Sigma*: Symbolic Learning of Input-Output Specifications

Matko Botinčan
University of Cambridge
matko.botincan@cl.cam.ac.uk

Domagoj Babić *
Facebook, Inc.
babic@eecs.berkeley.edu

Abstract

We present Sigma*, a novel technique for learning symbolic models of software behavior. Sigma* addresses the challenge of synthesizing models of software by using symbolic conjectures and abstraction. By combining dynamic symbolic execution to discover symbolic input-output steps of the programs and counterexample guided abstraction refinement to over-approximate program behavior, Sigma* transforms arbitrary source representation of programs into faithful input-output models. We define a class of stream filters—programs that process streams of data items—for which Sigma* converges to a complete model if abstraction refinement eventually builds up a sufficiently strong abstraction. In other words, Sigma* is complete relative to abstraction. To represent inferred symbolic models, we use a variant of symbolic transducers that can be effectively composed and equivalence checked. Thus, Sigma* enables fully automatic analysis of behavioral properties such as commutativity, reversibility and idempotence, which is useful for web sanitizer verification and stream programs compiler optimizations, as we show experimentally. We also show how models inferred by Sigma* can boost performance of stream programs by parallelized code generation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software Verification—Formal methods; D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Languages, Theory, Verification

Keywords Inductive learning, Specification synthesis, Behavioral properties, Equivalence checking, Stream programs, Compiler optimization, Parallelization

1. Introduction

Modern software systems often process large amounts of data in a stream-like fashion. By streams, we mean sequences of data items, processed in some order. Such stream processing is inherent to many domains such as digital signal processing, embedded applications, network event processing, financial applications, web applications, and even to common desktop software. Likewise, big-data services running on a cloud often continuously process streams of moving data and batches of past data.

We focus on a simple notion of programs that applies to many of these systems, called a *filter*. Filters are typically small fragments of code, but they facilitate large scale stream-processing [15, 18, 23, 26, 29, 32, 54]. A filter iteratively reads a bounded number of items from an input stream at once, performs some computation on those items, and writes the results to an output stream. Filters exchange data via stream(s) with other parts of the program, and can also be glued together into more complex structures. Unfortunately, reasoning about even these relatively simplistic programs is difficult as their low-level implementation details often obscure the high-level input-output behavior that we want to reason about.

Our aim is to provide ways for automated analysis of the input-output behavior of programs such as stream filters. Although the internal implementation of such programs may be rather delicate, many of their properties rely solely on their input-output behavior. To exemplify a couple of such properties, let us consider intentionally simple filters P and P' (coming from, say, two revisions of the code or from two different vendors), that implement a difference encoder (used in, e.g., JPEG compression [53]):[1]

```
   | out(peek(0));                   | state = 0;
   | while (true) {                  | while (true) {
P  |   out(peek(1)−peek(0));     P'  |   out(peek(0)−state);
   |   in();                         |   state = in();
   | }                               | }
```

P and P' are syntactically different and have different operational semantics but they exhibit the same input-output behavior. Behavioral equivalence of P and P' ensures that the two implementations can be used interchangeably—for instance, a behaviorally equivalent stateless (or with few states) implementation might be preferable over a stateful (or with many states) one, as the former can be made amenable to data-parallel execution by a simple replication. For another property, consider a filter Q that looks like P but instead of the difference calculates the sum of the two consecutive values. If we connect P's output stream to Q's input stream in a pipeline then the resulting composition will have the same behavior as if we were to connect Q's output to P's input. Knowing that P and Q are commutative with respect to each other we can reorder them (e.g., for performance reasons) without affecting the behavior of the pipeline.

In this paper, we present a technique called Σ^* that enables automated analyses of input-output properties of programs by *learning*. Σ^* uses dynamic symbolic execution [16, 28, 49] to discover symbolic input-output steps of a program, and counterexample guided abstraction refinement [7, 20] to over-approximate program behavior, and combines the two techniques into a sound and complete (relative to abstraction) symbolic learning algorithm. Σ^* works on the control-flow graph without assuming any specific

* This work was done while the second author was with UC Berkeley.

[1] In these code snippets, function **out**() writes a data item to the output stream, **peek**() returns the item at the given offset in the input stream without consuming it, and **in**() consumes and returns the current item from the input stream.

POPL'13, January 23–25, 2013, Rome, Italy.

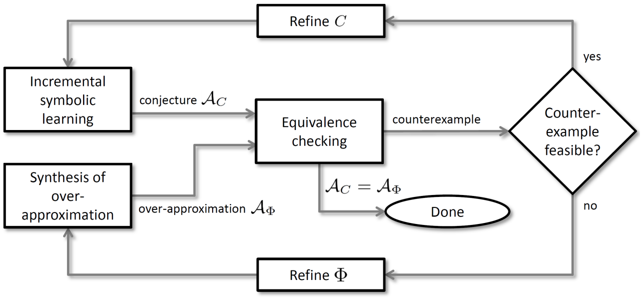

Refine C
Incremental symbolic learning
conjecture $\mathcal{A}_C$
Equivalence checking
counterexample
Counter-example feasible?
yes
no
Synthesis of over-approximation
over-approximation $\mathcal{A}_\Phi$
$\mathcal{A}_C = \mathcal{A}_\Phi$
Done
Refine Φ

Figure 1: The Design of Σ^*. Arrows indicate the data flow.

syntactic structure of the program and, by learning, synthesizes a faithful *symbolic model* of program behavior. It represents the models using a variant of symbolic transducers (STs) [14], which generalize classical finite-state machines by allowing transitions labeled with arbitrary predicates as input guards and terms as output. Assuming formulae from a decidable theory, STs can be effectively composed and equivalence-checked, enabling analysis of various behavioral properties such as commutativity, reversibility, and idempotence.

Σ^* targets applications beyond just verification. As we show, Σ^* enables domain-specific compiler optimizations (e.g., filter fusion and reordering [2, 45]) for a wider class of stream programs than currently possible. Also, STs inferred by Σ^* lend themselves to vectorized implementation (e.g., by utilising SIMD instructions [57], GPU architecture [15, 56] or field-programmable gate arrays [35, 39]) and data-parallel execution (e.g., by replication or using speculative techniques such as [48]).

Overview of Σ^****.*** In a nutshell, Σ^* can be seen as an extension of Angluin's L^* automata-learning algorithm [5] to learning models of programs. L^*, designed to learn regular languages in a black-box manner, is not applicable to the general software setting for two reasons. Firstly, L^* requires a priori knowledge of the concrete alphabet of the language. While that fits certain scenarios (such as inference of component interfaces [4, 27, 51], where the alphabet corresponds to the set of component's methods), the exact alphabet of software internals (including, for instance, all values that arise during the execution) is hard to specify in advance. Even if known, such alphabets are typically large (or practically infinite), so treating them concretely as in L^* hinders scalability. Secondly, L^* merely conjectures the inferred model and to check the conjecture relies upon an oracle (a teacher) that can answer equivalence queries. An omniscient oracle for equivalence checking against software artifacts is, however, computationally intractable.

Therefore, Σ^* is built around two key ideas:

1. Σ^* uses dynamic symbolic execution to discover constraints on input values (path predicates) and terms generating output values, and builds symbolic alphabet out of these. By using symbolic alphabets to represent equivalence classes of input-output steps, Σ^* enables discovery of complete input-output behavior independently of the concrete alphabet size.
2. Instead of equivalence checking against the program, Σ^* builds an abstraction that is a finite-state symbolic over-approximation of program behavior that we wish to learn. Checking equivalence of (deterministic) symbolic conjectures and such (possibly non-deterministic) over-approximations can be done algorithmically. Σ^* implements an algorithm that either generates a *separating sequence* (counterexample) showing how the conjecture and abstraction differ, or proves they are equivalent.

Starting with the trivial symbolic conjecture and the coarsest abstraction, Σ^* iteratively refines the conjecture or the abstraction until the two become equivalent (see Figure 1). If the counterexample returned by the equivalence checking algorithm is spurious (i.e., 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. Upon termination, the final conjecture is in a certain sense minimal behaviorally equivalent model of the program.

The symbolic conjecture maintained by Σ^* is a variant of symbolic finite-state transducers [14] that we call symbolic *lookback* transducers (SLTs). To generate outputs, SLTs use a sliding window over a bounded number of past inputs (SLTs can also be augmented with *lookahead* to allow peeking of incoming inputs). Along with conjectures, we also synthesize over-approximations of program behavior in the form of SLTs. We use predicate abstraction [7, 30] to abstract away the internal control flow of a program, however, in doing so we keep the input-output data flow intact. We then transform the obtained abstraction into an equivalent non-deterministic SLT. To check equivalence of the conjecture and the over-approximation, we use our algorithm for equivalence checking of a deterministic and non-deterministic SLT.

The main technical result of the paper is that Σ^* is relatively complete with respect to the abstraction. Assuming that the program behavior can be represented with an SLT, Σ^* terminates with an SLT behaviorally equivalent to the program if the overapproximation scheme eventually produces an inductive invariant implying the program's input-output relation. Since synthesis of the overapproximating SLT is guided by abstraction refinement, relative completeness of Σ^* is conditioned by completeness of the predicate selection method in refinement of the predicate abstraction [9, 42].

Other classes of program behavior could be learned by varying the type of symbolic conjectures. For programs whose behavior cannot be represented within the given class of symbolic models, Σ^* will end up conjecturing more refined models ad infinitum. However, if Σ^* is interrupted before convergence, the conjecture is guaranteed to be correct up to the explored depth.

Applications. To demonstrate practical utility of Σ^*, we report three experimental studies conducted with our implementation:

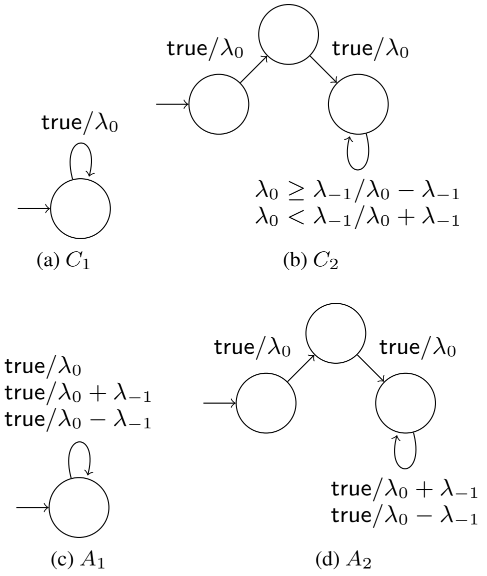
- *Web Sanitizers Verification.* We analyzed sanitizers from Google AutoEscape web sanitization framework and found out that 86% of them (12 out of 14) can be represented as SLTs. Σ^* successfully inferred all 12 sanitizers fully automatically, enabling their further analysis with systems such as Bek [37].
- *Stream Filter Optimization.* We applied Σ^* to infer filters from various stream processing applications [18, 23, 33, 45, 54]. We show how Σ^* enables optimizations such as reordering and fusion that are not possible with the current state-of-the-art.
- *Parallelization.* We show how SLT models of filters enable automated parallelization. From a variety of SLTs, we generated SIMD implementations using Intel SSE intrinsics and GPU implementations using NVIDIA CUDA, and obtained speedups of up to 3.89 against the most optimized `gcc` code.

Contributions. We claim the following contributions.

- *Symbolic Learning Framework* Σ^*. We present an automated learning framework Σ^* for learning symbolic models of programs's input-output behavior. The key insight behind Σ^*, besides use of symbolic alphabets to represent program steps, is use of over-approximation to detect convergence.
- *Symbolic Lookback Transducers.* We introduce a carefully limited variant of symbolic transducers that is amenable to learning

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

Figure 2: Code Snippet for the Running Example in §2.


true/λ_0
true/λ_0
true/λ_0
$\lambda_0 \geq \lambda_{-1}/\lambda_0 - \lambda_{-1}$
$\lambda_0 < \lambda_{-1}/\lambda_0 + \lambda_{-1}$
true/λ_0
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$
true/λ_0
true/λ_0
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$

(a) C_1

(b) C_2

(c) A_1

(d) A_2

Figure 3: Conjectures and Abstractions for the Example in Figure 2.

and develop an equivalence checking algorithm. We experimentally show that SLTs are useful in practice through three case studies, two of which demonstrate novel use of transducers.

- *Synthesis of Over-approximations.* We show how to transform predicate abstraction of a program into an SLT over-approximating the program behavior, and how to refine it. The key step of the synthesis is finitization of the unabstracted components of states that correspond to the input-output flow.
- *Relative Completeness.* Our main technical result is that Σ^* converges to a sound and complete model of SLT programs, as long as the abstraction refinement eventually builds a strong enough set of predicates. Effectively, this result reduces the completeness of learning to the completeness of abstraction.

2. Σ^* Illustrated by Example

We begin by showing how Σ^* works on the example given in Figure 2. The program reads input in an infinite loop using commands **peek**() and **in**(); **peek**(0) reads the current input symbol and leaves it in the input stream, while **in**() reads the current input symbol and moves onto the next one. Both commands halt 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.

We represent learned conjectures and synthesized abstractions in the example as SLTs with lookback 1. Each transition is labeled by a symbolic predicate-term pair of the form p/t. The transition is taken when the predicate p evaluates to true for a given sequence of concrete input symbols. The term t describes the computed output. Predicates and terms are expressions over input symbols and constants. We use λ_0 to denote the current symbol and λ_{-i} to denote symbols read i transitions before. SLTs with lookback k are allowed to read only the current and the last k symbols.

We use (dynamic) symbolic execution [16, 28, 49] to gather sequences of predicates and output terms occurring along the paths in the program execution. For instance, given an input of length four, a possible execution might take twice the then branch of the first if, then the else branch of the second if, and finally the then branch of the second if. For that execution we would obtain a predicate sequence (the path condition) $\vec{p} = \mathsf{true} \cdot \mathsf{true} \cdot (\mathsf{cur}_3 \geq \mathsf{prev}_2) \cdot (\mathsf{cur}_4 < \mathsf{prev}_3)$, and an output sequence $\vec{t} = \mathsf{cur}_1 \cdot \mathsf{cur}_2 \cdot (\mathsf{cur}_3 - \mathsf{prev}_2) \cdot (\mathsf{cur}_4 + \mathsf{prev}_3)$. In sequences such as $\vec{p}$ and $\vec{t}$, we use "$\cdot$" to denote the concatenation operator.

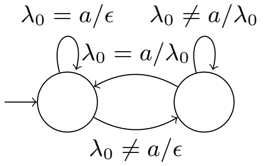
To obtain predicates and output terms formulated with regard to the input, we relativize all references of variables in $\vec{p}$ and $\vec{t}$ so that they become expressed with respect to the current position in the input stream. For instance, the relativized version of $\vec{p}$ would be the sequence $\mathsf{true} \cdot \mathsf{true} \cdot (\lambda_0 \geq \lambda_{-1}) \cdot (\lambda_0 < \lambda_{-1})$, and of $\vec{t}$, the sequence $\lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_{-1}) \cdot (\lambda_0 + \lambda_1)$. As for SLTs, the variable λ_{-j} in each element $\vec{p}|_i$ of the sequences above corresponds to the input symbol j positions before the current position (i) in the input stream. Thus, e.g., λ_0 in the third and fourth element of the sequences correspond to the last two symbols read by the program. To generate a concrete input from relativized path conditions, we need to derelativize them and pass the derelativized formula to an SMT solver. In the case above, the solver might return a concrete sequence $0 \cdot 1 \cdot 3 \cdot 2$ satisfying the relativized path condition.

Σ^* begins processing the example by learning the first conjecture C_1 (Figure 3a), and compares it against predicate abstraction A_1 of the program with respect to an empty set of predicates (Figure 3c). Note that, unlike in the classic predicate abstraction, we keep the input-output data flow intact. Equivalence checking of C_1 and A_1 produces the first symbolic counterexample, say a predicate $\vec{p_1} = \mathsf{true}$ and output $\vec{t_1} = (\lambda_0 + \lambda_{-1})$, which is a behavior captured by the abstraction, but not the conjecture. Any concrete sequence satisfying those two formulae is a concrete counterexample. For example, take input $\lambda_{-1} = 1, \lambda_0 = 1$. The program will produce output λ_0 for that input without reading λ_{-1}, showing that $\vec{p_1}/\vec{t_1}$ is spurious because symbolic outputs do not match. We refine the abstraction with respect to a new set of predicates, obtained by some predicate selection method (e.g., by taking atomic predicates in the weakest precondition computed along the path [6]). Suppose such method returns $\{i = 0, i \geq 2\}$ at this instance. The refined abstraction A_2 is shown in Figure 3d. 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 $\vec{t_2} = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_{-1})$. 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, Σ^* learns conjecture C_2 shown in Figure 3b. C_2 is correct, but Σ^* 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 $\vec{t_3} = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 + \lambda_1)$. The third counterexample is not spurious either. Thus, once again, we need to refine the abstraction. Using the same method for refinement as above, we extend the set of predicates with $\lambda_0 < \lambda_{-1}$. Predicate abstraction computes A_3, which is equivalent to C_2, and the process terminates, faithfully inferring the program's input-output relation, i.e., the specification of program's input-output behavior.

For the following example with nested loops, Σ^* terminates inferring a SLT with lookback 0, shown on the right.

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



$\lambda_0 = a/\epsilon$
$\lambda_0 \neq a/\lambda_0$
$\lambda_0 = a/\lambda_0$
$\lambda_0 \neq a/\epsilon$

3. Transductive Programs

We now turn to the formal development of Σ^*. In this section, we formalize the syntax and semantics of programs that transform an input stream into an output stream of symbols. Behavior of certain such programs can be represented by SLTs (§4). We also formally define concepts associated with the dynamic symbolic execution, which we use in development of over-approximations (§5) and learning (§6). To simplify exposition, henceforth we assume programs working with streams of integers.

3.1 Preliminaries

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. For a set D, we denote by D^* the free monoid generated by D with concatenation as the operation and the empty word ϵ as identity. We refer to words in D^* as sequences. Sequences of predicates (resp. terms) we denote by $\vec{p} = p_1 \cdots p_m$ (resp. $\mathbf{t} = t_1 \cdots t_n$). For sequences of sequences of terms, we write $\vec{t}$. We denote the length of $\vec{p}$ by $|\vec{p}|$, the i-th element of $\vec{p}$ by $\vec{p}|_i$ and the subsequence $p_i \cdots p_{i+k}$ by $\vec{p}|_{[i,i+k]}$. For $f \colon D \to C$ and $D' \subseteq D$, we write $f|_{D'}$ to denote the restriction of f on D'. For $\vec{a}$ and $\vec{b}$ such that $n \triangleq |\vec{a}| = |\vec{b}|$, we write $f[\vec{a} \leftarrow \vec{b}]$ to denote the function f' such that for all $i \in \{1, \ldots, n\}$, $f'(\vec{a}|_i) = \vec{b}|_i$ and for all $x \notin \{\vec{a}|_1, \ldots, \vec{a}|_n\}$, $f'(x) = f(x)$. We use $\mathbf{t}[x \mapsto y]$ to denote the term obtained from $\mathbf{t}$ by simultaneously replacing all free occurrences of x with y. $\mathsf{Vars}(t)$ denotes the set of all free variables in t.

3.2 Syntactic Representation

We first define syntactic representation of programs. Let V be the set of program variables. Expressions (terms) $\mathit{Exp}[\mathsf{V}]$, predicates $\mathit{BExp}[\mathsf{V}]$, and atomic commands $\mathit{Cmd}[\mathsf{V}]$ over V are given by:

$$\begin{array}{rcll} e & ::= & \mathsf{k} \mid x \mid \mathsf{f}(\mathbf{e}) \mid \ldots & \in \mathit{Exp}[\mathsf{V}] \\ b & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg b \mid b \wedge b \mid b \vee b \mid & \\ & & e = e \mid e \neq e \mid \mathsf{R}(\mathbf{e}) \mid \ldots & \in \mathit{BExp}[\mathsf{V}] \\ c & ::= & \mathsf{assume}(b) \mid x := e \mid x := \mathsf{peek}(\mathsf{k}) \mid & \\ & & x := \mathsf{in}() \mid \mathsf{out}(e) & \in \mathit{Cmd}[\mathsf{V}] \end{array}$$

where $\mathsf{k} \in \mathbb{Z}$, $x \in \mathsf{V}$, while f and R are constructors for terms and predicates. As long as the quantifier-free theory of $\mathit{Exp}[\mathsf{V}]$ with equality and satisfiability of $\mathit{BExp}[\mathsf{V}]$ are decidable, particular details of the grammar are not important. The command $x := \mathsf{peek}(\mathsf{k})$ reads a data item at offset k from the current symbol in the input stream and stores the item to x if the input is available, otherwise it causes the program to halt. The $\mathsf{in}()$ command works like $\mathsf{peek}(0)$ but in addition consumes the item from the input stream and moves onto the next input symbol. The command $\mathsf{out}(e)$ writes the value of e to the output stream.

We represent programs using control-flow automata [36] over the language of atomic commands. The control-flow automaton is determined by a set of control nodes N containing a distinguished node $\mathsf{s_N} \in \mathsf{N}$ representing the starting point of the program and a function $\mathsf{succ} \colon \mathsf{N} \times \mathit{Cmd}[\mathsf{V}] \rightharpoonup \mathsf{N}$ 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.

$$\begin{array}{rcl} [\![\mathsf{k}]\!]_\sigma & = & \mathsf{k} \\ [\![x]\!]_\sigma & \doteq & \begin{cases} \sigma(x) & \text{if } x \in \mathsf{V}, \\ x & \text{if } x \in \mathsf{V_I}, \\ \text{undefined} & \text{otherwise} \end{cases} \\ [\![\mathsf{f}(\mathbf{e})]\!]_\sigma & \doteq & \mathsf{f}([\![e_1]\!]_\sigma, \ldots, [\![e_n]\!]_\sigma) \end{array}$$

Figure 4: Symbolic Evaluation of Expressions.

$$\begin{array}{lcl} (\mathsf{n}, \sigma, \rho, \imath, \jmath) & \overset{\mathsf{assume}(b)}{\rightsquigarrow} & (\mathsf{n}', \sigma, \rho, \imath, \jmath) \\ (\mathsf{n}, \sigma, \rho, \imath, \jmath) & \overset{x:=e}{\rightsquigarrow} & (\mathsf{n}', \sigma[x \leftarrow [\![e]\!]_\sigma], \rho, \imath, \jmath) \\ (\mathsf{n}, \sigma, \rho, \imath, \jmath) & \overset{x:=\mathsf{peek}(\mathsf{k})}{\rightsquigarrow} & (\mathsf{n}', \sigma[x \leftarrow [\![in_{\imath+\mathsf{k}}]\!]_\sigma], \rho, \imath, \jmath) \\ (\mathsf{n}, \sigma, \rho, \imath, \jmath) & \overset{x:=\mathsf{in}()}{\rightsquigarrow} & (\mathsf{n}', \sigma[x \leftarrow [\![in_\imath]\!]_\sigma], \rho, \imath + 1, \jmath) \\ (\mathsf{n}, \sigma, \rho, \imath, \jmath) & \overset{\mathsf{out}(e)}{\rightsquigarrow} & (\mathsf{n}', \sigma, \rho[out_\jmath \leftarrow [\![e]\!]_\sigma], \imath, \jmath + 1) \end{array}$$

Figure 5: Symbolic Semantics of Commands.

3.3 Symbolic Semantics

Given a program $\mathcal{P}$ we now define its symbolic semantics. We formalize the contents of the input and the output stream with sets of variables $\mathsf{V_I} \triangleq \{in_1, in_2, \ldots\}$ (in-variables) and $\mathsf{V_O} \triangleq \{out_1, out_2, \ldots\}$ (out-variables), respectively. On a particular run of $\mathcal{P}$, all except finitely many of these variables are undefined, and for those which are defined, $in_\imath$ corresponds to the $\imath$-th element of the input stream, and $out_\jmath$ to the $\jmath$-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 ℓ_{in} to denote the length of the data sequence in the input stream.

We evaluate expressions on a memory $\sigma \in \mathsf{Mem} \triangleq \mathsf{V} \rightharpoonup \mathit{Exp}[\mathsf{V_I}]$ as indicated in Figure 4. We execute commands on states of the form $\mathsf{State} \triangleq \mathsf{N} \times \mathsf{Mem} \times \mathsf{OutStr} \times \mathbb{N} \times \mathbb{N}$. The third component, $\mathsf{OutStr} \triangleq \mathsf{V_O} \rightharpoonup \mathit{Exp}[\mathsf{V_I}]$, represents the symbols in the output stream. The last two components of the state, the in-index and the out-index, store indices of the current element in the input stream and the output stream, respectively. We denote the components of a state $s \in \mathsf{State}$ by $s.\mathsf{n}$, $s.\sigma$, $s.\rho$, $s.\imath$, and $s.\jmath$, respectively. The initial state is $s_0 = (\mathsf{s_N}, \emptyset, \emptyset, 1, 1)$.

We represent the symbolic semantics of $\mathcal{P}$ by a labeled transition system (LTS) $\mathcal{T} = (\mathsf{State}, C, \rightsquigarrow)$ with State as the set of states, $C \subseteq \mathit{Cmd}[\mathsf{V}]$ as the finite set of labels, and $\rightsquigarrow$ as the labeled transition relation. The relation $\rightsquigarrow \subseteq \mathsf{State} \times C \times \mathsf{State}$ is defined by the rules given in Figure 5, showing 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 $(s, c, s') \in \rightsquigarrow$.

3.4 Dynamic Symbolic Traces, Path Predicates and Outputs

The LTS $\mathcal{T}$ encodes the symbolic semantics of $\mathcal{P}$ with respect to any input. Given a concrete input $\vec{a} \in \mathbb{Z}^*$, we can construct a sequence of states in $\mathcal{T}$ representing the (dynamic) symbolic trace of $\mathcal{P}$ driven by $\vec{a}$, as in [16, 28, 49]. We start with the initial state s_0 and at each step we follow the transition $s_i \overset{c}{\rightsquigarrow} s_{i+1}$, such that if c is of the form $\mathsf{assume}(b)$ then b is true in $s_i.\sigma[\vec{in} \leftarrow \vec{a}]$. We stop if we ever reach a state s_f, which we call ending, such that the in-index $s_f.\imath$ equals $\ell_{\mathsf{in}} + 1$ or s_f has an outgoing $\overset{\_:=\mathsf{peek}(\mathsf{k})}{\rightsquigarrow}$-transition with k such that $s_f.\imath + \mathsf{k} > \ell_{\mathsf{in}}$. We define the symbolic trace of

$\mathcal{P}$ on input $\vec{a}$, denoted $\tau_{\mathcal{P}}(\vec{a})$, as the finite sequence $s_0 s_1 \dots s_f$ if an ending state is reached, or as the infinite sequence $s_0 s_1 \dots$ otherwise.

We refer to a state s_i as *in-state* if $s_i = s_0$, $s_i = s_f$ or the incoming transition reads an input symbol, i.e., $s_{i-1} \overset{\_:=\mathsf{in}()}{\rightsquigarrow} s_i$. Let $\tau^{\mathsf{in}}_{\mathcal{P}}(\vec{a}) \triangleq \tilde{s}_0 \tilde{s}_1 \dots \tilde{s}_n$ be the sequence of in-states from $\tau_{\mathcal{P}}(\vec{a})$. We define *big-step transition* as a transition between two successive states in $\tau^{\mathsf{in}}_{\mathcal{P}}(\vec{a})$. For $1 \leq i \leq n$, let p_i be the conjunction of predicates b of the form $\mathsf{assume}(b)$ encountered on all transitions between $\tilde{s}_{i-1}$ and $\tilde{s}_i$. In case no $\overset{\mathsf{assume}(\_)}{\rightsquigarrow}$-transitions exist between $\tilde{s}_{i-1}$ and $\tilde{s}_i$ we set p_i to $\mathtt{true}$. Furthermore, for $1 \leq i \leq n$, let $\mathbf{t}_i$ be the sequence of symbolic outputs (as written to out-variables by $\overset{\mathsf{out}(\_)}{\rightsquigarrow}$-transitions) between $\tilde{s}_{i-1}$ and $\tilde{s}_i$. If no output is generated we set $\mathbf{t}_i$ to ϵ.

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

3.5 Transductive and k-lookback Programs

In order to be able to learn program behavior, we have to make some assumptions about programs. Firstly, when considering the input-output behavior of a program, no path in a program should end up in a loop without reading an input, i.e., we will not allow infinite periods of silence during which the program would not read any input. This first assumption will allow us to extend incomplete conjectures of program behavior with new big-step transitions. Secondly, we will assume that programs produce uniformly bounded number of output symbols per each big-step transition. This second assumption is inherent to symbolic transducers [14], on which we base SLTs (§4), the model of program behaviour used in Σ^*.

More formally, let $\mathcal{T}$ be the LTS of $\mathcal{P}$ and ξ any symbolic trace in $\mathcal{T}$ starting with s_0 and ending with an in-state. Let us denote by $\#in(\xi)$ the number of $\overset{\_:=\mathsf{in}()}{\rightsquigarrow}$-transitions in ξ and by $\#out(\xi)$ the number of $\overset{\mathsf{out}(\_)}{\rightsquigarrow}$-transitions in ξ. We say that $\mathcal{P}$ is *transductive* if there are no infinite symbolic traces in $\mathcal{T}$ with only finitely many in-states, and there exists $M \in \mathbb{N}$ such that for every ξ as above we have $\#out(\xi) < M \cdot \#in(\xi)$. As a consequence, if $\mathcal{P}$ is transductive then for every input $\vec{a}$, $\tau_{\mathcal{P}}(\vec{a})$ is finite and $|\theta_{\mathcal{P}}(\vec{a})| = |\gamma_{\mathcal{P}}(\vec{a})| < M \cdot |\vec{a}|$.

Besides transductiveness, we impose an additional requirement that is specific to the type of transducers that we use to represent program behavior in Σ^*, but that could be relaxed if we would use more expressive models (such as, e.g., transducers with registers [3, 14]). This additional requirement asks that transductive programs produce outputs from a bounded number of inputs in the past. More formally, we say that $\mathcal{P}$ is *k-lookback* if $\mathcal{P}$ is transductive and for every ξ as above, each $\overset{\mathsf{out}(e)}{\rightsquigarrow}$-transition in ξ depends only on previous k inputs, i.e., if $s \overset{\mathsf{out}(e)}{\rightsquigarrow} s'$ then $\mathsf{Vars}(\llbracket e \rrbracket_{s.\sigma}) \subseteq \{in_{s.\imath - k}, \dots, in_{s.\imath}\}$. In §5 and §6 we focus on a class of k-lookback programs whose behavior can be represented with SLTs.

3.6 Input Relativization

In development of over-approximations (§5) and the learning algorithm (§6), we will need to rephrase symbolic expressions pertaining to program states so that they use the offset from the current position in the input stream rather than from the beginning. We show how to relativize these expressions so that the offset becomes relative to the current in-index.

Let $\mathsf{V}_\lambda \triangleq \{\dots, \lambda_{-1}, \lambda_0, \lambda_1, \dots\}$ be an infinite set of λ-variables, indexed by a relative offset. We use λ-variables so that λ_0 stands for the current symbol in the input stream, λ_{-i} for the symbol i positions back and λ_i for the symbol i positions ahead. Let $\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}$ denote a variable substitution that maps each in-variable $in_\jmath$ to λ-variable $\lambda_{\jmath - \imath}$. We define *relativization* of a state $s = (\mathsf{n}, \sigma, \rho, \imath, \jmath)$ by $\Lambda(s) \triangleq (\mathsf{n}, \sigma[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \rho[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \jmath)$. Intuitively, $\Lambda(s)$ relativizes symbolic expressions in s with respect to the current in-index. For general transductive programs, expressions in σ- and ρ-components of $\Lambda(s)$ may use unboundedly many λ-variables as the expressions can refer to inputs from arbitrary far in the past. However, k-lookback programs use only up to k λ-variables with negative index, namely, $\lambda_{-k}, \dots, \lambda_{-1}$.

4. Symbolic Transducers with Lookback

In the last section, we formalized the syntax and semantics of programs. In this section, we introduce symbolic lookback transducers (SLTs) with a constructive equivalence checking algorithm. We use SLTs to represent synthesized over-approximations (§5) and symbolic conjectures in the learning algorithm (§6).

4.1 Background

Finite-state transducers are an extension of classic finite-state automata obtained by allowing output on transitions (see [46]). Symbolic finite-state transducers [14] extend finite-state transducers by symbolic transitions, which are defined using predicates for input guards and symbolic terms for output. Our notion of SLTs extends symbolic finite-state transducers with a sliding window over input, which allows a transducer to generate outputs based on a bounded number of inputs from the past.

4.2 Definitions

We now formally define symbolic finite-state transducers with k-lookback — 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 k-SLT is effectively a window of size $k + 1$, reading the current and the last k symbols. Equivalently, such a transducer can 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, we use the set of λ-variables $\mathsf{V}^k_\lambda := \{\lambda_{-k}, \dots, \lambda_{-1}, \lambda_0\}$ so that λ_0 is the current symbol and λ_{-i} is the symbol i positions back.

Definition 1. *A symbolic finite transducer with lookback k (k-SLT) is a tuple $\mathcal{A} = (Q, q_0, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq Q \times BExp[\mathsf{V}^k_\lambda] \times Q \times Exp[\mathsf{V}^k_\lambda]^*$ is a finite transition relation.*

In other words, SLT is a variant of a symbolic sequential ϵ-input-free (i.e., real-time) transducer having only final states,[2] and in general can be non-deterministic and does not have to be bounded-valued.

We say that SLT $\mathcal{A}$ 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. SLTs that are inferred by the Σ^*'s learning algorithm will always be deterministic and even more, transitions from every state will have mutually disjoint guards.

We next formally define how SLTs produce output. Before defining a run of an SLT, we introduce some convenient notation.

[2] It is unclear how to learn transducers with non-final states, as such transducers allow inherent ambiguity in where the output is produced. Indeed, existing algorithms for learning concrete transducers (e.g., [50, 58]) require all states to be final.

For brevity, we refer to the sequence $\lambda_{-k} \dots \lambda_0$ as $\overleftarrow{\lambda}$. To process the input, 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 $\vec{a}^{\perp} \triangleq \perp^k \cdot \vec{a}$.

A *run of* k-SLT $\mathcal{A} = (Q, q_0, \Delta)$ on $\vec{a} \in \mathbb{Z}^n$ is a finite sequence of states $q_0 \dots q_n$ 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 $\varphi_1, \dots, \varphi_n \in \mathit{BExp}[\mathsf{V}_\lambda^k]$ and $\mathbf{t}_1, \dots, \mathbf{t}_n \in \mathit{Exp}[\mathsf{V}_\lambda^k]^*$ such that for all $1 \le i \le n$, $\vec{a}^{\perp}|_{[i,i+k]}$ satisfies φ_i. We say that $\mathcal{A}$ on the input $\vec{a}$ *produces the output* $\vec{o} \in \mathbb{Z}^*$ and write $\vec{a} \twoheadrightarrow_{\mathcal{A}} \vec{o}$ if for all $1 \le i \le n$, $\mathbf{o}_i = \mathbf{t}_i \left[\overleftarrow{\lambda} \mapsto \vec{a}^{\perp}|_{[i,i+k]}\right]$, where $\vec{o}|_i = \mathbf{o}_i$.

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

We define the *transduction* of (a possibly nondeterministic) $\mathcal{A}$ as a function $\mathcal{T}_{\mathcal{A}} \colon \mathbb{Z}^* \to 2^{\mathbb{Z}^*}$ defined by $\mathcal{T}_{\mathcal{A}}(\vec{a}) \triangleq \{\vec{o} \mid \vec{a} \twoheadrightarrow_{\mathcal{A}} \vec{o}\}$. We say that $\mathcal{A}$ is *single-valued* if for all $\vec{a}$, $|\mathcal{T}_{\mathcal{A}}(\vec{a})| \le 1$.

4.3 Composition of SLTs

SLTs are closed under composition. Given a k-SLT $\mathcal{A}$ and a l-SLT $\mathcal{B}$, their composition is a $(k+l)$-SLT $\mathcal{A} \circ \mathcal{B}$ such that $\mathcal{T}_{\mathcal{A} \circ \mathcal{B}}(\vec{a}) = \{\mathcal{T}_{\mathcal{B}}(\vec{o}) \mid \vec{o} \in \mathcal{T}_{\mathcal{A}}(\vec{a})\}$. The composition of SLTs can be constructed similarly as the composition of symbolic finite-state transducers [14].

4.4 Equivalence Checking

Unfortunately, general equivalence checking of SLTs is undecidable as it is already undecidable to check equivalence of concrete non-deterministic ϵ-free finite-state transducers [41]. While the technique of Bjørner et al. [14] could be adapted to SLTs to decide the single-valued case, that would not suffice in our case as over-approximations of programs (§5) need not to be bounded-valued. Fortunately, our learning conjectures are always deterministic, so in fact we only need to check equivalence between a deterministic and a (possibly) non-deterministic SLT.

We present an efficient algorithm for equivalence checking between deterministic and non-deterministic SLTs, which is a symbolic adaptation of the algorithm from Demers et al. [22] for deciding single-valuedness of finite-state transducers. To check whether a deterministic and non-deterministic SLT are equivalent, it suffices to check whether their union (which is a non-deterministic SLT) is single-valued. Checking whether a non-deterministic SLT $\mathcal{A} = (Q, q_0, \Delta)$ is single-valued is efficiently decidable in $\mathcal{O}(|Q|^2)$ time [22] by checking whether a linear grammar generated from $\mathcal{A}$ generates a language of palindromes [38].

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 $N ::= TNT \mid \epsilon$, and the start symbol $S \in N$. From a k-SLT $\mathcal{A}$, generate a grammar $\mathcal{G} = (Q \times Q, \mathit{BExp}[\mathsf{V}_\lambda^k] \times \mathit{Exp}[\mathsf{V}_\lambda^k]^*, P, [q_0, q_0])$, 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 $\mathbf{t}_i \ne \perp$. By $\perp$, our learning algorithm denotes outputs on transitions that are either infeasible because of the constraints on the path condition, or subsumed by other predicates. $\mathcal{A}$ is single-valued iff $\mathcal{G}$ 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 $\mathcal{O}(|Q|^2)$ calls to the prover. If $\mathcal{A}$ is not single valued then we construct a witness for disequality between the deterministic and the non-deterministic SLT called *separating sequence* by finding the shortest path from the start symbol to the first reachable rule that does not generate a palindrome.

4.5 Extension with Lookahead

SLTs can be straightforwardly extended to incorporate lookahead. In our model of SLTs, the sliding window begins at the current input tape head position and covers k last symbols. Such window can also be made to cover symbols ahead of the current head position by using additional λ-variables $\lambda_1, \dots, \lambda_l$ such that λ_i stands for the symbol i positions ahead. A k-SLT with a lookahead l can be reduced to a $(k+l)$-SLT, so equivalence checking of SLTs with lookahead can be reduced to equivalence checking of SLTs.

5. Synthesis of Over-Approximations

We now show how to synthesize sound program over-approximations for a class of programs with finite lookback. Overapproximations are computed in two steps. In the first step, we construct an over-approximation of the program that is based on predicate abstraction [8, 30]. In the second step, we transform the obtained abstraction to an equivalent non-deterministic SLT. We refine such an SLT by augmenting the set of predicates in predicate abstraction.

5.1 Abstraction of Transductive Programs

The goal of the abstraction is to abstract away the internal computation of the program but in doing so to keep all of its original input-output behavior. Let us consider the LTS $\mathcal{T} = (\mathsf{State}, C, \rightsquigarrow)$ of a transductive program $\mathcal{P}$ as defined in §3.3. We parameterize our abstraction by a set of predicates Φ over program variables, interpreted over $\mathsf{V} \rightharpoonup \mathbb{Z}$. Let us denote by $\mathsf{Pred}(\Phi)$ the set of boolean combinations over predicates from Φ (i.e., all minterms). We define the abstraction of $\mathcal{T}$ as the LTS $\mathcal{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$ constructed as follows. The set of abstract states $\mathsf{State}^\sharp$ is given by

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

where the valuations of program variables are mapped to predicates in $\mathsf{Pred}(\Phi)$ satisfied by the valuation, while the data variables $\mathsf{V_D} \subseteq \mathsf{V}$ that are live (i.e., for which there exists data flow to output terms), are kept intact along with other components of the state. 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 2. *For every input $\vec{a}$, if $\tau_{\mathcal{P}}(\vec{a}) = s_0 \dots s_n$ is a trace in $\mathcal{T}$, then there exists a trace $\tau^\sharp_{\mathcal{P}}(\vec{a}) = s_0' \dots s_n'$ in $\mathcal{T}^\sharp$ such that for every $0 \le i \le n$, $s_i.\mathsf{n} = s_i'.\mathsf{n}$, $s_i.\rho = s_i'.\rho$ $s_i.\imath = s_i'.\imath$, $s_i.\jmath = s_i'.\jmath$, $s_i.\sigma|_{\mathsf{V_D}} = s_i'.\sigma$ and $s_i.\sigma$ satisfies $s_i'.\varphi$. Consequently, the output $\gamma^\sharp_{\mathcal{P}}(\vec{a})$ corresponding to the trace $\tau^\sharp_{\mathcal{P}}(\vec{a})$ is equal to $\gamma_{\mathcal{P}}(\vec{a})$.*

5.2 Translation to SLTs

We now translate the abstract LTS $\mathcal{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$ into an equivalent (possibly non-deterministic) SLT. 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[3] 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 $\sigma_\lambda = \sigma_\lambda'$.

Relation $\sim$ can have infinitely many equivalence classes in general, however, we focus on programs having finite-index $\sim$, as we can represent such programs with SLTs and learn with Σ^*.

Definition 3. *We say that $\mathcal{P}$ is SLT-*representable *if for any choice of Φ, the corresponding relation $\sim$ is of finite index.*

SLT-representable programs necessary have a bounded lookback.

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

[3] We extend the definition of input-relativization to $s = (\mathsf{n}, \varphi, \sigma, \rho, \imath, \jmath) \in \mathsf{State}^\sharp$ by $\Lambda(s) \triangleq (\mathsf{n}, \varphi[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \sigma[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \rho[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \jmath)$.

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 ξ and let $\theta^{\sharp}(\xi)$ be the produced symbolic output. We need the following lemma for our translation to an SLT to be well-defined.

Lemma 4. *If $s \sim s'$ and $t \sim t'$ then for every path $\xi \in \mathsf{paths}^{\sharp}_{\mathsf{in}}(s,t)$, there exists an equivalent 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 $\mathcal{A}_\Phi$ to be the SLT (Q, q_0, Δ) with $Q \triangleq \{[s]_\sim \mid s \text{ is } in\text{-state}\}$ as the set of states, $q_0 \triangleq [s_0]_\sim$ as the initial state and Δ 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, $\mathcal{A}_\Phi$ represents all isomorphism classes of big-step transitions between the abstracted in-states.

Lemma 5. *If $\mathcal{P}$ is SLT-representable then $\mathcal{A}_\Phi$ is an SLT.*

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

Proposition 6. *For all $\vec{a}$, $\gamma^{\sharp}_{\mathcal{P}}(\vec{a}) = \vec{o}$ iff $\vec{a}^{\perp} \twoheadrightarrow_{\mathcal{A}_\Phi} \vec{o}$.*

Corollary 7. *For all $\vec{a}$, if $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$ then $\vec{a}^{\perp} \twoheadrightarrow_{\mathcal{A}_\Phi} \vec{o}$.*

5.3 Refinement

Suppose that $\mathcal{A}_\Phi$ strictly over-approximates the behavior of an SLT-representable $\mathcal{P}$ on some input $\vec{a}$, i.e., there exist $\vec{o}$ and $\vec{o}'$, $\vec{o} \neq \vec{o}'$, such that $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$ and $\vec{a} \twoheadrightarrow_{\mathcal{A}_\Phi} \vec{o}'$. Then we need to refine the abstraction $\mathcal{A}_\Phi$. We do so by adding enough predicates to Φ to evidence infeasibility of the spurious run in $\mathcal{A}_\Phi$ that generates $\vec{o}'$.

Our approach to finding a new set of predicates for refining $\mathcal{A}_\Phi$ is based on standard counterexample feasibility analysis with weakest preconditions [6]. The basic idea is to build a predicate α from the spurious trace in $\mathcal{A}_\Phi$ generating $\vec{o}'$ so that α is unsatisfiable if and only if the given trace is infeasible in $\mathcal{P}$. Starting with false, α is obtained by applying the weakest precondition of commands in $\mathcal{P}$ along the trace, until the beginning of the trace. To maintain the syntactic form of assume-predicates collected along the trace, we keep all substitutions in α explicitly. The set of all atomic predicates in α is then used to augment Φ.[4] Although more involved methods (e.g., based on Craig interpolation) are possible, our empirical evaluation shows that this heuristic works well in practice for our target classes of programs.

Completeness of the predicate selection method. Since our abstraction is based on predicate abstraction and fully precise isomorphic representation of other components of the state, $\mathcal{A}_\Phi$ in fact defines the strongest inductive invariant containing the reachable set of states of $\mathcal{P}$ that is expressible as a Boolean combination of the given set of predicates, while faithfully preserving the input-output relation in the relativized form. If there exists a quantifier-free inductive invariant ψ which can be built using some set of predicates Φ such that ψ uniquely identifies the reachable states of $\mathcal{P}$, then the construction of abstraction would converge. Assuming a complete decision procedure for the underlying theory and a predicate selection method that would eventually build such Φ, by the relative completeness of predicate abstraction [9, 42], we could generate an invariant as strong as ψ.

Although it is not clear how to construct such ψ directly from $\mathcal{P}$, we can construct it if the number of states n of an SLT $\mathcal{A}_{\mathcal{P}}$ that is behaviorally equivalent to $\mathcal{P}$ is known a priori—by explicitly encoding a checking sequence [46] that distinguishes $\mathcal{A}_{\mathcal{P}}$ from all other transducers up to n states. To abstract away the complexity of such construction, we assume existence of a predicate selection method that eventually yields a set Φ resulting in an abstraction $\mathcal{A}_\Phi$ equivalent to $\mathcal{P}$. We will say that a predicate selection method is *complete* if it is guaranteed to eventually generate a sufficient set of predicates to construct $\mathcal{A}_\Phi$ equivalent to $\mathcal{P}$. Our main result expresses completeness of our learning algorithm relative to existence of such a complete predicate selection method. However, we emphasize that the predicate selection mechanism does not affect the soundness of Σ^*, nor the quality of inferred models, only the eventual convergence.

6. Learning

In this section, we describe Σ^*'s learning algorithm and prove its main properties. Our algorithm iteratively builds conjectures similarly as L^*. To make the presentation reasonably self-contained, we first give a brief overview of L^* (§6.1). We then define the representation of conjectures (§6.2), followed by the detailed presentation of the learning algorithm (§6.3) and its properties (§6.4).

6.1 Background: Overview of L^*

Here we give an informal overview of the classic L^*. See the paper of Angluin [5] for a more detailed description (or Shahbaz and Groz [50] for the Mealy machines version of L^*).

The goal of L^* is to learn an unknown regular language D by generating a deterministic finite automaton (DFA) that accepts D. L^* starts by asking a teacher, who knows D, membership queries over a known concrete alphabet to check whether certain words are in D. The results of these queries are recorded in a so-called observation table. Membership queries are iteratively asked until certain technical conditions are met, upon which L^* conjectures an automaton. L^* asks the teacher an equivalence query to check if the conjectured language is equivalent to D. The teacher either confirms the equivalence or returns a counterexample, which is a word that distinguishes D from the conjecture. L^* uses the counterexample to devise new membership queries, refining the conjecture. This procedure is repeated until the conjecture becomes equivalent to D. If D is indeed a regular language then L^* is guaranteed to find a (minimal) DFA for D after at most a polynomial number of membership and equivalence queries.

In Σ^*, we answer the membership queries by using a combination of concrete and symbolic execution [16, 28, 49], and the equivalence queries by equivalence checking learned conjectures and increasingly refined abstractions, using the equivalence checking algorithm for SLTs (§4.4).

6.2 Definitions

We proceed by giving notational conveniences used in the section and defining Σ^*'s representation of the observation table.

We first define a relativization function for path predicates and symbolic outputs by $\Lambda(s_1 \dots s_n) \triangleq s_1[\vec{in} \overset{1}{\mapsto} \overleftarrow{\lambda}] \dots s_n[\vec{in} \overset{n}{\mapsto} \overleftarrow{\lambda}]$, where $\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}$ is the variable substitution defined in §3.6 and $\vec{s}$ is either a path predicate or symbolic output. If $\vec{s}$ is a symbolic output $\vec{t}$, the same relativization function is applied to each individual term in the subsequence, i.e., if $\mathbf{t} = t_1 \dots t_m$ then $\mathbf{t}[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] = t_1[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] \dots t_m[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}]$. We next define $\mathsf{witness}$ as a function that takes a sequence of relativized predicates, derelativizes them by applying the inverse of the Λ 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 an SMT solver, and returns a concrete sequence of input symbols of length $|\vec{p}|$ satisfying the conjunction, or $\perp$ if the conjunction is infeasible. Finally, we point out that all equality

[4] In some examples we employ additional heuristic that uses templates built from syntactic predicates in the code.

(resp. inequality) checks $=$ (resp. $\neq$) over predicates and terms in this section are syntactic equality (resp. inequality) checks.[5]

Σ^* constructs symbolic observation table similarly as L^*, but table entries are path predicates and symbolic outputs (§3.4), rather than concrete words. The finished table can be easily translated into an SLT, representing a conjecture. As in L^*, such conjecture will always be deterministic.

Definition 8. Symbolic observation table *is a quadruple* $(R, S, E, T) \subseteq (BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^* \times BExp[\mathsf{V}_\lambda]^* \to (Exp[\mathsf{V}_\lambda] \cup \{\bot, \epsilon\})^*)$*, where*

- $R \subseteq S$ *represents a set of identified states,*
- *S (resp. E) is a prefix- (resp. suffix-) closed set of relativized path predicates, and*
- *T is a map indexed by* $\vec{p}_\mathsf{p} \in S, \vec{p}_\mathsf{s} \in E$*, containing the suffix of the relativized symbolic output generated when processing* $\vec{p}_\mathsf{s}$ *immediately after* $\vec{p}_\mathsf{p}$*, i.e., if* $\vec{a} = \mathsf{witness}(\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s})$ *and* $\vec{t} = \Lambda(\theta_\mathcal{P}(\vec{a}))$ *then* $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] = \vec{t}_\mathsf{s}$*, where* $\vec{t}_\mathsf{s}$ *is a suffix of* $\vec{t}$ *such that* $|\vec{t}_\mathsf{s}| = |\vec{p}_\mathsf{s}|$*.*

For $\vec{p} \in S$*, 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 transitions in the table by* $\bot$*.*

Intuitively, R represents a set of shortest paths leading to discovered states, $S \supseteq R$ 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 SLT. Finally, E is the set of distinguishing tests that distinguish different states.

The classic L^* makes a conjecture when the table is *closed*, which means that every sequence in S has a representative in R, i.e., $\forall \vec{p} \in S \,.\, \exists \vec{r} \in R \,.\, \vec{p}\text{-}row = \vec{r}\text{-}row$. We define closedness in the same way as L^*. From a closed table, we can easily construct a complete (for all states and input symbols, all transitions are defined) SLT using standard techniques (e.g., see [5, 50]).[6]

6.3 The Learning Algorithm

We begin by describing the FILLROWS algorithm that computes the missing entries of the observation table, continue with the EXTENDTABLE algorithm that explores the successor states of all states discovered at certain step, and end with the Σ^*'s learning algorithm.

Algorithm 1 computes the missing entries in the observation 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 (Line 2). While the sets S (and R) contain path predicates that are collected along prefixes of some feasible paths in $\mathcal{P}$ starting from the initial state, the set E 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 $\mathcal{P}$ using concolic execution (Line 4) and collect the predicates ($\vec{r}$) and output terms ($\vec{t}$) 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 $\bot$. Lines 6–9 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 14–15 close the table.

```
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      $\vec{a} := \mathsf{witness}(\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s})$
3      if $\vec{a} \neq \bot$ then
4          $(\vec{r}, \vec{t}) := (\Lambda(\pi_\mathcal{P}(\vec{a})), \Lambda(\theta_\mathcal{P}(\vec{a})))$ // assert($|\vec{r}| = |\vec{t}|$)
5          $\vec{t}_\mathsf{s} := \epsilon$
6          forall the $1 \le i \le |\vec{t}|$ 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 $\vec{t}_\mathsf{s} := \vec{t}_\mathsf{s} \cdot \vec{t}|_i$
9                  else $\vec{t}_\mathsf{s} := \vec{t}_\mathsf{s} \cdot \bot$
10             else
                   // assert($\vec{p}_\mathsf{p}|_i = \vec{r}|_i$)
11         $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] := \vec{t}_\mathsf{s}$
12     else
13         $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] := \bot$
   // Close the table
14 forall the $\vec{p} \in S$ s.t. $\neg\exists \vec{r} \in R \,.\, \vec{p}\text{-}row = \vec{r}\text{-}row$ do
15     $R := R \cup \vec{p}$ // New state
16 Return $OT$
```

Algorithm 1: The FILLROWS Algorithm.

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. 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/$\bot$. The algorithm creates such a state automatically, if needed, by filling the corresponding row with $\bot$, as even the prefix to the ghost state is infeasible.

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 set E. In the next three lines, we extend and close the table, producing the first $\mathcal{A}_C$ conjecture and the first abstraction $\mathcal{A}_\Phi$ of $\mathcal{P}$. The loop beginning on Line 15 checks the equivalence between the conjecture and abstraction. If they are equivalent, the algorithm terminates returning the exact transducer implemented by $\mathcal{P}$. Otherwise, the counterexample is checked against $\mathcal{P}$. If it is spurious, we refine the abstraction, otherwise, we refine the conjecture. We use Shahbaz and Groz's [50] technique for processing the counterexamples adapted for our symbolic setting. First, we collect the predicates from $\mathcal{P}$ 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.

6.4 Properties

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

[5] 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 SLTs.

[6] In L^*, one also defines the *consistency* property, which would in our setting say that if two sequences $\vec{p}_1, \vec{p}_2$ from R are equivalent, then both states reached by $\gamma_\mathcal{P}(\mathsf{witness}(\vec{p}_1))$ and $\gamma_\mathcal{P}(\mathsf{witness}(\vec{p}_2))$ 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 $OT$
forall the $\vec{r} \in R$ do // Extend all sequences from $R$
    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
        $s := \mathsf{false}$
        while $s \not\Leftrightarrow \mathsf{true}$ do
            $r_{\mathsf{s}} := \epsilon$
            $\vec{a} := \mathsf{witness}(\vec{r} \cdot \neg s)$
            if $a = \perp$ then
                $r_s := \neg s$
                $s := \mathsf{true}$
            else
                $\vec{p} := \Lambda(\pi_{\mathcal{P}}(\vec{a}))$
                // assert$(|\vec{p}| = |\vec{r}| + 1)$ $r_{\mathsf{s}} := \vec{p}|_{|\vec{r}|+1}$
                $s := s \vee r_{\mathsf{s}}$
            $E := E \cup \{r_{\mathsf{s}}\}$
            $S := S \cup \{\vec{r} \cdot r_{\mathsf{s}}\}$
            $OT :=$FILLROWS$(OT)$
Return $OT$
```

Algorithm 2: The EXTENDTABLE Algorithm.

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

Algorithm 3: The Σ^*'s Learning Algorithm.

Lemma 9. *Let $T = (R, S, E, T)$ be a symbolic observation table. Then Σ^* 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 $\bigvee_{i=1}^n r_{\mathsf{s}}^i \Leftrightarrow$ true and for all i, it holds that $\vec{r} \cdot r_{\mathsf{s}}^i \in S$.*
4. *The conjecture $\mathcal{A}_C$ is closed at the end of each step.*

Correctness. R represents the part of the symbolic execution tree of $\mathcal{P}$ on which the conjecture faithfully represents the behavior of $\mathcal{P}$, which is stated with the following proposition.

Proposition 10 (Bounded correctness)**.** *After each step, for all $\vec{r} \in R$ and $\vec{a}$ such that $\vec{a} = \mathsf{witness}(\vec{r})$, $\gamma_{\mathcal{P}}(\vec{a}) = \gamma_{\mathcal{A}_C}(\vec{a})$ holds.*

One can rephrase bounded correctness as a guarantee given by Σ^* in case Σ^* is interrupted before reaching a convergence. In other words, we can consider Σ^* as performing bounded model checking of the program with a state-space exploration strategy guided by the learned model.

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

Lemma 11. *If at some step of Σ^*, $(\vec{a}, \vec{o})$ is a separating sequence such that $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$, then at the end of the step, for all $\vec{a}'$ such that $\vec{a}' = \mathsf{witness}(\pi_{\mathcal{P}}(\vec{a}))$, $\gamma_{\mathcal{A}_C}(\vec{a}') = \gamma_{\mathcal{P}}(\vec{a}')$ holds.*

We state the completeness of Σ^* relative to the completeness of the predicate selection method.

Theorem 12 (Relative completeness)**.** *If $\mathcal{P}$ is SLT-representable and the predicate selection method for refinement is complete, then Σ^* terminates with $\mathcal{A}_C$ being behaviorally equivalent to $\mathcal{P}$.*

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

The equivalence check of $\mathcal{A}_\Phi$ and $\mathcal{A}_C$ 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. □

Computational complexity. Next, we analyze the complexity of Σ^*. Let n be the number of states of the inferred 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 $n \cdot k \cdot (k + m \cdot n - m) = \mathcal{O}(n \cdot k^2 + m \cdot n^2 \cdot k)$ entries. The k factor is likely to be small in practice, and our equivalence checking algorithm finds the minimal counterexample. The SMT 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 $\mathcal{O}(n \cdot k + n^2)$, not including the number of calls required for abstraction refinement, which depends on the abstraction technique used.

7. Applications

In this section, we demonstrate practical utility of Σ^* by applying it in three different application domains and reporting experimental results obtained with our prototype implementation.

We built our implementation on top of KLEE [17] with STP [25] and CPAchecker [13]. We patched KLEE to generate relativized symbolic traces and answer membership queries for the learning algorithm (§6), and used STP in the equivalence checking algorithm (§4.4). We used CPAchecker for counterexample-guided predicate abstraction refinement (§5).

All experiments were run on a machine with Intel Core2 Quad 2.8 GHz CPU and NVIDIA GeForce GTX 460 GPU (with 7

Benchmark	Learned	Int. bits	#States	#Transitions	Lookback	$\#\mathcal{A}_C$ refinements	$\#\mathcal{A}_\Phi$ refinements	$\lvert\Phi\rvert$
`EncodeHtml` [37]	✓	8	1	2	0	0	1	10
`GetTags` [14]	✓	48	3	7	1	4	5	6
`CleanseAttribute`	✓	40	2	4	1	0	3	14
`CleanseCss`	✓	40	1	2	0	0	2	16
`CssUrlEscape`	✓	40	1	11	0	0	1	12
`HtmlEscape`	✓	8	$L+1$	$7 \cdot L$	L	L	$L+3$	$L+11$
`JavascriptEscape`	✓	16	3	16	2	2	6	20
`JavascriptNumber`	✓	41	17	19	16	17	23	41
`JsonEscape`	✓	8	$L+1$	$11 \cdot L$	L	L	$L+3$	$L+12$
`PreEscape`	✓	8	$L+1$	$5 \cdot L$	L	L	$L+3$	$L+6$
`UrlQueryEscape`	✓	8	2	11	0	2	5	19
`XMLEscape`	✓	8	$L+1$	$7 \cdot L$	L	$L-1$	$L+2$	$L+11$
`PrefixLine`	–	8						
`SnippetEscape`	–	8						

Table 1: Experimental Results. The benchmarks that Σ^* successfully learned are denoted by ✓. The last benchmark could be learned if Σ^* were extended to handle subsequential transducers. *Int. bits* is the size of the internal control and data state in bits, *#States* the number of states in the final SLT, *#Transitions* the number of transitions in the final SLT, *#$\mathcal{A}_C$ refinements* the number of refinements of the conjecture, *#$\mathcal{A}_\Phi$ refinements* the number of refinements of the abstraction, and $|\Phi|$ the size of the set of predicates in the final abstraction. In each of the four benchmarks whose rows contain symbolic expressions dependent on L, the particular L denotes the bound on the size of the buffer internally used by the benchmark. The value of L depends on the characteristics of the input language of the benchmark and is a priori fixed in our four benchmarks (e.g., for `HtmlEscape`, it is bounded by the maximal length of tag names, attribute names and special symbols).

1.42 GHz multiprocessors, 48 cores each). The learning algorithm inferred all learnable benchmarks described below in less than three seconds.

7.1 Web Sanitizers Verification

To make web applications more secure, sanitization is used to remove possibly malicious elements from user's input. Although small in numbers of lines of code, sanitizers in real-world applications are surprisingly difficult to implement correctly [10, 37], often because low-level implementation tricks are obscuring the intended input-output behavior. By automatically inferring input-output models of sanitizers, we can automatically check their properties such as commutativity, reversibility, and idempotence using tool such as Bek [37].

We evaluated Σ^* on the sanitizers from Google AutoEscape (GA) web sanitization framework, the same benchmarks used by Hooimeijer et al. [37]. We dropped the `ValidateUrl` sanitizer, as it is effectively just a wrapper for calling other sanitizers. In addition to GA sanitizers, we added to our benchmark suite the sanitizers `EncodeHtml` [37] and `GetTags` [14].

We found out that 86% of GA sanitizers (12 out of 14) along with `EncodeHtml` and `GetTags` are SLT-representable and were successfully learned by Σ^* (Table 1). In comparison, the authors of [37] had to invest several hours of a human expert's time into manual synthesis of each transducer from the source code. Only two sanitizers could not be inferred by Σ^*. `PrefixLine` is effectively a 0-SLT, but it is implemented by calling the `memchr` function in guards, which could only be represented by nondeterminism, as `memchr`'s return value could be arbitrarily far ahead from the start of the input string. `SnippetEscape` generates output from a predefined set of symbols only at the end of the input, but it could be learned if Σ^* were extended to subsequential transducers by generalizing Vilar's algorithm [58] to the symbolic setting.

7.2 Stream Filter Optimization

Stream programs often consist of many interacting filters that have to be optimized against various goals (e.g., speed, latency, throughput, ...). Two vital filter optimizations are reordering and fusion. For instance, reordering might be profitable when it would place a filter that is selective (i.e., drops some data) before a costly filter that does some processing. However, we must ensure that the two filters are commutative in order for reordering to be safe. Filter fusion is used when it would be profitable to trade pipeline parallelism for lower communication cost.

Benchmark	Linear	Stateless
Saxpy kernel [56]	+	+
Radix-2 complex FFT filter [43]	+	+
FIR filter [45]	+	+
Zig-zag descrambling filter in MPEG-2 [23]	+	+
Clsfr in L3-Switch bridge [18]	−	+
FM demodulator in StreamIt [54]	−	+
MPEG-2 internal parser in StreamIt [54]	−	−
GSM encoder/decoder in MiBench [33]	−	−

Table 2: Properties of Eight Filters Inferred by Σ^*. First four filters are linear and can be optimized with existing techniques. Σ^* enables fusion and reordering optimizations for all eight filters. By stateless, we mean single-state transducers.

We conducted a set of experiments to explore ability of Σ^* to enable stream filter optimizations such as reordering and fusion. Table 2 shows eight filters inferred by Σ^* indicating their two key properties: whether they encode a linear or non-linear output function, and whether they are stateless (i.e., have a single state) or stateful (i.e., have more then one state). First four filters can be optimized with existing techniques for optimizing linear filters [2, 45]. To our knowledge, no optimization techniques used in existing stream program compilers can optimize the remaining filters in Table 2. On the other hand, Σ^* enables fusion and reordering optimizations for all eight filters.

7.3 Parallelization

Finally, we show how SLT models of filters enable automated parallelization. We consider two approaches to improving utilization of hardware concurrency by mapping filters to (1) multiple cores and (2) SIMD instructions. Stateless filters are inherently data-parallel and lend themselves naturally to both multi-core and vectorized implementations by simple replication. Stateful filters normally cannot be parallelized this way as their next invocation depends on the previous invocation. However, SLT filters have usually small, a priori known, number of states, so we can run each replica of an SLT filter from each state in parallel and merge suitable outputs.

To provide some insight into possible practical speedups, we generated SIMD implementations using Intel SSE4 intrinsics and GPU implementations using NVIDIA CUDA 4.2 of 11 single-state SLTs (3 from Table 1, 6 from Table 2, and 2 hand-crafted) and one three-state SLT inferred by Σ^*. Original source code representations of all SLTs are stateful and some of them are with non-affine loop nests. For the stateless SLTs, we obtained speedups in the range of 1.72 to 3.89 for the SSE versions, and 1.5 to 2.7 for the CUDA versions. For the 3-state SLT, the speedup of the CUDA version was 2.3. CUDA versions were run using 1024 threads (resp. 1024 threads in 3 blocks) for the single-state (resp. three-state) SLTs. We established the baseline by running the sequential version of SLTs compiled with `gcc` with all optimizations (`-O3`) and SSE flags turned on.

8. Discussion

Predicate abstraction. To obtain complete models, Σ^* relies on convergence of the abstraction. One could rightfully ask why we need the whole learning machinery of Σ^* as we presume the abstraction eventually becoming equivalent (although, in a possibly non-deterministic form) to the behavioral model. Our program is deterministic, so as long as the abstraction is not single-valued it must contain infeasible behavior. Therefore, we could consider a procedure that just iteratively refines the abstraction until the abstraction becomes single-valued.

Setting aside how to efficiently refine abstractions alone, an abstraction-only approach suffers from a fundamental limitation: models generated by the abstraction can be unboundedly larger than the ones learned by Σ^*. For instance, consider the program from Figure 1 in [31], which is considered as an example on which predicate abstraction does not work well. Let us adapt the example by adding reading an input and writing some output in the loop, and enclosing it with an outer **while**(`true`)-loop to become a transductive program. Then Σ^* infers a single-state SLT, while by using predicate abstraction alone we would obtain an SLT with 1000 states.

SLT limitations. Along with the dependency on the abstraction, ability of Σ^* to infer complete models of program behavior is limited by the expressive power of symbolic conjectures. We have judiciously restricted SLTs so as to be able to represent interesting real-world examples while still being able to learn and equivalence check them efficiently. However, there are still other interesting classes of programs that are not SLT-representable. At the cost of additional complexity, we believe it would be possible to extend Σ^* to work with symbolic versions of more expressive transducers such as subsequential [58] and streaming transducers [3] and in this way enable learning of larger classes of programs.

Incomplete models. While the main concern of this paper is learning of exact behavioral models, incomplete models inferred by Σ^* could also be useful, e.g., for automated testing based on dynamic symbolic execution [16, 28, 49]. The learning component of Σ^* can be seen as systematically exploring all paths in the execution tree of a program, but in addition conjecturing the behavior of not yet explored branches. Conjectures could be used to guide the exploration strategy, similarly as in MACE [19], which showed how concrete transducer conjectures enable more effective testing of protocol implementations.

9. Other Related Work

Learning of symbolic transducers. Previous work on learning some notion of symbolic transducers [1, 12, 40] focuses on limited variations that a priori fix the shape of predicates and generate concrete values. Learning algorithms for these transducers postulate existence of an equivalence checking oracle and work over concrete alphabets, relying on the shape restrictions to translate the L^*-inferred concrete transducer to a symbolic one. As software artefacts feature predicates that are unknown prior to the analysis, if techniques [1, 12, 40] were to be applied in the software setting, they would need to exhaustively enumerate predicates, whose set is unbounded.

Learning in verification. Techniques for learning various types of automata have been applied in a wide variety of verification settings: inference of interface invariants [4, 27, 51], learning-guided concolic execution [19], compositional verification [21], and regular model checking [34]. In those settings, except for [27], learning is performed using concrete alphabets in a black-box manner, even though source code is usually available. Independently to us, Giannakopoulou et al. [27] have recently developed a technique for learning component interfaces that combines L^* with symbolic execution to discover method input guards. However, their approach treats method invocations as a single input step with no output, and, unlike Σ^*, assumes that each method has a finite number of paths. Under this later constraint, equivalence checking can be done by merely executing a sufficiently large number of membership queries. Alur and Černý's approach to synthesis of interface specifications with JIST [4] also uses predicate abstraction as Σ^*. However, predicate abstraction in JIST is used to check whether the synthesized interfaces are safe for a given safety property. In general, such interfaces are approximate and may incorporate infeasible behavior. In addition, the paper [4] does not demonstrate that the technique can be fully automated. In contrast, Σ^* is fully automated and infers faithful models upon termination.

Property checking. Although the goal of Σ^* is different, it resembles some ideas used in property checking. Most closely related, collecting predicates on conditional statements as in automated testing [16, 28, 49] has been previously combined with predicate abstraction in the SYNERGY algorithm [31]. While SYNERGY uses a combination of *must* and *may* analyses to check safety properties faster, Σ^* uses *must* (learning) and *may* (abstraction) to infer a more compact input-output relation.

Optimizations of filters. The closest work to our application of Σ^* for stream compiler optimizations is on optimizations of linear filters [45] and (slightly more general) linear state space systems [2] in StreamIt. In another line of work [47], fusion of filters for Brook has been developed using an affine model. Soulé et al. [52] developed a proof calculus that allows analysis of non-linear filters, but they can perform reordering optimizations only if filters are stateless. In contrast, Σ^* allows fusion and reordering of stateful filters with non-linear input-output relations. Approaches for numerical static analysis of linear filters (e.g., [24]) do not relate directly to ours as their goal is to obtain deep numerical properties of filters (such as numerical bounds) for safety-critical applications.

Automated parallelization. Thies et al. [55] extract pipeline partitions from legacy streaming applications written in C by relying on programmer-provided annotations indicating pipeline boundaries. In contrast, our approach does not require any annotations. For synchronous dataflow streaming applications, parallelism has been exploited either by replicating stateless filters [15, 29, 44], or by using affine partitioning of loop nests [47]. In contrast, Σ^* enables parallelization of non-linear stateful filters with non-affine loop bounds. There is a massive amount of work on general cross-loop iteration dependence scheduling (e.g., see [11]). Σ^* is aimed at complementing such advanced techniques by enabling naïve parallelization of behaviorally simple loop nests that are not necessary polyhedral, disjoint or with a clear pipeline structure.

Acknowledgments

This work was supported by the Gates trust, the Lawrence Livermore National Lab grant B597718, and by the NSERC PDF fellowship. Thanks to Mike Dodds, Eric Koskinen, Matthew Parkinson, Dawn Song, John Wickerson and the anonymous reviewers for comments and suggestions.

References

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proc. of the 22nd IFIP WG 6.1 Int. Conf. on Testing Software and Systems*, pages 188–204, 2010.

[2] S. Agrawal, W. Thies, and S. P. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Proc. of the 2005 Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 126–136, 2005.

[3] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 599–610, 2011.

[4] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 98–109, 2005.

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

[6] T. Ball. Formalizing counterexample-driven refinement with weakest preconditions. In *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 121–139. 2005.

[7] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[8] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, 2001.

[9] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. of the 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–172, 2002.

[10] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[11] M. M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 219–228, 2009.

[12] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *Proc. of the Theory and practice of software, 11th Int. Conf. on Fundamental approaches to software engineering*, pages 317–331, 2008.

[13] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Proc. of the 23rd Int. Conf. on Computer Aided Verification*, pages 184–190, 2011.

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

[15] I. Buck, T. Foley, D. R. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

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

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

[18] M. K. Chen, X.-F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 224–236, 2005.

[19] 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 *USENIX Security Symposium*, 2011.

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the 12th Int. Conf. on Computer Aided Verification*, pages 154–169, 2000.

[21] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619, pages 331–346, 2003.

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

[23] M. Drake, H. Hoffmann, R. M. Rabbah, and S. P. Amarasinghe. MPEG-2 decoding in a stream programming language. In *Proc. of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[24] J. Feret. Static analysis of digital filters. In *Programming Languages and Systems, 13th European Symposium on Programming*, pages 33–48, 2004.

[25] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the 19th Int. Conf. on Computer Aided Verification*, pages 519–531, 2007.

[26] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the System S declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134, 2008.

[27] D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In *19th Int. Symp. on Static Analysis*, pages 248–264, 2012.

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

[29] M. I. Gordon, W. Thies, and S. P. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of the 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[30] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Int. Conf. on Computer Aided Verification*, pages 72–83, 1997.

[31] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 117–127, 2006.

[32] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, 2008.

[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workload Characterization. WWC-4 IEEE Int. Workshop*, pages 3–14, 2001.

[34] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.*, 138:21–36, 2005.

[35] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. M. Rabbah. A computing origami: folding streams in FPGAs. In *Proc. of the 46th Design Automation Conference*, pages 282–287, 2009.

[36] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 58–70, 2002.

[37] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, 2011.

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

[39] A. Hormati, M. Kudlur, S. A. Mahlke, D. F. Bacon, and R. M. Rabbah. Optimus: efficient realization of streaming applications on FPGAs. In *Proc. of the 2008 Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 41–50, 2008.

[40] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Proc. of the 13th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 251–266, 2012.

[41] O. H. Ibarra. The unsolvability of the equivalence problem for ϵ-free NGSM's with unary input (output) alphabet and applications. In *Proc. of the 18th Annual Symp. on Foundations of Computer Science*, pages 74–81, 1977.

[42] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proc. of the 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 459–473, 2006.

[43] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The imagine stream processor. In *Proc. of the 20th Int. Conf. on Computer Design, VLSI in Computers and Processors*, pages 282–288, 2002.

[44] M. Kudlur and S. A. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 114–124, 2008.

[45] A. A. Lamb, W. Thies, and S. P. Amarasinghe. Linear analysis and optimization of stream programs. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 12–25, 2003.

[46] 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, 1996.

[47] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for Brook streaming applications on multiprocessors. In *Proc. of the 4th IEEE/ACM Int. Symp. on Code Generation and Optimization*, pages 196–207, 2006.

[48] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 50–61, 2010.

[49] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 263–272, 2005.

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

[51] R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *Proc. of the 22nd Int. Conf. on Computer Aided Verification*, pages 527–542, 2010.

[52] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *Programming Languages and Systems, 19th European Symposium on Programming*, pages 507–528, 2010.

[53] W. Thies and S. P. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of the 19th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–376, 2010.

[54] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[55] W. Thies, V. Chandrasekhar, and S. P. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 356–369, 2007.

[56] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proc. of the 7th Int. Symp. on Code Generation and Optimization*, pages 200–209, 2009.

[57] M. Veanes, D. Molnar, B. Livshits, and L. Litchev. Generating fast string manipulating code through transducer exploration and SIMD integration. Technical Report MSR-TR-2011–124, Microsoft Research, 2011.

[58] J. M. Vilar. Query learning of subsequential transducers. In *Proc. of the 3rd Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 72–83, 1996.