Hybrid Learning: Interface Generation through Static, Dynamic, and Symbolic Analysis

Falk Howar
Carnegie Mellon University,
Moffett Field, CA, USA
howar@cmu.edu

Dimitra Giannakopoulou
NASA Ames Research Center,
Moffett Field, CA, USA
dimitra.giannakopoulou
@nasa.gov

Zvonimir Rakamarić
School of Computing,
University of Utah, USA
zvonimir@cs.utah.edu

ABSTRACT

This paper addresses the problem of efficient generation of component interfaces through learning. Given a white-box component $\mathcal{C}$ with specified unsafe states, an interface captures safe orderings of invocations of $\mathcal{C}$'s public methods. In previous work we presented PSYCO, an interface generation framework that combines automata learning with symbolic component analysis: learning drives the framework in exploring different combinations of method invocations, and symbolic analysis computes method guards corresponding to constraints on the method parameters for safe execution. In this work we propose solutions to the two main bottlenecks of PSYCO. The explosion of method sequences that learning generates to validate its computed interfaces is reduced through partial order reduction resulting from a static analysis of the component. To avoid scalability issues associated with symbolic analysis, we propose novel algorithms that are primarily based on dynamic, concrete component execution, while resorting to symbolic analysis on a limited, as needed, basis. Dynamic execution enables the introduction of a concept of state matching, based on which our proposed approach detects, in some cases, that it has exhausted the exploration of all component behaviors. On the other hand, symbolic analysis is enhanced with symbolic summaries. Our new approach, X-PSYCO, has been implemented in the Java Pathfinder (JPF) software verification platform. We demonstrated the effectiveness of X-PSYCO on a number of realistic software components by generating more complete and precise interfaces than was previously possible.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specification—*Tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2483783

General Terms

Algorithms, Verification

Keywords

Static Analysis, Dynamic Analysis, Concolic Execution, Learning, Software Components, Interfaces

1. INTRODUCTION

Modern software components are open building blocks that are reused or connected dynamically to form larger systems. Well-defined component interfaces are key to successful component-based software engineering. This work focuses on "temporal" interfaces, which capture ordering relationships between invocations of component methods. For example, for the NASA Crew Exploration Vehicle (CEV) component used in our experiments in Sec. 5, an interface prescribes that a lunar lander cannot dock with a lunar orbiter without first jettisoning the launch abort subsystem. Temporal interfaces are well-suited for components that exhibit a protocol-like behavior; control-oriented components, such as NASA control software, device drivers, and web-services, often fall into this category.

In previous work [16], we presented PSYCO: a framework that implements a novel combination of automata learning with symbolic execution to automatically generate temporal interfaces for components that include methods with parameters. The generated interfaces are finite-state automata whose transitions are labeled with method names and guarded with constraints on the corresponding method parameters. The guards partition the input spaces of parameters, and enable a more precise characterization of legal orderings than was previously possible in a fully automatic fashion.

The performance bottleneck faced by PSYCO on larger examples stems from two main factors. In order to validate computed interface approximations, the learning framework generates equivalence queries consisting of sequences of method invocations of the target component. The number of sequences that each equivalence query generates grows exponentially in the length of these sequences. Moreover, to handle methods with parameters completely, each sequence is fully explored symbolically, thus further contributing to the cost of the approach. As a consequence, for complex components, such as the mentioned NASA CEV module, PSYCO was only able to guarantee correctness of the generated interface to a relatively small depth, within the budgeted running time of one hour [16].

This work proposes X-PSYCO, a hybrid "mutant" of the PSYCO algorithm that significantly improves performance through a novel combination of learning with static, concrete, and symbolic analysis. Although we like to refer to X-PSYCO as a mutant of PSYCO, it is in fact a new framework, with novel algorithms for performing most of the work at a concrete level, while using symbolic techniques lazily to guarantee completeness. The explosion of sequences in equivalence queries that the learning framework generates is reduced through partial order reduction (POR) resulting from an adequate static analysis of the component. Static analysis determines groups of component methods that are mutually independent. This information is used on-the-fly, to always generate a single order for any two or more consecutive independent methods during sequence construction; no alternative orderings are generated or explored, often achieving significant savings in the number of sequence-forming method combinations.

To avoid scalability issues associated with symbolic analysis, we primarily rely on concrete component execution for interface generation, while resorting to symbolic analysis on a limited, as needed, basis. Concrete execution enables the introduction of a concept of state matching, based on which our proposed approach detects, in some cases, that it has exhausted the exploration of all component behaviors. On the other hand, symbolic analysis is enhanced with symbolic summaries. Our experiments demonstrate that the various components of our novel approach complement each other in improving the performance of interface generation. More precisely, we are able to guarantee the correctness of generated interfaces to a larger depth than could be achieved previously. As a result, we are able to generate *more precise* interfaces for some of the examples. Moreover, in other cases, state-matching enables us to terminate with the guarantee that the interface generated is *correct to any depth*.

We implemented X-PSYCO within the Java Pathfinder (JPF) software verification toolset [23] for the Java virtual machine. JPF is an open-source project developed at the NASA Ames Research Center. X-PSYCO is a new extension of JPF, and has few commonalities with the original PSYCO, which is also available as a separate JPF extension. We have applied X-PSYCO to all the benchmark problems that we also applied PSYCO to in our previous work [16], as well as to some new components in the domain of embedded software and network protocols. We demonstrate that X-PSYCO significantly outperforms PSYCO in terms of precision of the generated interfaces.

2. MOTIVATING EXAMPLE

Our running example is the *PipedOutputStream* class taken from the *java.io* package. Similar to previous work [3, 30], we removed unnecessary details from the example. In particular, we model objects and object references as primitive data types since our implementation currently does not support composite data types. (Hence, note that currently we are not dealing with orthogonal, well-known issues related to creating objects on the heap, such as aliasing.) Fig. 1 gives the simplified code. The example has a private field *sink* of integer type, a private field *sinkConnected* of boolean type, and four public methods called *connect*, *write*, *flush*, and *close*. Throwing exceptions is modeled by asserting *false*, denoting an undesirable error state.

```
class PipedOutputStream {
  private int sink = 0;
  private boolean sinkConnected = false;

  public void connect(int snk, boolean snkConnected) {
    if (snk == 0) {
      assert false;
    } else if (sink != 0 || snkConnected) {
      assert false;
    }
    sink = snk;
    sinkConnected = true;
  }

  public void write() {
    if (sink == 0) {
      assert false;
    } else { ; }
  }

  public void flush() {
    if (sink != 0) { ; }
  }

  public void close() {
    if (sink != 0) { ; }
  }
}
```

Figure 1: Motivating example.

The class initializes field *sink* to 0 and field *sinkConnected* to *false*. Method *connect* takes a parameter *snk* of integer type and a parameter *snkConnected* of boolean type. It goes to an error state (i.e., throws an exception) either if *snk* is 0 or if a stream has already been connected; otherwise, it connects the input and output streams. Method *write* can be called only if *sink* is not 0, and otherwise an error state is reached. Methods *flush* and *close* have no effect, i.e., they do not throw an exception.

The lower part of Fig. 2 illustrates the interface generated with our approach, implemented in PSYCO, for this example. The interface captures the fact that *flush* and *close* can be invoked unconditionally, whereas *write* can only occur after a successful invocation of *connect*. The guard $snk \neq 0 \wedge \neg snkConnected$, over the parameters *snk* and *snkConnected* of the method *connect*, captures the condition for a successful connection. PSYCO was the first approach to combine the automatic generation of interfaces with the computation of symbolic method guards. Without support for guards in our component interfaces, we would obtain the more imprecise interface shown at the top of the figure, which precludes several legal sequences of method invocations. For example, method *connect* is blocked from the interface since it cannot be called unconditionally. As a consequence, *write* is blocked as well. Therefore, support for automatically generating guards enables PSYCO to produce richer and more precise component interfaces for components that have methods with parameters.

However, this capability comes at a cost. The learning-based framework that PSYCO implements potentially generates an exponentially large number of queries, where each query is a sequence of method invocations. Each sequence is explored symbolically over parameters of all methods. The large number of such symbolic executions makes the performance of PSYCO suffer for components that include sev-

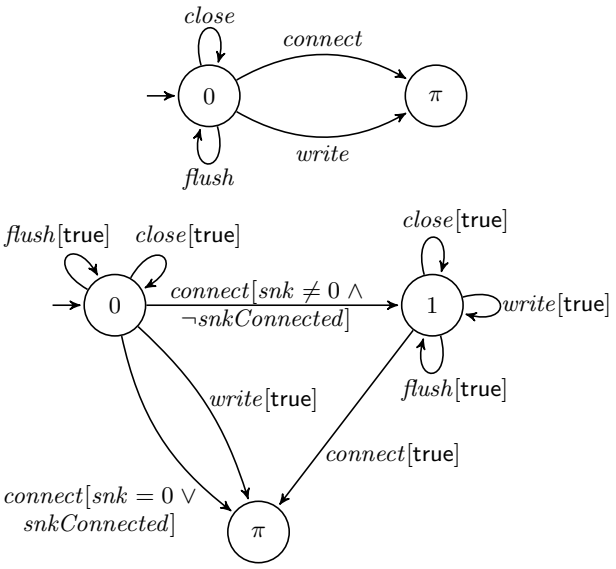

Figure 2: Interfaces for our motivating example. Above, there is no support for guards, while below Psyco is used to generate guards. Initial states are marked with arrows that have no origin; error states are marked with π. Edges are labeled with method names (with guards, when applicable).

eral methods with parameters. For example, for our largest previous benchmark, namely CEV, PSYCO only managed to guarantee the correctness of the generated interface to depth 3 within the time-budget of one hour [16].

X-PSYCO addresses these performance bottlenecks by introducing several new concepts. First of all, X-PSYCO performs as much of the learning as possible using concrete execution, and only reverts to symbolic execution when necessary. More precisely, our approach uses pools of concrete method invocations to explore different behaviors of a component. For instance, coming back to our motivating example, when exploring sequences that involve method *connect*, our algorithms explore the following concrete invocations: $connect(0, false)$, $connect(1, false)$, and $connect(1, true)$. As our experiments later will show, concrete execution is significantly faster than symbolic analysis. Symbolic analysis in X-PSYCO is used in computing method guards, and in ensuring completeness of the approach; in other words, it generates additional concrete values to ensure exploration of all paths up to some length.

Moreover, X-PSYCO introduces a notion of partial order reduction. In Fig. 1, we can observe that methods *write*, *flush*, and *close* do not touch field *sinkConnected*, and only read field *sink* but never write to it. These methods are therefore independent from each other, and their relative consecutive ordering does not affect the outcome of a sequence. In other words, performing sequence *connect*, *write*, *flush*, *close* is equivalent to performing *connect*, *flush*, *write*, *close*; if one sequence leads to an error, then so does the other, and the same holds for non-errors. We therefore propose to apply an appropriate static analysis as a preprocessing step in order to automatically and conservatively identify such independent methods. During sequence generation, only a single order of such methods is explored when they are executed consecutively.

To conclude, the described optimizations we introduced in X-PSYCO enable us to explore our running example to depth 54, as opposed to depth 8 with our original PSYCO (see Table 1, column X-PSYCO with POR+Matching, and Table 2).

3. PRELIMINARIES

This section presents material required for understanding our X-PSYCO hybrid learning algorithm described in Sec. 4.

3.1 Background

Labeled Transition Systems (LTS). We use deterministic LTSs to express temporal component interfaces. Symbols π and υ denote a special *error* and *unknown* state, respectively. The former models unsafe states and the latter captures the lack of knowledge about whether a state is safe or unsafe. States π and υ have no outgoing transitions.

A deterministic LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$, where Q is a finite non-empty set of states, αM is a set of observable actions called the *alphabet* of M, $\delta : (Q \times \alpha M) \mapsto Q$ is a transition function, and $q_0 \in Q$ is the initial state. LTS M is complete if each state except π and υ has an outgoing transition for every action in αM.

A *trace*, also called an *execution* or *word*, of an LTS M is a finite sequence of observable actions that label the transitions that M can perform starting from q_0. A trace is illegal (resp., unknown) if it leads M to state π (resp., υ). Otherwise, it is legal. The illegal (resp., unknown, legal) language of M, denoted as $\mathcal{L}_{illegal}(M)$ (resp., $\mathcal{L}_{unknown}(M)$, $\mathcal{L}_{legal}(M)$), is the set of illegal (resp., unknown, legal) traces of M.

Automata Learning with L*. Our work uses an adaptation [31] of the classic L* learning algorithm [6, 28], to learn LTSs over some alphabet αM. In our setting, learning is based on partitioning the words over αM into three unknown regular languages L_1, L_2, and L_3, with L* using this partition to infer an LTS that is consistent with the partition. To infer an LTS, L* interacts with a teacher that answers two types of questions. The first type is a *membership query* that takes as input a string $w \in \alpha M^*$ and answers *true* if $w \in L_1$, *false* if $w \in L_2$, and *unknown* otherwise. The second type is an *equivalence query* or *conjecture* , which determines, given a candidate LTS M, whether the following conditions hold: $\mathcal{L}_{legal}(M) = L_1$, $\mathcal{L}_{illegal}(M) = L_2$, and $\mathcal{L}_{unknown}(M) = L_3$. If they hold of the candidate M, then the teacher answers *true*, at which point L* has achieved its goal and returns M. Otherwise, the teacher returns a counterexample, which is a string w that invalidates one of the above conditions. The counterexample is used by L* to drive a new round of membership queries in order to produce a new, refined candidate. Each candidate M that L* constructs is smallest, meaning that any other LTS consistent with the information provided to L* up to that stage has at least as many states as M. Given a correct teacher, L* is guaranteed to terminate with a minimal (in terms of the number of states) LTS for L_1, L_2, and L_3.

Symbolic Execution. Symbolic execution is a program analysis technique for systematically exploring a large number of program execution paths [12, 24]. It uses symbolic

$$
\begin{aligned}
\mathit{Component} &::= \textbf{class}\ \mathit{Ident}\ \{\ \mathit{Global}^*\ \mathit{Method}^+\ \}\\
\mathit{Method} &::= \mathit{Ident}\ (\mathit{Parameters})\ \{\ \mathit{Stmt}\ \}\\
\mathit{Global} &::= \mathit{Type}\ \mathit{Ident};\\
\mathit{Parameters} &::= \epsilon \mid \mathit{Pararameter}\ (,\ \mathit{Parameter})^*\\
\mathit{Parameter} &::= \mathit{Type}\ \mathit{Ident}\\
\mathit{Stmt} &::= \mathit{Stmt};\ \mathit{Stmt}\\
&\quad \mid \mathit{Ident} = \mathit{Expr}\\
&\quad \mid \textbf{assert}\ \mathit{Expr}\\
&\quad \mid \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt}\\
&\quad \mid \textbf{while}\ \mathit{Expr}\ \textbf{do}\ \mathit{Stmt}\\
&\quad \mid \textbf{return}
\end{aligned}
$$

Figure 3: Component grammar. *Ident*, *Expr*, **and** *Type* **have the usual meaning.**

values as program inputs in place of concrete (actual) values. The resulting output values are then statically computed as symbolic expressions over symbolic input values and constants, using a specified set of operators. A symbolic execution tree, or constraints tree, characterizes all program execution paths explored during symbolic execution. Each node in the tree represents a symbolic state of the program, and each edge represents a transition between two states. A symbolic state consists of a unique program location identifier, symbolic expressions for the program variables currently in scope, and a path condition defining conditions (i.e., constraints) that have to be satisfied in order for the execution path to this state to be taken. The path condition describing the current path through the program is maintained during symbolic execution by collecting constraints when conditional statements are encountered. Path feasibility is established using a constraint solver to check satisfiability of the corresponding path condition.

3.2 Components

Components and Methods. A *component* is defined by the grammar in Fig. 3. A component $\mathcal{C}$ has a set of global variables (also referred to as fields or component variables) representing its internal state, and a set of one or more methods. We assume methods have no recursion. (Note that this is a common assumption since handling recursion in symbolic techniques is a well-known issue orthogonal to this work.) A component method is either a public interface method or a private method. For simplicity of exposition, we assume that public interface methods may call only private methods. Furthermore, we assume all method calls are inlined, though in our implementation we handle method calls and returns, and unroll recursion to a bounded depth. We also assume the usual statement semantics. Let Ids be the set of component method identifiers (i.e., names), Stmts the set of all component statements, and Prms the set of all input parameters of component methods. We define the signature Sig_m of a method m as a pair $(Id_m, P_m) \in \mathsf{Ids} \times \mathsf{Prms}^*$; we write $Id_m(P_m)$ for the signature Sig_m of the method m. A *method* m is then defined as a pair (Sig_m, s_m) where $s_m \in \mathsf{Stmts}$ is its top-level statement.

Let $\mathcal{M}$ be the set of methods in a component $\mathcal{C}$ and $\mathcal{G}$ be the set of its global variables. For every method $m \in \mathcal{M}$,

$$
\xi_{connect} = \begin{cases}
(\mathsf{error},\ (snk = 0))\\
(\mathsf{error},\ (snk \neq 0) \wedge (sink \neq 0))\\
(\mathsf{error},\ (snk \neq 0) \wedge (sink = 0) \wedge (snkConnected))\\
(\mathsf{ok},\ (snk \neq 0) \wedge (sink = 0) \wedge (\neg snkConnected)\\
\qquad \wedge (sink' = snk) \wedge (sinkConnected')\ )
\end{cases}
$$

Figure 4: Symbolic summary for method *connect* **of** ***PipedOutputStream.***

each parameter $p \in P_m$ takes values from a domain D_p based on its type; similarly for global variables. We expect that all method parameters are of basic types. Given a method $m \in \mathcal{M}$, an execution $\theta \in \mathsf{Stmts}^*$ of m is a finite sequence of visited statements $s_1, \ldots, s_n$ where s_1 is the top-level method statement s_m. The set $\Theta_m \in 2^{\mathsf{Stmts}^*}$ is the set of all unique executions of m. We assume that each execution $\theta \in \Theta_m$ of a method visits a bounded number of statements (i.e., $|\theta|$ is bounded), and also that the number of unique executions is bounded (i.e., $|\Theta_m|$ is bounded); in other words, the methods have no unbounded loops. (Similar to recursion, this is a common assumption and a well-known issue orthogonal to this work.) A *valuation* over P_m, denoted $\vec{P}_m$, is a mapping that assigns to each parameter $p \in P_m$ a value in D_p. We denote a valuation over global variables in $\mathcal{G}$ with $\vec{G}$. We define $\vec{G}_0$ as the valuation representing the initial values of all global variables. Given valuations $\vec{P}_m$ and $\vec{G}$, we assume that the execution of m visits exactly the same sequence of statements; in other words, the methods are deterministic.

Symbolic Expressions. We interpret all the parameters of methods symbolically, and use the name of each parameter as its symbolic name; with a slight abuse of notation, we take Prms to also denote the set of symbolic names. A *symbolic expression* e is defined as follows:

$$e\ ::=\ c\ \mid\ p\ \mid\ (e \circ e),$$

where $c \in C$ is a constant, $p \in \mathsf{Prms}$ a parameter, and $\circ \in \{+, -, *, /, \%\}$ an arithmetic operator. The set of constants C includes constants used in component statements and the initial values of component state variables in $\vec{G}_0$.

Constraints. We define a *constraint* φ as follows:

$$\varphi\ ::=\ \mathit{true}\ \mid\ \mathit{false}\ \mid\ e \oplus e\ \mid\ \varphi \wedge \varphi\ \mid\ \varphi \vee \varphi,$$

where $\oplus \in \{<, >, =, \leq, \geq\}$ is a comparison operator.

Method Invocations and Paths. For a component $\mathcal{C}$ with global variables $\mathcal{G}$ and a method signature $Sig_m = (Id_m, P_m)$, we use the following three concepts throughout the paper.

A *concrete method invocation* is a pair $\rho_c = (Sig_m, \vec{P}_m)$ consisting of a method signature and a valuation of P_m.

A *guarded method invocation* is a pair $\rho_g = (Sig_m, \gamma)$ consisting of a method signature and a symbolic expression γ over parameters in P_m and constants. As discussed later in the paper, a guarded method invocation ρ_g represents all concrete invocations that satisfy its constraint γ.

A *symbolic method path* is a pair $\rho_s = (s, \varphi_m)$ consisting of a *path state* $s \in \{\mathsf{ok}, \mathsf{error}, \mathsf{unknown}\}$ and a *path constraint* φ_m. Path state s is the resulting state of the component for this path through m, and φ_m is an extended symbolic expression over variables from $\mathcal{G}$ and their primed versions, P_m, and constants. We will produce such paths by concolic execution of methods. The path state s reflects the result of the symbolic analysis. Execution of error-paths always leads to the violation of an assertion, while ok-paths execute successfully. The unknown state represents the fact that the symbolic analysis was not able to explore beyond the point reached by this path. This can happen, for example, due to limitations of the underlying constraint solver. The path constraint φ_m captures the pre- and post-conditions of a method path in a symbolic expression. We use primed names in φ_m when referring to updated component variables (i.e., after the execution of ρ_s).

By $\psi[\vec{P}_m/P_m]$ we denote the (not necessarily symbolic) expression that results from replacing every occurrence of some $p \in P_m$ in the symbolic expression ψ by the value $\vec{P}_m(p)$. We write $\vec{G}/\mathcal{G}'$ to indicate that we replace the primed occurrences of variables x from $\mathcal{G}$ by their values $\vec{G}(x)$ in an expression.

Then, for a concrete method invocation $\rho_c = (Sig_m, \vec{P}_m)$, valuations $\vec{G}_1$ and $\vec{G}_2$, and a symbolic method path $\rho_s = (s, \varphi_m)$ we say that $(\vec{G}_1, \rho_c, \rho_s, \vec{G}_2)$ is a *step* of $\mathcal{C}$ if the expression $\varphi_m[\vec{P}_m/P_m][\vec{G}_1/\mathcal{G}][\vec{G}_2/\mathcal{G}']$ evaluates to true. Intuitively, we replace all symbolic names in φ_m with concrete values from the three valuations and check if the expression evaluates to true, meaning that there could exist an execution of $\mathcal{C}$ during which the concrete invocation ρ_c triggers path ρ_s and changes the state of $\mathcal{C}$ from $\vec{G}_1$ to $\vec{G}_2$.

We say that a sequence $\sigma_c = \rho_{c1}, \ldots, \rho_{ck}$ of concrete method invocations *satisfies* a sequence $\sigma_s = \rho_{s1}, \ldots, \rho_{sk}$ of symbolic method paths from symbolic summaries (see below) of $\mathcal{C}$ if there is a sequence of valuations $\vec{G}_0, \ldots, \vec{G}_k$ such that $(\vec{G}_{i-1}, \rho_{ci}, \rho_{si}, \vec{G}_i)$ with $1 \leq i \leq k$ are steps of $\mathcal{C}$. (Of course, we require that the state of all ρ_{si} with $i < k$ in σ_s is ok.) We then write $\sigma_c \models \sigma_s$ and refer to σ_c as a *concrete execution* of $\mathcal{C}$. The *result* $eval(\sigma_c)$ of the concrete execution σ_c is the state of ρ_{sk}. Accordingly, we refer to σ_s as a *symbolic execution* of $\mathcal{C}$ and let $eval(\sigma_s)$ be the state of ρ_{sk}.

Finally, for a concrete method invocation $\rho_c = (Sig_m, \vec{P}_m)$ and a guarded method invocation $\rho_g = (Sig_m, \gamma)$, we say that ρ_c *satisfies* ρ_g, denoted by $\rho_c \models \rho_g$, if $\gamma[\vec{P}_m/P_m]$ evaluates to true. We extend this definition to concrete executions σ_c and sequences of guarded methods (denoted by σ_g).

Symbolic Summaries. For a method m of a component $\mathcal{C}$, the *symbolic summary* ξ_m of m is a concise symbolic representation of the behavior of m. We obtain summaries of component methods using symbolic execution. Formally, let ξ_m be the set of symbolic method paths of m. Fig. 4 shows the four method paths for the *connect* method of the *PipedOutputStream* component from Fig. 1.

3.3 Symbolic Interfaces

Similar to PSYCO, we use interface LTSs, or iLTS, to describe temporal component interfaces with guards on method parameters. An iLTS is a tuple $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$, where $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is a deterministic and complete LTS, $\mathcal{S}$ a set of method signatures, Γ a set of guards for method signatures in $\mathcal{S}$, and $\Delta_G : \alpha M \mapsto \mathcal{S} \times \Gamma$ a function that maps each $\mathtt{a} \in \alpha M$ into a method signature $Sig_m \in \mathcal{S}$ and a guard $\gamma_m \in \Gamma$. In addition, the mapping Δ_G is such that the set of all guards for a given method signature forms a partition of the input space of the corresponding method. More formally, let $\Gamma_m = \{\gamma \mid \exists \mathtt{a} \in \alpha M \,.\, \Delta_G(\mathtt{a}) = (Sig_m, \gamma)\}$ be the set of guards belonging to a method signature Sig_m. Then, the guards for a method (1) are non-overlapping: $\forall \mathtt{a}, \mathtt{b} \in \alpha M, \gamma_\mathtt{a}, \gamma_\mathtt{b} \in \Gamma, Sig_m \in \mathcal{S} \,.\, (\mathtt{a} \neq \mathtt{b} \wedge \Delta_G(\mathtt{a}) = (Sig_m, \gamma_\mathtt{a}) \wedge \Delta_G(\mathtt{b}) = (Sig_m, \gamma_\mathtt{b})) \Rightarrow \neg\gamma_\mathtt{a} \vee \neg\gamma_\mathtt{b}$, (2) cover all of the input space: $\forall Sig_m \in \mathcal{S} \,.\, \bigvee_{\gamma \in \Gamma_m} \gamma$, and (3) are non-empty.

Given an iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$, an execution of A is a sequence of guarded method invocations $\sigma_g = \rho_{g1}, \ldots, \rho_{gk}$. Every such sequence σ_g has a corresponding trace $w = \mathtt{a}_1, \ldots, \mathtt{a}_k$ in M such that for $1 \leq i \leq k$, $\Delta_G(\mathtt{a}_i) = \rho_{gi}$. Moreover, σ_g represents all concrete executions σ_c where $\sigma_c \models \sigma_g$. Then a sequence of guarded method invocations σ_g is a legal (resp., illegal, unknown) execution in A if its corresponding trace in M is legal (resp., illegal, unknown). Based on this distinction, we define the languages $\mathcal{L}_{legal}(A)$, $\mathcal{L}_{illegal}(A)$, and $\mathcal{L}_{unknown}(A)$ as the sets of legal, illegal, and unknown executions of A, respectively.

An iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$ is an interface for a component $\mathcal{C}$ if $\mathcal{S}$ is a subset of method signatures of the methods $\mathcal{M}$ in $\mathcal{C}$. However, not all such interfaces are acceptable and a notion of interface correctness also needs to be introduced. Traditionally, correctness of an interface for a component $\mathcal{C}$ is associated with two characteristics: *safety* and *permissiveness*. A safe interface blocks all erroneous executions of $\mathcal{C}$; a permissive interface allows all good executions (i.e., executions that do not lead to an error) of $\mathcal{C}$. A *full* interface is then an interface that is both safe and permissive [20].

We extend this definition to iLTSs as follows. Let iLTS A be an interface for a component $\mathcal{C}$. We say a concrete execution σ_c of a component is illegal if it results in an assertion violation; otherwise, the execution is legal. Then, A is a *safe* interface for $\mathcal{C}$ if for every execution $\sigma_g \in \mathcal{L}_{legal}(A)$, we determine that all the corresponding concrete executions σ_c of component $\mathcal{C}$ (i.e., the ones with $\sigma_c \models \sigma_g$) are legal. It is *permissive* if for every execution $\sigma_g \in \mathcal{L}_{illegal}(A)$, we determine that all the corresponding concrete executions σ_c of component $\mathcal{C}$ are illegal.

Finally, A is *tight* if for every execution $\sigma_g \in \mathcal{L}_{unknown}(A)$, we cannot determine whether the corresponding concrete executions σ_c of component $\mathcal{C}$ are legal or illegal (i.e., if $eval(\sigma_c) = \mathsf{unknown}$); this explicitly captures possible incompleteness of the underlying analysis techniques. We say A is *full* if it is safe, permissive, and tight. Moreover, we say A is *k-full* for some $k \in \mathbb{N}$ if it is safe, permissive, and tight for all method sequences of length up to k.

For the remainder of the paper we will use a mapping $[\![A]\!]$ from concrete executions to $\{\mathsf{ok}, \mathsf{error}, \mathsf{unknown}\}$, where $[\![A]\!](\sigma_c) = \mathsf{ok}$ (resp., error, unknown) if σ_g with $\sigma_c \models \sigma_g$ is in $\mathcal{L}_{legal}(A)$ (resp., $\mathcal{L}_{illegal}(A)$, $\mathcal{L}_{unknown}(A)$). This shorthand will allow us to compare the results of concrete executions on A and $\mathcal{C}$ directly.

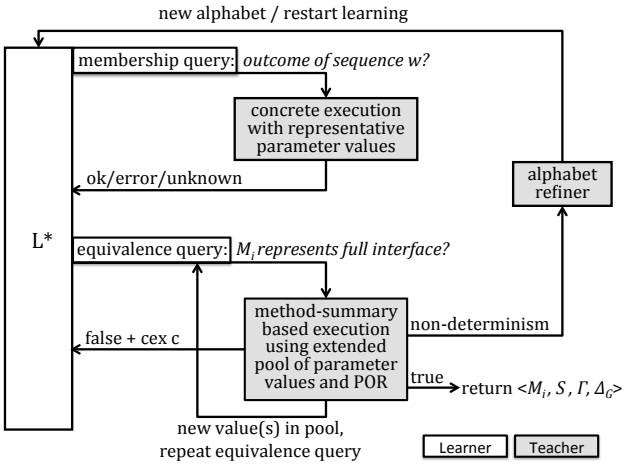

Figure 5: X-Psyco framework during iteration i of learning algorithm.

4. HYBRID INTERFACE LEARNING

Let $\mathcal{C}$ be a component and $\mathcal{S}$ the set of signatures of the methods $\mathcal{M}$ in $\mathcal{C}$. Our goal is to automatically compute an interface for $\mathcal{C}$ as an iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$. We achieve this through a combination of L* to generate LTS M, and symbolic execution to compute the set of guards Γ and the mapping Δ_G.

X-Psyco aims at reducing the use of expensive symbolic execution as much as possible by using concrete execution instead. In order to achieve this, our algorithm maintains a symbolic representation for every $\mathtt{a} \in \alpha M$, but also two pools of corresponding concrete method invocations, which the teacher of the L* algorithm uses to answer membership and equivalence queries.

The teacher uses the smaller *membership-pool* to answer membership queries. This pool contains exactly one concrete method invocation ρ_c for each guarded invocation ρ_g in the image of Δ_G. This pool reflects the current partitioning of methods in the conjectured iLTS.

The larger *equivalence-pool*, on the other hand, contains one concrete invocation for each symbolic method path ρ_s explored so far. The equivalence-pool represents the current knowledge of the teacher about symbolic paths of $\mathcal{C}$. As its name suggests, it is used by the teacher to process equivalence queries.

Both pools are initialized and extended automatically during the learning process. They are extended whenever the used method partitioning changes or when the teacher discovers new feasible symbolic paths of the component. We use I to denote the equivalence-pool of all concrete invocations that X-Psyco maintains, and the membership-pool function $\Delta_C : \alpha M \to I$ to characterize the selection of representative invocations. The image of Δ_C constitutes the membership-pool.

At a high level, our proposed framework operates as follows (see Fig. 5 and Alg. 1). Initially, X-Psyco executes each method symbolically using the initial values of globals, and partitions its parameters to differentiate between different possible outcomes (i.e., legal, illegal, unknown). Symbolic descriptions of the partitions are used as initial guards Γ in Δ_G (lines 2–3). Moreover, it produces concrete parameter values (i.e., concrete method invocations) to represent each partition. These values are used to initialize the membership-pool Δ_C (cf. Alg. 5). It then produces concrete parameter values for all symbolic paths that are enabled from the initial valuation $\vec{G}_0$. These values (a superset of the membership-pool) make up the equivalence-pool I (lines 4–6).

For our motivating example (see Fig. 1), we start with the alphabet $\alpha M = \{close, flush, connect_1, connect_2, write\}$, set of signatures $\mathcal{S} = \{connect(snk, snkConnected), close(), flush(), write()\}$, and Δ_G such that

$$\begin{aligned}
&\Delta_G(close) = (close(), true),\\
&\Delta_G(flush) = (flush(), true),\\
&\Delta_G(connect_1) = (connect(snk, snkConnected),\\
&\qquad\qquad snk \neq 0 \wedge \neg snkConnected),\\
&\Delta_G(connect_2) = (connect(snk, snkConnected),\\
&\qquad\qquad snk = 0 \vee snkConnected),\\
&\Delta_G(write) = (write(), true).
\end{aligned}$$

Note that method *connect* is partitioned during the initial phase of our algorithm into $connect_1$ and $connect_2$, where $connect_1$ (resp., $connect_2$) represent all concrete invocations of the *connect* method that are legal (resp., illegal), given initial values $\vec{G}_0$ of the component globals. The concrete invocations in the membership-pool are $connect_1(1, false)$ and $connect_2(1, true)$, corresponding to the two last paths in Fig. 4. The additional concrete invocation $connect_2(0, false)$ is generated for the equivalence-pool. It corresponds to the first path in Fig. 4. The second path from the figure is not enabled from $\vec{G}_0$ since $sink = 0$. Note that if our algorithms at some point reach a valuation $\vec{G}_i$ for which this path becomes enabled, we will add concrete invocations for the path to the equivalence-pool.

After generating these initial pools of concrete invocations, L* is used to learn an LTS over the alphabet αM that corresponds to the partitioned methods of $\mathcal{C}$ (line 7). The algorithm is executed (alternating membership and equivalence queries) until a termination criterion is met (e.g., a fixed time limit as in our benchmarks) and the most recent conjectured iLTS A is returned as the conjectured interface of the component.

Partial order reduction (cf. Sec. 4.2) enables the generation of fewer sequences during equivalence queries, as already mentioned in Sec. 2. The second optimization that we use during equivalence queries is state matching. This allows us to reduce the number of tests in general, and in some cases even puts a finite bound on the number of tests to be performed (cf. Sec. 4.1). The teacher and the algorithms invoked by top-level Alg. 1 are described in detail in the next section.

4.1 Concolic Teacher

As described at a high level in the previous section, the concolic teacher answers membership and equivalence queries for the L* algorithm using two pools of concrete method invocations. In this section we describe how the teacher answers such queries.

A membership query over the alphabet αM is a sequence $w = \mathtt{a}_1, \ldots, \mathtt{a}_n$ such that for $1 \leq i \leq n$, $\mathtt{a}_i \in \alpha M$. It inquires about the outcome of the corresponding sequence of guarded invocations $\sigma_g = \Delta_G(\mathtt{a}_1), \ldots, \Delta_G(\mathtt{a}_n)$. To answer

Algorithm 1 Learning an iLTS for a component.

Input: A set of method signatures $\mathcal{S}$ of a component $\mathcal{C}$
Output: An iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$

```
1: Γ ← {true}, αM ← ∅, Δ_G ← ∅, I ← ∅, Δ_C ← ∅
2: for all Sig_m ∈ S do
3:     RefineAlphabet(ϵ, (Sig_m, true), ϵ)                (Alg. 4)
4: for all ρ_s of all method summaries do
5:     if ∃ρ_c . ρ_c ⊨ ρ_s then
6:         I ← I ∪ {ρ_c}
7: M ← LearnAutomaton()                                   (Fig. 5)
8: return A = ⟨M, S, Γ, Δ_G⟩
```

this query, our teacher computes the result (ok, error, or unknown) for the corresponding sequence of *concrete* method invocations $\sigma_c = \Delta_C(\mathtt{a}_1), \ldots, \Delta_C(\mathtt{a}_n)$ from the membership-pool. While learning with concrete sequences is much faster than evaluating membership queries symbolically, it comes with one caveat: our concrete representatives may not cover all the possible outcomes of the guarded method sequence. However, we detect this case during equivalence queries, which results in refinement of method guards, as described later in this section.

Conceptually, an equivalence query checks whether a conjectured iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$ is full (i.e., safe, permissive, and tight). However, conjectured interfaces have unbounded loops, and symbolic techniques typically handle such loops through bounded unrolling. For this reason, similarly to PSYCO, X-PSYCO checks whether A is k-full by reducing equivalence queries to membership queries of bounded depth. The generated membership queries in this context are checked using the equivalence-pool, as discussed below.

To check whether the iLTS A is k-full, we need to verify that for every trace w of length $\leq k$ in its LTS M, if w is in $\mathcal{L}_{legal}(M)$, $\mathcal{L}_{illegal}(M)$, or $\mathcal{L}_{unknown}(M)$, then the corresponding symbolic sequence $\Delta_G(w)$ of the component leads to ok, error, or unknown, respectively. As discussed later in this section, we achieve this by gradually incrementing up to the bound k the length of checked sequences (although for simplicity our algorithms do not show k).

Concolic Equivalence Queries. As mentioned, an equivalence query reduces to a number of membership queries used to check all sequences of length $\leq k$. Each membership query is exercised using the equivalence-pool. Specifically, for a sequence $w = \mathtt{a}_1, \ldots, \mathtt{a}_n$, the teacher computes at least one concrete execution σ_c for every symbolic path sequence σ_s within the sequence $\sigma_g = \Delta_G(\mathtt{a}_1), \ldots, \Delta_G(\mathtt{a}_n)$. To create the required concrete sequences σ_c, the teacher uses existing values from the equivalence-pool if possible, and extends the pool using a constraint solver only when necessary.

The result of each concrete execution is compared to the result expected by the current conjectured iLTS. In case of a mismatch, σ_c signifies a counterexample, at which point the outcome of σ_c is compared to the outcome of the concrete sequence constructed for w by using the representative values from the membership-pool. If the two agree, then a counterexample is returned to L* to refine the conjectured iLTS.

However, if the two disagree, this signifies to the teacher the need for refinement, due to non-determinism of the current partitions (see Fig. 5). The teacher then refines Γ and Δ_G using symbolic execution on a program derived from w. After refining the alphabet, L* restarts. Note that, in this fashion, all cases of non-determinism of the current partitions are guaranteed to trigger refinement at some point, because when concrete sequences have different results, then at least one of them will disagree with the expected result from the conjectured iLTS.

Algorithm 2 Concolic equivalence query.

Input: An iLTS $A = \langle M, \mathcal{S}, \Gamma, \Delta_G \rangle$, concrete invocations I, a set $\mathcal{S}$ of method signatures
Output: A counterexample or an alphabet refinement

```
1: Q ← {ϵ}, V ← ∅
2: while Q ≠ ∅ do
3:   σ_s ← dequeue(Q)
4:   for all ρ_g in the image of Δ_G do
5:     for all ρ_s within ρ_g do
6:       σ'_s ← σ_s · ρ_s
7:       if not skip σ'_s due to POR then
8:         R ← {σ_c ∈ I* | σ_c ⊨ σ'_s}
9:         if R = ∅ then
10:          if SAT(σ'_s) then
11:            Let σ_c^new such that σ_c^new ⊨ σ'_s
12:            I ← I ∪ {ρ_c ∈ σ_c^new}
13:            goto line 1
14:        else
15:          if ∃σ_c ∈ R . [[A]](σ_c) ≠ eval(σ_c) then
16:            AnalyzeCounterexample(σ_c)                (Alg. 3)
17:          else if eval(σ'_s) = ok then
18:            v ← {G⃗ | G⃗_0 --σ_c--> G⃗ for σ_c ∈ R}
19:            if v not subsumed by state in V then
20:              V ← V ∪ {v}
21:              Q.enqueue(σ'_s)
```

Alg. 2 shows the technical details. We organize the exploration as a breadth-first search at the level of symbolic paths. We start by initializing a work list Q to contain only the empty sequence and a set of visited states V (line 1). The algorithm then performs the main loop until the work list is empty. A symbolic path sequence σ_s from the work list is extended by all method paths from all summaries (line 6). Here, we optionally filter out all sequences that partial order reduction identifies as redundant (line 7). Details are discussed in Sec. 4.2.

For every remaining new symbolic path sequence σ'_s, we test if in the set of concrete invocations I the teacher already has the necessary values for exercising it concretely (line 8). If I does not contain the necessary concrete method invocations for exercising σ'_s, we use a constraint solver to generate adequate concrete values. Our implementation ensures that the concrete values generated in lines 8 and 11 as well as the test in line 10 are within the guarded method path from line 4.

If the generation of values with a constraint solver is successful, we add the new invocations to I, and the algorithm is restarted, guaranteeing that σ'_s can now be exercised with concrete invocations (lines 11–13). If σ'_s cannot be satisfied from $\vec{G}_0$ (i.e., if it is not a proper symbolic path sequence of $\mathcal{C}$), we can simply disregard it.

In case I contains the necessary concrete invocations to exercise σ'_s, we compare the iLTS and the component on these sequences (line 15). If their behavior diverges, we have

Algorithm 3 Counterexample analysis.

Input: A counterexample $\sigma_c = \rho_{c1}, \ldots, \rho_{ck}$, mappings Δ_C, Δ_G such that $\rho_{ci} \models \Delta_G \circ \Delta_C^{-1}(\rho_{ci}^R)$
Output: A counterexample or an alphabet refinement

```
1: if ρ^R_c1, ..., ρ^R_ck is counterexample then
2:    return ρ^R_c1, ..., ρ^R_ck to Learner
3: for 1 ≤ i ≤ k do
4:    σ^R_c = ρ^R_c1, ..., ρ^R_ci, ρ_c(i+1), ..., ρ_ck
5:    if eval(σ^R_c) ≠ eval(σ_c) then
6:       σ^u_c ← ρ^R_c1, ..., ρ^R_c(i−1)
7:       (Sig_m, γ) ← Δ_G ∘ Δ_C^{-1}(ρ^R_ci)
8:       σ^v_c ← ρ_c(i+1), ..., ρ_ck
9:       return RefineAlphabet(σ^u_c, (Sig_m, γ), σ^v_c)
```

found a counterexample, which we process further as described below and sketched in Alg. 3. If no counterexample is found, we add the sequence σ_s' to the queue given that it did not lead to error or unknown, in which case it is dropped. Additionally, when using state matching, we may drop the sequence if we already encountered a state that covers it.

State Matching. Since we rely on concrete execution, it is relatively simple to compare (concrete) states reached by tests for a symbolic path sequence σ_s using all the corresponding concrete executions in I^*. When comparing states, little attention has to be paid to the path that led to a state. We want to drop a state (i.e., the corresponding symbolic path sequence) only if we know that we have already found another prefix that will allow us to explore the same (or more) behaviors of the component.

Usually, this issue is addressed by defining some notion of *subsumption* at the symbolic level [5, 36]. In these works, the shape of the heap and the valuation of variables is stored and compared symbolically. Our approach is different in two respects. First, we currently do not consider the heap; hence we ignore it here and leave it as an area of future work. Second, instead of checking implication at the symbolic level, we just check set-containment at the level of concrete valuations.

However, we share with these works the idea that the sequence that led to a state, has to be considered part of the state since, along this sequence, some values might have been copied from parameters to variables of the component. For a symbolic path sequence σ_s, we say that the resulting state is *unaffected* by parameters in σ_s if the single resulting valuation $\vec{G}$ is invariant under different valuations of method parameters in σ_s, i.e., if for every concrete execution $\sigma_c \models \sigma_s$ the final valuation $\vec{G}$ is the same. The intuitive idea is that in σ_s parameters are never copied into variables.

This yields the following simple definition of subsumption for concrete states. Let $\vec{G}$ be the single, unaffected valuation of $\mathcal{G}$ that is reached by σ_s, and v the set of valuations reached by σ_s' using concrete executions from I^*. We say that $\vec{G}$ is subsumed by v if $\vec{G}$ is in v. Together with $\vec{G}$ being unaffected, this guarantees that v has a truly bigger potential for exploration than $\vec{G}$ in $\mathcal{C}$.

Although one might argue that our notion of state matching is not sophisticated enough, it in fact allows for completeness in a limited number of cases for which it also helps increase the efficiency of X-Psyco, as demonstrated by our experiments. However, more importantly, it provides a first step in this direction and opens up the path for improved notions of state matching, which could be the basis for achieving completeness in a much larger set of practical examples.

Alphabet Refinement. Once a counterexample is found during an equivalence query (line 15, Alg. 2), we have to check whether the counterexample indicates that the concrete representative method invocations used during learning are not representative of their assigned symbolic counterparts. This is detected if there is no corresponding representative counterexample, and signifies that we have to refine (at least) one of the symbolic symbols. If, on the other hand, we find a corresponding counterexample using only representative method invocations, it means that we can return the counterexample to L*.

The details are covered in Alg. 3, Alg. 4, and Alg. 5. Technically, a counterexample is a sequence $\sigma_c = \rho_{c1}, \ldots, \rho_{ck}$ for which $[\![A]\!](\sigma_c) \neq eval(\sigma_c)$. Using mappings Δ_C and Δ_G maintained by the teacher, we can compute the representative invocation ρ_{ci}^R for every ρ_{ci} in σ_c. For $\rho_{ci} = (Sig_m, \vec{P}_m)$, we first select a with $\Delta_G(\mathsf{a}) = (Sig_m, \gamma)$, where $\vec{P}_m$ satisfies γ. We then select the concrete representative $\rho_{ci}^R = \Delta_C(\mathsf{a})$ for a. Doing this for every invocation of the counterexample, we get a sequence σ_c^R of concrete method invocations that all belong to the membership-pool.

If σ_c^R is a counterexample, it is passed to the learning algorithm (lines 1–2, Alg. 3). Otherwise, σ_c is a counterexample to the current partitioning of methods by guards: since by construction $[\![A]\!](\sigma_c) = [\![A]\!](\sigma_c^R)$, and σ_c is a counterexample whereas $[\![A]\!](\sigma_c^R)$ is not, the two sequences correspond to the same guarded sequence but have different outcomes.

In such a case, we use symbolic execution to find a refinement of the symbolic guard γ of some method in the counterexample. Before, however, we have to determine which method to refine. Since $eval(\sigma_c) \neq eval(\sigma_c^R)$ there has to be (at least one) index i for which in $\mathcal{C}$

$$eval(\ \rho_{c1}^R, \ldots, \rho_{c(i-1)}^R,\ \rho_{ci},\ \rho_{c(i+1)}, \ldots \rho_{ck}\ )$$
$$\neq eval(\ \rho_{c1}^R, \ldots, \rho_{c(i-1)}^R,\ \rho_{ci}^R,\ \rho_{c(i+1)}, \ldots \rho_{ck}\ ).$$

A detailed discussion of this argument is presented in previous work [21].

The algorithm determines two sequences that differ only at position i, meaning that ρ_{ci}^R is not a faithful representative for ρ_{ci}. As shown in lines 6–9 of Alg. 3, the actual refinement is done using symbolic execution of a program with prefix $\rho_{c1}^R, \ldots, \rho_{c(i-1)}^R$, symbolic method invocation of (Sig_m, γ) corresponding to ρ_{ci}, and a concrete remainder $\rho_{c(i+1)}, \ldots, \rho_{ck}$. The symbolic parameters of γ will be traced in the concrete suffix of the generated program, making visible constraints on these parameters that may only be introduced in the suffix. Note that this is the only time except for the generation of summaries when we use symbolic execution.

Finally, the actual refinement of the alphabet is described in Alg. 4 and Alg. 5. Alg. 4 starts with a call to symbolic execution to generate a summary ξ of the program derived from the counterexample. Then, the symbol to be refined is removed from the alphabet αM of the learner. We repeat the same steps for the ok-, error-, and unknown-paths in the summary by calling Alg. 5.

We discuss the ok case in detail. First, the disjunction of the conditions of all ok-path is intersected with γ to determine the guard φ_m^{ok} of the refined symbol (line 4, Alg. 4). If the intersection is empty, the new guard cannot be satisfied,

Algorithm 4 Concolic alphabet refinement.

Input: A method signature with precondition (Sig_m, γ), concrete prefix σ_c^u, and concrete suffix σ_c^v
Output: Refined αM, Γ, Δ_G, Δ_C, and I

1: $P \leftarrow GenerateProgram(\sigma_c^u;$ if γ then $m;$ $\sigma_c^v)$
2: $\xi \leftarrow SymbolicallyExecute(P)$
3: $\alpha M \leftarrow \alpha M \setminus \{\Delta_G^{-1}(\ (Sig_m, \gamma)\ )\}$
4: $\varphi_m^{ok} \leftarrow \gamma \wedge \left(\bigvee_{\langle \mathsf{ok}, \varphi\rangle \in \xi} \varphi\right)$
5: **if** $SAT(\varphi_m^{ok})$ **then**
6: $\quad ExtendAlphabet(Sig_m, \varphi_m^{ok})$ (Alg. 5)
7: $\varphi_m^{err} \leftarrow \gamma \wedge \left(\bigvee_{\langle \mathsf{error}, \varphi\rangle \in \xi} \varphi\right)$
8: **if** $SAT(\varphi_m^{err})$ **then**
9: $\quad ExtendAlphabet(Sig_m, \varphi_m^{err})$ (Alg. 5)
10: $\varphi_m^{unk} \leftarrow \gamma \wedge \left(\bigvee_{\langle \mathsf{unknown}, \varphi\rangle \in \xi} \varphi\right)$
11: **if** $SAT(\varphi_m^{unk})$ **then**
12: $\quad ExtendAlphabet(Sig_m, \varphi_m^{unk})$ (Alg. 5)

Algorithm 5 Extend alphabet.

Input: A method signature with a (strengthened) precondition $\rho_g = (Sig_m, \gamma)$
Output: Refined αM, Γ, Δ_G, Δ_C, and I

1: $\mathtt{a}_{new} \leftarrow CreateSymbolForLstar()$
2: $\alpha M \leftarrow \alpha M \cup \{\mathtt{a}_{new}\}$
3: $\Gamma \leftarrow \Gamma \cup \{\gamma\}$
4: $\Delta_G(\mathtt{a}_{new}) \leftarrow (Sig_m, \gamma)$
5: $\Delta_C(\mathtt{a}_{new}) \leftarrow \rho_c$ such that $\rho_c \models \rho_g$
6: $I \leftarrow I \cup \{\rho_c\}$

and we do not have to create a new symbol for the ok part. If φ_m^{ok} is satisfiable, we invoke Alg. 5, which introduces a new symbol $\mathtt{a}_{new}$, updates Δ_G with the new guard, and finally selects a new unique representative concrete invocation for $\mathtt{a}_{new}$, which is added to Δ_C and to I. After a successful refinement, L* is restarted with the refined alphabet.

Mixed Parameters. Similarly to Psyco, X-Psyco does not handle mixed parameters, the case where symbolic analysis identifies the need to relate parameters of one method with values passed to parameters of previously invoked methods. We are working towards dealing with mixed parameters by learning more expressive automata. At the moment, our algorithms identify the case of mixed parameters and return a warning to the user.

4.2 Static Analysis for Partial Order Reduction (POR)

As mentioned in Sec. 2, we wish to take advantage of method independence in order to apply partial order reduction during sequence generation for equivalence queries. Two methods are mutually independent if the outcome of their execution does not depend on the order in which they are invoked. For example, in Fig. 1, methods *write* and *flush* are clearly independent from each other—they only touch field *sink* of the component, and they both only read this field. When executed consecutively, it is therefore sufficient to explore them in a single order.

To take advantage of this observation, X-Psyco first applies a static analysis that computes read and write effects of methods, i.e., it determines the fields that each method reads and writes to. For each method m, we store this information into the following two bitsets, where each bit corresponds to a field of the component to which m belongs: (1) R_m, where a bit i is set if m reads the field associated with position i, and (2) W_m, where a bit i is set if m writes to the field associated with position i. Then, two methods m_1 and m_2 are independent if:

- R_{m_1} does not intersect with W_{m_2},
- W_{m_1} does not intersect with R_{m_2}, and
- W_{m_1} does not intersect with W_{m_2}.

In other words, two methods are independent if whenever one method writes to a field, the other method cannot read or write to the same field.

A form of partial order reduction can then be applied based on the above information. If two methods are independent, their consecutive execution is only explored according to a pre-defined total order. During sequence generation, X-Psyco recalls the last method in the sequence so far, and when selecting a new method to add that is independent of the last method, the sequence is only extended if the two methods appear according to the pre-defined order. Otherwise, the selected method is skipped and a different option is explored for extending the sequence.

Our experiments demonstrate that the cumulative results of such partial order reduction can be significant, especially for components with a large number of methods, where the occurrence of independent methods is more likely.

5. EXPERIMENTS

We implemented our approach in a tool called X-Psyco within the JPF (Java Pathfinder) open-source software verification framework [23]. Note that X-Psyco is not an extension of Psyco, but rather an almost independent implementation of the algorithms described in this paper. X-Psyco consists of three modular JPF extensions: (1) `jpf-learnlib` implements an interface to the well-known LearnLib framework for automata learning [26]; (2) `jpf-jdart` is our concolic [17, 29] execution engine that uses the state-of-the-art Z3 SMT solver [13]; (3) `jpf-xpsyco` implements the core hybrid learning algorithm. Finally, in our experiments, static analysis for partial order reduction was performed using the OCSEGen tool [32, 33].

We performed the evaluation of X-Psyco on the following set of examples:

Signature A class from the *java.security* package used in a paper by Singh et al. [30].

Stream The *PipedOutputStream* class from the *java.io* package and our motivating example (see Fig. 1). Also taken from a paper by Singh et al. [30].

IntMath A class from the Google Guava repository [18]. It implements arithmetic operations on integer types.

AltBit Implements a communication protocol that has an alternating bit style of behavior. Howar et al. [21] use it as a case study.

CEV NASA Crew Exploration Vehicle (CEV) 1.5 EORLOR example modeling flight phases of a spacecraft. The example is based on a Java state-chart model available in the JPF project `jpf-statechart` under `examples/jpfESAS`. We translated the example from state-charts into plain Java.

AccMeter A Java Micro Edition (J2ME) class implementing a mobile phone accelerometer interface for a simple game [2].

Socket A class from the *java.net* package that implements client sockets.

Table 1 summarizes the obtained experimental results for X-Psyco. All experiments were performed on a 2GHz Intel Core i7 laptop with 8GB of memory running Mac OS X. We budgeted a total of one hour running time for each experiment, after which the used tool was terminated. The different groups of columns give experimental results for running various configurations of the tool: "Baseline" is our barebone X-Psyco algorithm that eagerly uses concrete values during learning and lazily resorts to symbolic reasoning when needed; "POR" and "Matching" give results for running X-Psyco when partial order reduction and state matching are enabled, respectively; "POR+Matching" combines the two. Table 2 shows the results of running the old Psyco algorithm [16] on the examples.

In the experiments, the depth k for equivalence queries gets incremented whenever no counterexample is obtained after exhausting exploration of the conjectured iLTS to depth k. In this way, we are able to report the maximum depth k_{max} that we can guarantee for our generated interfaces within the allocated time of one hour. We also report k_{min} at which the final iLTS gets generated. The difference $k_{max} - k_{min}$ gives us a notion of a "confidence interval" since it tells us for how many steps there was no refinement needed. Of course, users can increase the total time budget if they require additional guarantees. Note that when a component cannot assign to the internal state from parameters of its methods, by relying on our state matching algorithm we know when we completely explored the component. Then, we can stop the learning algorithm before the budgeted one hour expires, which is marked with $*$ in the experimental results.

Discussion. First, we compare the results for Psyco with the baseline of X-Psyco. The main difference between the two is that Psyco relies solely on concolic exploration, while X-Psyco eagerly performs concrete exploration during learning and lazily resorts to the more expensive concolic engine only when needed in order to achieve completeness. The baseline of X-Psyco clearly outperforms Psyco on all the benchmarks: the maximum explored depth k_{max} increases from 1.2x to 8.3x. We can therefore conclude that X-Psyco benefits from our hybrid eager-concrete/lazy-symbolic learning combination.

Next, we compare our optimizations implemented within X-Psyco. Both partial order reduction (POR) and state matching perform at least as good as the baseline. For POR, the most significant gains with respect to k_{max} are achieved for Stream (2.2x), IntMath (2x), and Socket (1.3x) examples since these examples have many independent methods. On the other hand, for state matching the most significant gains can be seen for Signature (completes in 51 seconds), IntMath (completes in 60 seconds), and CEV (3.8x) examples. Note that when using state matching, both Signature and IntMath components get completely explored in less than a minute. State matching works especially well for these examples because both have a finite concrete state space: Signature uses only one global integer variable to store its internal state (one of three integer constants). IntMath has no variables at all and thus only one concrete state. The exploration depth k_{max} of CEV improves almost 4x under state matching, which enables two additional interface refinements. Hence, state matching substantially improves the precision and confidence of the generated interface. Similarly, POR improves the precision and confidence of the Socket example over baseline.

Table 2: Experimental results for Psyco. We used the same setup as for X-Psyco (cf. Table 1).

Example	$\|\mathcal{M}\|$	$\|\alpha M\|$	$\|Q\|$	k_{min}/k_{max}
AltBit	3	6	6	4/35
Stream	5	6	4	2/8
Signature	6	6	5	2/7
IntMath	8	9	3	1/1
AccMeter	9	12	8	2/5
CEV	19	24	9	3/3
Socket	51	60	42	2/2

Finally, the combination of POR and state matching compounds benefits of each individual optimization. By relying on state matching, it completes exploration of Signature and IntMath components in less than a minute. For other examples, the explored depth increases when compared to the baseline from 1.2x to 4.9x, and when compared to Psyco from 1.4x to 8.4x. These results clearly show the potential of our hybrid learning approach.

Both optimizations as well as their combination have basically no influence on the AltBit example. POR does not rule out any sequences as the protocol has only two methods that access the same variables. State matching does not help because the implementation uses a sequence number that is increased after every other method invocation so that no reachable concrete state is subsumed by any other concrete state.

Note that the algorithm for learning complete interfaces of software components is doubly-exponential in depth k. First, checking equivalence up to depth k generates $|\alpha M|^k$ method sequences. Then, on top of that, each method sequence has $methodPaths^k$ paths that have to be explored, which when combined with the exponential number of sequences gives us doubly-exponential time in k. Therefore, even to slightly improve k_{max} requires significant improvements to the algorithm, which is what our optimizations achieved as proved by the experimental results.

Finally, we analyzed where X-Psyco spends most of its execution time, which is also hinting at potential future bottlenecks. Unsurprisingly, running times are dominated by k-bounded exploration of conjectured iLTSs, and time spent in the learning algorithm itself is negligible.

6. RELATED WORK

Interface generation for white-box components has been studied extensively in the literature (e.g., [3, 15, 16, 20, 30]). However, as discussed, none of the existing approaches that we are aware of employ a combination of static, dynamic, and symbolic analysis as proposed in this paper.

Automatically creating component models for black-box components is a related area of research. For methods with parameters, abstractions are introduced that map alphabet symbols into sets of concrete argument values. A set of argument values represents a partition, and is used to invoke

Table 1: Experimental results for X-Psyco. Time budget is set to one hour. $|\mathcal{M}|$ is the number of component methods (and also the size of the initial alphabet); k_{min} the value of k at which the final iLTS gets generated; k_{max} the maximum value of k explored (i.e., the generated iLTS is k_{max}-full); $|\alpha M|$ the size of the final alphabet; $|Q|$ the number of states in the final iLTS. Completely explored components are marked with $*$.

		Baseline			POR			Matching			POR+Matching		
Example	$\|\mathcal{M}\|$	k_{min}/k_{max}	$\|\alpha M\|$	$\|Q\|$	k_{min}/k_{max}	$\|\alpha M\|$	$\|Q\|$	k_{min}/k_{max}	$\|\alpha M\|$	$\|Q\|$	k_{min}/k_{max}	$\|\alpha M\|$	$\|Q\|$
AltBit	3	4/291	6	6	4/298	6	6	4/296	6	6	4/293	6	6
Stream	5	1/11	6	4	1/24	6	4	1/12	6	4	1/54	6	4
Signature	6	1/10	6	5	1/11	6	5	1/2*	6	5	1/2*	6	5
IntMath	8	1/8	9	3	1/16	9	3	1/1*	9	3	1/1*	9	3
AccMeter	9	2/6	12	8	2/7	12	8	2/6	12	8	2/7	12	8
CEV	19	4/4	25	15	4/5	25	15	5/15	27	34	5/16	27	34
Socket	51	2/3	60	42	2/4	60	42	2/4	60	42	2/4	60	42

a component method. In the work by Aarts et al. [1], abstractions are user-defined. Howar et al. [21] discover such abstraction mappings through an automated refinement process. In contrast to these works, availability of the component source code enables us to lazily use symbolic analysis and generate guards that characterize precisely each method partition, making the generated automata more informative. MACE [11] combines black- and white-box techniques to discover concrete input messages that generate new system states. These states are then used as starting points for symbolic exploration of component binaries. The input alphabet is refined based on a user-provided abstraction of output messages. MACE is focused on increasing path coverage to discover bugs and not the generation of precise component interfaces, which is the focus of our work. Therefore, MACE sacrifices completeness in order to achieve scalability, while our algorithm guarantees completeness up to depth k.

Interface generation is also related to assumption generation for compositional verification, where several learning-based approaches have been proposed [9, 10, 19, 27]. A type of alphabet refinement developed in this context [8, 14] is geared towards computing smaller assumption alphabets that guarantee compositional verification achieves conclusive results. None of these works address the automatic generation of method guards in the computed interfaces.

Recent work on the analysis of multi-threaded programs for discovering concurrency bugs involves computing traces and preconditions that aid component interface generation [7, 22]. However, the data that these works generate is limited and cannot serve the purpose of temporal interface generation as presented in this paper.

Finally, other approaches generate interfaces using static analysis [34], or a combination of static and dynamic analyses [35], or by extracting information from sample execution traces [4, 25]. None of these approaches combines the power of static, dynamic, and symbolic analysis in such a way to generate complete interfaces with method guards of unprecedented precision, while preserving scalability.

7. CONCLUSIONS

We presented a novel hybrid learning algorithm for effectively generating precise temporal component interfaces enriched with method guards. The algorithm is based on an innovative combination of static, dynamic, and symbolic analysis. We implemented the approach in X-Psyco, and demonstrated its effectiveness on a number of realistic software components. We showed that X-Psyco significantly outperforms our previous tool called Psyco, and therefore generates interfaces of unprecedented precision and confidence. Both state matching and partial order reduction can be refined to achieve further improvements of our approach. In the future, we plan to explore interface generation in the context of compositional verification and automatic test generation.

8. ACKNOWLEDGMENTS

We would like to thank Vishwanath Raman for many fruitful discussions and his help with `jpf-jdart`, Peter Mehlitz for the suggested improvements concerning the integration with JPF, Oksana Tkachuk for providing support for the static analysis engine in OCSEGen, and Malte Isberner and Marko Dimjašević for their helpful feedback. This research was partly sponsored by United States National Aeronautics and Space Administration (NASA) under Prime Contract No. NNA10DE60C.

9. REFERENCES

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *International Conference on Testing Software and Systems (ICTSS)*, pages 188–204, 2010.

[2] Circuit—simple J2ME game. `http://gamesdev.wordpress.com/2008/11/03/circuit-simple-j2me-game/`.

[3] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[5] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *International conference on Model Checking Software (SPIN)*, pages 163–181, 2006.

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

[7] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI), pages 330–340, 2010.

[8] S. Chaki and O. Strichman. Three optimizations for assume-guarantee reasoning with L*. *Formal Methods in System Design (FMSD)*, 32(3):267–284, 2008.

[9] Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *International Conference on Computer Aided Verification (CAV)*, pages 511–526, 2010.

[10] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 31–45, 2009.

[11] 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.

[12] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2:215–222, 1976.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[14] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 292–307, 2007.

[15] D. Giannakopoulou and C. S. Pasareanu. Interface generation and compositional verification in JavaPathfinder. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 94–108, 2009.

[16] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *International Static Analysis Symposium (SAS)*, pages 248–264, 2012.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[18] Guava: Google core libraries. `http://code.google.com/p/guava-libraries/`.

[19] A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. In *International Conference on Computer Aided Verification (CAV)*, pages 420–432, 2007.

[20] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *European Software Engineering Conference (ESEC) held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 31–40, 2005.

[21] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 263–277, 2011.

[22] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 19–30, 2012.

[23] JPF (Java Pathfinder). `http://babelfish.arc.nasa.gov/trac/jpf`.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[25] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *International Conference on Software Engineering (ICSE)*, pages 501–510, 2008.

[26] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 220–223, 2011.

[27] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design (FMSD)*, 32(3):175–205, 2008.

[28] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference (ESEC) held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 263–272, 2005.

[30] R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *International Conference on Computer Aided Verification (CAV)*, pages 527–542, 2010.

[31] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems (SFM)*, pages 256–296, 2011.

[32] O. Tkachuk. OCSEGen. `http://code.google.com/p/envgen/`.

[33] O. Tkachuk. OCSEGen: Open components and systems environment generator. In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP)*, 2013. To appear.

[34] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *European Software Engineering Conference (ESEC) held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 188–197, 2003.

[35] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, 2002.

[36] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.