $\Sigma^*$ could help elucidating the connection between predicate selection heuristics and completeness.

On the practical side, the specialized version of $\Sigma^*$ developed in this paper was expressive enough to infer symbolic transducers on a number of real-world examples, but $\overleftarrow{k}$-SLTs are not expressive enough to represent functions like `fromLast` and `uptoLast` [27], which inherently require non-determinism. Learning such more expressive classes of symbolic transducers remains an open problem.

## References

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *ICTSS'10: Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, volume 6435 of *LNCS*, pages 188–204. Springer, 2010.

[2] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 599–610. ACM, 2011.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109. ACM, 2005.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08: Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272. ACM, 2008.

[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[7] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS'01: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 268–283, 2001.

[8] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS 2002: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172, 2002.

[9] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 3–14, 2008.

[10] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *FASE'08: Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, volume 4961 of *LNCS*, pages 317–331. Springer, 2008.

[11] N. Bjørner and M. Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, 2011.

[12] N. Bjørner, P. Hooimeijer, B. Livshits, D. Molnar, and M. Veanes. Symbolic finite state transducers: Algorithms and applications. In *POPL'12: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2012.

[13] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05: Proc. of the 12th Int. SPIN Workshop on Model Checking Software*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08: Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[15] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for

protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Security Symposium*, Aug 2011.

[16] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *TACAS 2005: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[17] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS'03: Proc. of the 9th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 331–346. Springer Berlin, 2003.

[18] A. J. Demers, C. Keleman, and B. Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.

[19] E. Elkind, B. Genest, D. Peled, and H. Qu. Grey-box checking. In *FORTE'06: Proceedings of the 2006 Formal Techniques for Networked and Distributed Systems*, volume 4229 of *LNCS*, pages 420–435. Springer, 2006.

[20] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202. ACM, 2002.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI'05: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[22] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010.

[23] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97: Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[24] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE'06: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 117–127. ACM, 2006.

[25] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138:21–36, 2005.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70. ACM, 2002.

[27] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Security Symposium*, Aug 2011.

[28] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Theory of Computing Systems*, 3:119–124, 1969.

[29] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI'12: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012. To appear.

[30] O. H. Ibarra. The unsolvability of the equivalence problem for $\varepsilon$-free NGSM's with unary input (output) alphabet and applications. In *SFCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 74–81. IEEE Computer Society, 1977.

[31] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS 2006: Proceedings of the 12th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 459–473, 2006.

[32] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. In *Proc. of the IEEE*, volume 84, pages 1090–1123. IEEE Computer Society, 1996.

[33] K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06: Proceedings of 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[34] J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 15:448–458, May 1993.

[35] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30:263–272, 2005.

[36] M. Shahbaz and R. Groz. Inferring Mealy machines. In *FM'09: Proc. of the 2nd World Congress on Formal Methods*, pages 207–222. Springer, 2009.

[37] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234. ACM, 2009.

[38] G. van Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

[39] J. M. Vilar. Query learning of subsequential transducers. In *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 72–83. Springer-Verlag, 1996.