Symbolic Learning of Component Interfaces*

Dimitra Giannakopoulou[1], Zvonimir Rakamarić**[2], and Vishwanath Raman[3]

[1] NASA Ames Research Center, USA
dimitra.giannakopoulou@nasa.gov
[2] School of Computing, University of Utah, USA
zvonimir.rakamaric@gmail.com
[3] Carnegie Mellon University, USA
vishwa.raman@sv.cmu.edu

Abstract. Given a white-box component $\mathscr{C}$ with specified unsafe states, we address the problem of automatically generating an interface that captures safe orderings of invocations of $\mathscr{C}$'s public methods. Method calls in the generated interface are guarded by constraints on their parameters. Unlike previous work, these constraints are generated automatically through an iterative refinement process. Our technique, named PSYCO (Predicate-based SYmbolic COmpositional reasoning), employs a novel combination of the L* automata learning algorithm with symbolic execution. The generated interfaces are three-valued, capturing whether a sequence of method invocations is safe, unsafe, or its effect on the component state is unresolved by the symbolic execution engine. We have implemented PSYCO as a new prototype tool in the JPF open-source software model checking platform, and we have successfully applied it to several examples.

1 Introduction

Component interfaces are at the heart of modular software development and reasoning techniques. Modern components are open building blocks that are reused or connected dynamically to form larger systems. As a result, component interfaces must step up, from being purely syntactic, to representing component aspects that are relevant to tasks such as dynamic component retrieval and substitution, or functional and non-functional reasoning about systems. This paper focuses on "temporal" interfaces, which capture ordering relationships between invocations of component methods. For example, for the NASA Crew Exploration Vehicle (CEV) model discussed in Sec. 7, an interface prescribes that a lunar lander cannot dock with a lunar orbiter without first jettisoning the launch abort sub-system. 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.

An ideal interface should precisely represent the component in all its intended usages. In other words, it should include all the good interactions, and exclude all problematic interactions. Previous work presented approaches for computing temporal interfaces using techniques such as predicate abstraction [16] and learning [2, 11, 25]. Our

* This research was supported by the NASA CMU grant NNA10DE60C.
** The author did this work while at Carnegie Mellon University.

work studies a more general problem: automatic generation of precise temporal interfaces for components that include methods with parameters. Whether a method call is problematic or not may depend on the actual values passed for its formal parameters. Therefore, we target the generation of interfaces which, in addition to method orderings, also include method *guards* (i.e., constraints on the parameters of the methods), as illustrated in Fig. 2. We are not aware of any existing approaches that provide a systematic and automated way of introducing method guards for temporal interface generation.

Our proposed solution is based on a novel combination of learning with symbolic execution techniques. In particular, we use the L* [3, 23] automata-learning algorithm to automatically generate a component interface expressed as a finite-state automaton over the public methods of the component. L* generates approximations of the component interface by interacting with a *teacher*. The teacher uses symbolic execution to answer queries from L* about the target component, and provides counterexamples to L* when interface approximations are not precise. The teacher may also detect a need for partitioning the space of input parameter values based on constraints computed by the underlying symbolic engine. The alphabet is then refined accordingly, and learning restarts on the refined alphabet. Several learn-and-refine cycles may occur during interface generation. The generated interfaces are three-valued, capturing whether a sequence of method invocations is safe, unsafe, or its effect on the component state is unresolved by the underlying symbolic execution engine.

We have implemented our approach within the JPF (Java Pathfinder) software verification toolset [20]. JPF is an open-source project developed at the NASA Ames Research Center. The presented technique is implemented as a new tool called PSYCO in the JPF project `jpf-psyco`. We have applied PSYCO to learn component interfaces of several realistic examples that could not be handled automatically and precisely using previous approaches. Our main contributions are summarized as follows:

- This work is the *first* to combine learning and symbolic techniques for temporal interface generation, including method guards. The automated generation and refinement of these guards is based on constraints that are computed by symbolic execution. A significant challenge, which our proposed algorithm addresses, is to develop a refinement scheme that guarantees progress and termination.
- We use three-valued automata to account for potential incompleteness of the underlying analysis technique. These automata record *precisely* whether a sequence of method invocations is safe, unsafe, or unresolved. As a result, subsequent alternative analyses can be targetted to unresolved paths.
- We implemented the approach in an *open-source* and *extensible* tool within JPF and successfully applied it to several realistic examples.

Related Work. Interface generation for white-box components has been studied extensively in the literature (e.g., [16, 2, 11, 25]). However, as discussed, we are not aware of any existing approach that provides a systematic and automated way of refining the interface method invocations using constraints on their parameters.

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 a component method. In the work by Aarts et

al. [1], abstractions are user-defined. Hower et al. [18] discover such abstraction mappings through an automated refinement process. In contrast to these works, availability of the component source code enables us to generate guards that characterize precisely each method partition, making the generated automata more informative. MACE [8] combines black- and white-box techniques to discover concrete input messages that generate new system states. These states are then used as initial states for symbolic exploration on component binaries. The input alphabet is refined based on a user-provided abstraction of output messages. MACE focuses on increasing path coverage to discover bugs, rather than generating precise component interfaces, as targeted here.

Interface generation is also related to assumption generation for compositional verification, where several learning-based approaches have been proposed [22, 15, 7, 6]. A type of alphabet refinement developed in this context is geared towards computing smaller assumption alphabets that guarantee compositional verification achieves conclusive results [10, 5]. None of these works address the automatic generation of method guards in the computed interfaces/assumptions. Finally, recent work on the analysis of multi-threaded programs for discovering concurrency bugs involves computing traces and preconditions that aid component interface generation [4, 19]. However, the data that these works generate is limited, and cannot serve the purpose of temporal interface generation, as presented in this paper.

2 Motivating Example

Our motivating example is the *PipedOutputStream* class taken from the *java.io* package. Similar to previous work [2, 25], we removed unnecessary details from the example; Fig. 1 shows the simplified code. The example has one private field *sink* of type *PipedInputStream*, and four public methods called *connect*, *write*, *flush*, and *close*. Throwing exceptions is modeled by asserting *false*, denoting an undesirable error state.

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

Fig. 2 shows on the right the interface generated with PSYCO for this example. Note that, as described in Section 4, PSYCO currently only handles basic types. Therefore, we transformed the example in Figure 1 accordingly. 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 $snk \neq null \wedge snk.connected = false$, over the parameter *snk* of the method *connect*, captures the condition for a successful connection. Without support for guards in our component interfaces, we would obtain the interface shown on the left. This interface allows only methods that can be invoked unconditionally, i.e., *close* and *flush*. Method *connect* is blocked from the interface since it cannot be called unconditionally. Since *connect* cannot be invoked, *write* is blocked as well. Clearly, the interface on the left, obtained using existing interface generation techniques, precludes several legal sequences of method invocations. In existing ap-

```
class PipedOutputStream {
  PipedInputStream sink = null;

  public void connect(
      PipedInputStream snk) {
    if (snk == null) {
      assert false;
    } else if (sink != null ||
        snk.connected) {
      assert false;
    }
    sink = snk;
    snk.connected = true;
  }
}
```

```
public void write() {
  if (sink == null) {
    assert false;
  } else {...}
}

public void flush() {
  if (sink != null) {...}
}

public void close() {
  if (sink != null) {...}
}
}
```

Fig. 1: Motivating example.

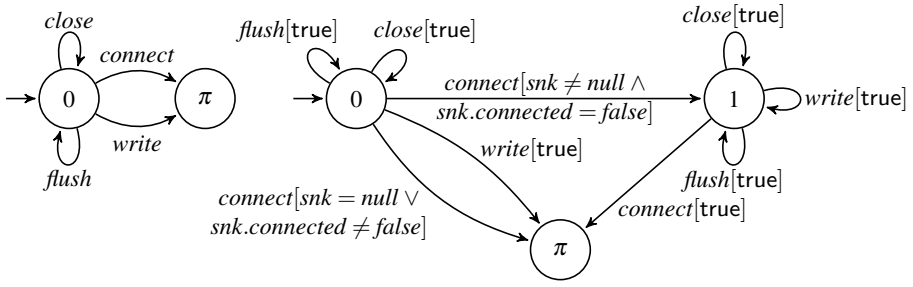

Fig. 2: Interfaces for our motivating example. On the left, there is no support for guards, while on the right, PSYCO is used to generate guards. Initial states are marked with arrows that have no origin; error states are marked with π. Edges are labelled with method names (with guards, when applicable).

proaches, a user is expected to manually define a refinement of the component methods to capture these additional legal behaviors. Our approach performs such a refinement automatically. Therefore, support for automatically generating guards enables PSYCO to generate richer and more precise component interfaces for components that have methods with parameters.

3 Preliminaries

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: 1) Q is a finite non-empty set of states, 2) αM is a set of observable actions called the *alphabet* of M, 3) $\delta : (Q \times \alpha M) \mapsto Q$ is a transition function, and 4) $q_0 \in Q$ is the initial state. LTS M is complete if each state except π and υ has outgoing transitions for every action in αM.

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

Three-Valued Automata Learning with L*. We use an adaptation [7] of the classic L* learning algorithm [3, 23], which learns a three-valued deterministic finite-state automaton (3DFA) over some alphabet Σ. In our setting, learning is based on partitioning the words over Σ into three unknown regular languages L_1, L_2, and L_3, with L* using this partition to infer an LTS with three values 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 $\sigma \in \Sigma^*$ and answers *true* if $\sigma \in L_1$, *false* if $\sigma \in L_2$, and *unknown* otherwise. The second type is an *equivalence query* or *conjecture*, i.e., given a candidate LTS M whether or not the following holds: $\mathscr{L}_{legal}(M) = L_1$, $\mathscr{L}_{illegal}(M) = L_2$, and $\mathscr{L}_{unknown}(M) = L_3$. If the above conditions 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 σ 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 numbers of states) LTS for L_1, L_2, and L_3.

Symbolic Execution. Symbolic execution is a static program analysis technique for systematically exploring a large number of program execution paths [21]. It uses symbolic values as program inputs in place of concrete (actual) values. The resulting output values are then statically computed as symbolic expressions (i.e., constraints), 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 conditions are checked for satisfiability using a constraint solver to establish whether the corresponding execution path is feasible.

$$
\begin{array}{rcl}
\textit{Component} & ::= & \textbf{class}\ \textit{Ident}\ \{\ \textit{Global}^*\ \textit{Method}^+\ \} \\
\textit{Method} & ::= & \textit{Ident}\ (\textit{Parameters})\ \{\ \textit{Stmt}\ \} \\
\textit{Global} & ::= & \textit{Type}\ \textit{Ident}; \\
\textit{Arguments} & ::= & \textit{Arguments},\ \textit{Expr} \mid \varepsilon \\
\textit{Parameters} & ::= & \textit{Pararameters},\ \textit{Parameter} \mid \varepsilon \\
\textit{Parameter} & ::= & \textit{Type}\ \textit{Ident}
\end{array}
\qquad
\begin{array}{rcl}
\textit{Stmt} & ::= & \textit{Stmt};\ \textit{Stmt} \\
& \mid & \textit{Ident} = \textit{Expr} \\
& \mid & \textbf{assert}\ \textit{Expr} \\
& \mid & \textbf{if}\ \textit{Expr}\ \textbf{then}\ \textit{Stmt}\ \textbf{else}\ \textit{Stmt} \\
& \mid & \textbf{while}\ \textit{Expr}\ \textbf{do}\ \textit{Stmt} \\
& \mid & \textbf{return}\ \textit{Expr}
\end{array}
$$

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

4 Components and Interfaces

Components and Methods. A *component* is defined by the grammar in Fig. 3. A component $\mathscr{C}$ has a set of global variables representing internal state and a set of one or more methods. Furthermore, components are sequential. For simplicity of exposition, we assume there is no recursion, and all method calls are inlined. Note, however, that our implementation handles calls without inlining. Moreover, as customary, our symbolic execution engine unrolls recursion to a bounded depth. We also assume the usual statement semantics. We expect that all unsafe states are implied by assert statements. 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 $\langle Id_m, P_m \rangle \in \mathsf{Ids} \times 2^{\mathsf{Prms}}$; we write $Id_m(P_m)$ for the signature Sig_m of the method m. A *method* m is then defined as a pair $\langle Sig_m, s_m \rangle$ where $s_m \in \mathsf{Stmts}$ is its top-level statement.

Let $\mathscr{M}$ be the set of methods in a component $\mathscr{C}$ and G be the set of its global variables. For every method $m \in \mathscr{M}$, 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 $m \in \mathscr{M}$, an execution $\theta \in \mathsf{Stmts}^*$ of m is a finite sequence of visited statements $s_1 s_2 \ldots s_n$ 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. Again, in our implementation, loops are unrolled to a bounded depth, as is customary in symbolic execution. A *valuation* over P_m, denoted $[P_m]$, is a function that assigns to each parameter $p \in P_m$ a value in D_p. We denote a valuation over variables in G with $[G]$. We take $[G_i]$ as the valuation representing the initial values of all global variables. Given valuations $[P_m]$ and $[G]$, 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 method parameters symbolically, using the name of each parameter as its symbolic name; we abuse notation and 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 is a constant, $p \in \mathsf{Prms}$ a parameter, and $\circ \in \{+, -, *, /, \%\}$ an arithmetic operator. The set of constants in an expression may include constants that are used in statements or the initial values of component state variables in $[G_i]$.

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.

Guards. Given a method signature $m = \langle Id_m, P_m \rangle$, a guard γ_m is defined as a constraint that only includes parameters from P_m.

Interfaces. Previous work uses LTSs to describe temporal component interfaces. However, as described in Sec. 2, a more precise interface ideally also uses guards to capture constraints on method input parameters.

We define an *interface LTS*, or *iLTS*, to take into account guards, as follows. An iLTS is a tuple $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$, where $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is a deterministic and complete LTS, $\mathscr{S}$ a set of method signatures, Γ a set of guards for method signatures in $\mathscr{S}$, and $\Delta : \alpha M \mapsto \mathscr{S} \times \Gamma$ a function that maps each $\mathtt{a} \in \alpha M$ into a method signature $m \in \mathscr{S}$ and a guard $\gamma_m \in \Gamma$. In addition, the mapping Δ is such that the set of all guards for a given method signature form a partition of the input space of the corresponding method. Let $\Gamma_m = \{\gamma \mid \exists \mathtt{a} \in \alpha M . \Delta(\mathtt{a}) = (m, \gamma)\}$ be the set of guards belonging to a method m. More formally, the guards for a method are (1) non-overlapping:

$$\forall \mathtt{a}, \mathtt{b} \in \alpha M,\ \gamma_\mathtt{a}, \gamma_\mathtt{b} \in \Gamma,\ m \in \mathscr{S}\ .\ \mathtt{a} \neq \mathtt{b} \wedge \Delta(\mathtt{a}) = (m, \gamma_\mathtt{a}) \wedge \Delta(\mathtt{b}) = (m, \gamma_\mathtt{b}) \Rightarrow \neg\gamma_\mathtt{a} \vee \neg\gamma_\mathtt{b},$$

(2) cover all of the input space: $\forall m \in \mathscr{S}\ .\ \bigvee_{\gamma \in \Gamma_m} \gamma = \mathsf{true}$, and (3) are non-empty.

Given an iLTS $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$, an execution of A is a sequence of pairs $\sigma = (m_0, \gamma_{m_0}), (m_1, \gamma_{m_1}), \ldots, (m_n, \gamma_{m_n})$, where for $0 \leq i \leq n$, pair (m_i, γ_{m_i}) consists of a method signature $m_i \in \mathscr{S}$ and its corresponding guard γ_{m_i}. Every execution σ has a corresponding trace $\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n$ in M such that for $0 \leq i \leq n$, $\Delta(\mathtt{a}_i) = (m_i, \gamma_{m_i})$. Then σ 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 $\mathscr{L}_{\mathit{legal}}(A)$, $\mathscr{L}_{\mathit{illegal}}(A)$, and $\mathscr{L}_{\mathit{unknown}}(A)$ as the sets of legal, illegal, and unknown executions of A, respectively.

An iLTS $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$ is an interface for a component $\mathscr{C}$ if $\mathscr{S}$ is a subset of method signatures of the methods $\mathscr{M}$ in $\mathscr{C}$. 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 $\mathscr{C}$ is associated with two characteristics: *safety* and *permissiveness*, meaning that the interface blocks all erroneous and allows all good executions (i.e., executions that do not lead to an error) of $\mathscr{C}$, respectively. A *full* interface is then an interface that is both safe and permissive [16].

We extend this definition to iLTSs as follows. Let iLTS A be an interface for a component $\mathscr{C}$. An execution $\sigma = (m_0, \gamma_{m_0}), (m_1, \gamma_{m_1}), \ldots, (m_n, \gamma_{m_n})$ of A then represents every concrete sequence $\sigma_c = (m_0, [P_{m_0}]), (m_1, [P_{m_1}]), \ldots, (m_n, [P_{m_n}])$ such that for

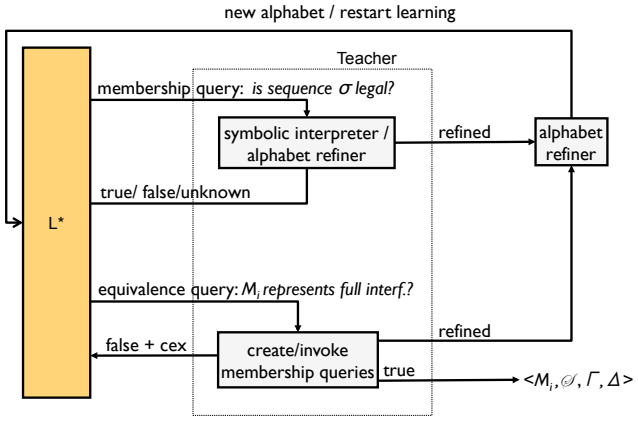

Fig. 4: PSYCO framework during iteration i of learning algorithm.

$0 \leq i \leq n$, $[P_{m_i}]$ satisfies γ_{m_i}. Each such concrete sequence defines an execution of the component $\mathscr{C}$. We say an execution of a component is illegal if it results in an assertion violation; otherwise, the execution is legal. Then, A is a *safe* interface for $\mathscr{C}$ if for every execution $\sigma \in \mathscr{L}_{legal}(A)$, we determine that all the corresponding concrete executions of component $\mathscr{C}$ are legal. It is *permissive* if for every execution $\sigma \in \mathscr{L}_{illegal}(A)$, we determine that all the corresponding concrete executions of component $\mathscr{C}$ are illegal. Finally, A is *tight* if for every execution $\sigma \in \mathscr{L}_{unknown}(A)$, we cannot determine whether the corresponding concrete executions of component $\mathscr{C}$ are legal or illegal; this explicitly captures possible incompleteness of the underlying analysis technique. To conclude, 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.

5 Symbolic Interface Learning

Let $\mathscr{C}$ be a component and $\mathscr{S}$ the set of signatures of a subset of the methods $\mathscr{M}$ in $\mathscr{C}$. Our goal is to automatically compute an interface for $\mathscr{C}$ as an iLTS $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$. We achieve this through a novel combination of L* to generate LTS M, and symbolic execution to compute the set of guards Γ and the mapping Δ. The termination criterion for symbolic execution is that all paths be characterized as either legal, illegal or unknown.

At a high level, our proposed framework operates as follows (see Fig. 4). It uses L* to learn an LTS over an alphabet that initially corresponds to a set of signatures $\mathscr{S}$ of the methods of $\mathscr{C}$. For our motivating example, we start with the alphabet $\alpha M = \{close, flush, connect, write\}$, set of signatures $\mathscr{S} = \{close(), flush(), connect(snk), write()\}$, and Δ such that $\Delta(close) = (close(), \mathsf{true})$, $\Delta(flush) = (flush(), \mathsf{true})$, $\Delta(connect) = (connect(snk), \mathsf{true})$, and $\Delta(write) = (write(), \mathsf{true})$. As mentioned earlier, L* interacts with a teacher that responds to its membership and equivalence queries. A membership query over the al-

phabet αM is a sequence $\sigma = \mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n$ such that for $0 \leq i \leq n$, $\mathtt{a}_i \in \alpha M$. Given a query σ, the teacher uses symbolic execution to answer the query. The semantics of executing a query in this context corresponds to exercising all paths through the methods in the query sequence, subject to satisfying the guards returned by the map Δ. Whenever the set of all paths through the sequence can be partitioned into proper subsets that are safe, lead to assertion violations, or to limitations of symbolic execution that prevent further exploration, we refine guards to partition the input space of the methods in the query sequence. We call this process *alphabet refinement*.

For our motivating example, the sequence $\sigma = \mathit{connect}$ will trigger refinement of symbol *connect*. As illustrated in Fig. 2, the input space of method *connect* must be partitioned into the case where: (1) $\mathit{snk} \neq \mathit{null} \wedge \mathit{snk.connected} = \mathit{false}$, which leads to safe executions, and (2) the remaining inputs, which lead to unsafe executions. When a method is partitioned, we replace the symbol in αM corresponding to the refined method with a fresh symbol for each partition, and the learning process is restarted with the new alphabet. For example, we partition the symbol *connect* into *connect_1* and *connect_2*, corresponding to the two cases above, before we restart learning. The guards that define the partitions are stored in Γ, and the mapping from each new symbol to the corresponding method signature and guard is stored in Δ.

Algo. 1 Learning an iLTS for a component.

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

```
 1: αM ← ∅, Γ ← {true}
 2: for all m ∈ 𝒮 do
 3:     a ← CreateSymbol()
 4:     αM ← αM ∪ {a}
 5:     Δ(a) ← (m, true)
 6: loop
 7:     AlphabetRefiner.init(αM, Δ)
 8:     SymbolicInterpreter.init(
 9:         αM, AlphabetRefiner)
10:     Teacher.init(Δ, SymbolicInterpreter)
11:     Learner.init(αM, Teacher)
12:     M ← Learner.learnAutomaton()
13:     if M = null then
14:         (αM, Γ, Δ)
15:         ← AlphabetRefiner.getRefinement()
16:     else
17:         return A = ⟨M, 𝒮, Γ, Δ⟩
```

Algo. 1 is the top-level algorithm implemented by our interface generation framework. First, we initialize the alphabet αM and the set of guards Γ on line 1. Then, we create a fresh symbol a for every method signature m, and use it to populate the alphabet αM and the mapping Δ (lines 2–5). The main loop of the algorithm learns an interface for the current alphabet; the loop either refines the alphabet and reiterates, or produces an interface and terminates. In the loop, an alphabet refiner is initialized on line 7, and is passed as an argument for the initialization of the *SymbolicInterpreter* on

```
void main(PipedInputStream snk) {
  assume true; close();
  assume snk != null && snk.connected == false; connect(snk);
  assume true; write();
}
```

Fig. 5: The generated program P_σ for the query sequence $\sigma = close, connect\_1, write$, where $\Delta(close) = (close(), \mathsf{true})$, $\Delta(connect\_1) = (connect(snk), snk \neq null \wedge snk.connected = false)$, and $\Delta(write) = (write(), \mathsf{true})$.

line 9. The *SymbolicInterpreter* is responsible for invoking the symbolic execution engine and interpreting the obtained results. It may, during this process, detect the need for alphabet refinement, which will be performed through invocation of *AlphabetRefiner*. We initialize a teacher with the current alphabet and the *SymbolicInterpreter* on line 10, and finally a learner with this teacher on line 11. The learning process then takes place to generate a classical LTS M (line 12). When learning produces an LTS M that is not *null*, then an iLTS A is returned that consists of M and the current guards and mapping, at which point the framework terminates (line 17). If M is *null*, it means that refinement took place during learning. We obtain the new alphabet, guards, and mapping from the *AlphabetRefiner* (line 15) and start a new learn-refine iteration.

Teacher. As discussed in Sec. 3, the teacher responds to membership and equivalence queries produced by L*. Given a membership query $\sigma = \mathsf{a}_0, \mathsf{a}_1, \ldots, \mathsf{a}_n$, the symbolic teacher first generates a program P_σ. For each symbol a_i in the sequence, P_σ invokes the corresponding method m_i while assuming its associated guard γ_{m_i} using an assume statement. The association is provided by the current mapping Δ, i.e., $\Delta(\mathsf{a}_i) = (m_i, \gamma_{m_i})$. The semantics of statement **assume** *Expr* is that it behaves as skip if *Expr* evaluates to true; otherwise, it blocks the execution. This ensures that symbolic execution considers only arguments that satisfy the guard, and ignores all other values.

For the example of Fig. 1, let $\sigma = close, connect\_1, write$ be a query, where $\Delta(close) = (close(), \mathsf{true})$, $\Delta(connect\_1) = (connect(snk), snk \neq null \wedge snk.connected = false)$, and $\Delta(write) = (write(), \mathsf{true})$. Fig. 5 gives the generated program P_σ for this query. Such a program is then passed to the *SymbolicInterpreter* that performs symbolic analysis and returns one of the following: (1) `TRUE` corresponding to a *true* answer for learning, (2) `FALSE` corresponding to a *false* answer, (3) `UNKNOWN` corresponding to an *unknown* answer, and (4) `REFINED`, reflecting the fact that alphabet refinement took place, in which case the learning process must be interrupted, and the learner returns an LTS $M = null$.

An equivalence query checks whether the conjectured iLTS $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$, with $M = \langle Q, \alpha M, \delta, q_0 \rangle$, is safe, permissive, and tight. One approach to checking these three properties would be to encode the interface as a program, similar to the program for membership queries. During symbolic execution of this program, we would check whether the conjectured iLTS correctly characterizes legal, illegal, and unknown uses of the component. However, conjectured interfaces have unbounded loops; symbolic techniques handle such loops through bounded unrolling. We follow a similar process,

but rather than having the symbolic engine unroll loops, we reduce equivalence queries to membership queries of bounded depth. Note that this approach, similar to loop unrolling during symbolic execution, is not complete in general. However, even in cases where we face incompleteness, we provide useful guarantees of the generated iLTS.

In order to provide guarantees of the generated interface to some depth k, we proceed as follows. During a depth-first traversal of M to depth k, whenever we reach state π or υ, we generate the sequence σ that leads to this state, where $\sigma = \mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_{n-1}, \mathtt{a}_n$ in $\mathscr{L}_{illegal}(M)$ or $\mathscr{L}_{unknown}(M)$, respectively. Moreover, we generate the sub-sequence $\sigma_L = \mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_{n-1}$, knowing $\sigma_L \in \mathscr{L}_{legal}(M)$, since π and υ have no outgoing transitions ($\mathscr{L}_{illegal}(M)$ and $\mathscr{L}_{unknown}(M)$ are suffix-closed). Whenever depth k is reached during traversal, and the state reached is not π and not υ, we generate the sequence $\sigma = \mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_k$ (with $\sigma \in \mathscr{L}_{legal}(M)$) leading to this state. In other words, we generate all legal sequences in $\mathscr{L}_{legal}(M)$ of depth exactly k, all sequences in $\mathscr{L}_{illegal}(M)$ and $\mathscr{L}_{unknown}(M)$ of depth less than or equal to k, as well as the largest prefix of each generated illegal and unknown sequence that is in $\mathscr{L}_{legal}(M)$.

Every generated sequence σ is then queried using the algorithm for membership queries. Since $\mathscr{L}_{legal}(M)$ is prefix-closed, and $\mathscr{L}_{illegal}(M)$ and $\mathscr{L}_{unknown}(M)$ are suffix-closed, the generated queries are sufficient to check the conjectured interface to depth k, as shown in the technical memorandum [12]. If the membership query for σ returns `REFINED`, learning is restarted since the alphabet has been refined. Furthermore, if the membership query for a sequence $\sigma \in \mathscr{L}_{legal}(M)$ (resp. $\sigma \in \mathscr{L}_{illegal}(M)$, $\sigma \in \mathscr{L}_{unknown}(M)$) does not return `TRUE` (resp. `FALSE`, `UNKNOWN`), the corresponding interface is not full and σ is returned to L* as a counterexample to the equivalence query. Otherwise, the interface is guaranteed to be k-full, i.e., safe, permissive, and tight up to depth k.

Symbolic Interpreter. Algo. 2 shows the algorithm implemented in *SymbolicInterpreter* and called by the teacher. The algorithm invokes a symbolic execution engine, and interprets its results to determine answers to queries. The input to Algo. 2 is a program P_σ as defined above, and a set of symbols Σ. The output is either `TRUE`, `FALSE`, or `UNKNOWN`, if no alphabet refinement is needed, or `REFINED`, which reflects that alphabet refinement took place.

Algo. 2 starts by executing P_σ symbolically (line 1), treating main method parameters (e.g., *snk* in Fig. 5) as symbolic inputs. Every path through the program is then characterized by a *path constraint*, denoted by *pc*. A *pc* is a constraint over symbolic parameters, with each conjunct in the constraint stemming from a conditional statement encountered along the path; a path constraint precisely characterizes a path taken through the program. A constraint partitions the set of all valuations over input parameters of the program (i.e., input parameters of the called component methods) into the set of valuations that satisfy the constraint and the set of valuations that do not satisfy the constraint. We denote a set of path constraints as *PC*.

We define a map $\rho : PC \mapsto \{\mathsf{error}, \mathsf{ok}, \mathsf{unknown}\}$ which, given a path constraint $pc \in PC$, returns error (resp. ok) if the corresponding path represents an erroneous (resp. good) execution of the program; otherwise, ρ returns unknown. Mapping *pc* to unknown represents a case when the path constraint cannot be solved by the underlying constraint solver used by the symbolic execution engine. Symbolic execution returns a

Algo. 2 Symbolic interpreter.

Input: Program P_σ and set of symbols Σ.
Output: TRUE, FALSE, UNKNOWN, or REFINED.

1: $(PC, \rho) \leftarrow SymbolicallyExecute(P_\sigma)$
2: $\varphi^{err} \leftarrow \varphi^{ok} \leftarrow \varphi^{unk} \leftarrow$ false
3: **for all** $pc \in PC$ **do**
4: **if** $\rho(pc) =$ error **then**
5: $\varphi^{err} \leftarrow \varphi^{err} \vee pc$
6: **else if** $\rho(pc) =$ ok **then**
7: $\varphi^{ok} \leftarrow \varphi^{ok} \vee pc$
8: **else**
9: $\varphi^{unk} \leftarrow \varphi^{unk} \vee pc$
10: **if** $\neg(SAT(\varphi^{err}) \vee SAT(\varphi^{unk}))$ **then**
11: **return** TRUE
12: **else if** $\neg(SAT(\varphi^{ok}) \vee SAT(\varphi^{unk}))$ **then**
13: **return** FALSE
14: **else if** $\neg(SAT(\varphi^{err}) \vee SAT(\varphi^{ok}))$ **then**
15: **return** UNKNOWN
16: **else**
17: $\Sigma_{new} \leftarrow AlphabetRefiner.refine(\varphi^{err}, \varphi^{unk})$
18: **if** $|\Sigma_{new}| = |\Sigma|$ **then**
19: **return** UNKNOWN
20: **else**
21: **return** REFINED

Algo. 3 Symbolic alphabet refinement.

Input: Set of symbols Σ, mapping Δ, and constraints φ^{err}, φ^{unk}.
Output: Refinement Σ_{new}, Γ_{new}, Δ_{new}.

1: $\Sigma_{new} \leftarrow \Gamma_{new} \leftarrow \emptyset$
2: **for all** $\mathtt{a} \in \Sigma$ **do**
3: $(m, \gamma) \leftarrow \Delta(\mathtt{a})$
4: $\varphi_m^{err} \leftarrow \Pi_m(\varphi^{err})$
5: $\varphi_m^{unk} \leftarrow \gamma \wedge \neg\varphi_m^{err} \wedge \Pi_m(\varphi^{unk})$
6: **if** $\neg MP(\varphi_m^{err}) \wedge \neg MP(\varphi_m^{unk})$ **then**
7: $\varphi_m^{ok} \leftarrow \gamma \wedge \neg\varphi_m^{err} \wedge \neg\varphi_m^{unk}$
8: **if** $SAT(\varphi_m^{err})$ **then**
9: $\mathtt{a}_{err} \leftarrow CreateSymbol()$
10: $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\mathtt{a}_{err}\}$
11: $\Gamma_{new} \leftarrow \Gamma_{new} \cup \{\varphi_m^{err}\}$
12: $\Delta_{new}(\mathtt{a}_{err}) \leftarrow (m, \varphi_m^{err})$
13: **if** $SAT(\varphi_m^{unk})$ **then**
14: $\mathtt{a}_{unk} \leftarrow CreateSymbol()$
15: $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\mathtt{a}_{unk}\}$
16: $\Gamma_{new} \leftarrow \Gamma_{new} \cup \{\varphi_m^{unk}\}$
17: $\Delta_{new}(\mathtt{a}_{unk}) \leftarrow (m, \varphi_m^{unk})$
18: **if** $SAT(\varphi_m^{ok})$ **then**
19: $\mathtt{a}_{ok} \leftarrow CreateSymbol()$
20: $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\mathtt{a}_{ok}\}$
21: $\Gamma_{new} \leftarrow \Gamma_{new} \cup \{\varphi_m^{ok}\}$
22: $\Delta_{new}(\mathtt{a}_{ok}) \leftarrow (m, \varphi_m^{ok})$
23: **else**
24: $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\mathtt{a}\}$
25: $\Gamma_{new} \leftarrow \Gamma_{new} \cup \{\gamma\}$
26: $\Delta_{new}(\mathtt{a}) \leftarrow (m, \gamma)$
27: **return** $\Sigma_{new}, \Gamma_{new}, \Delta_{new}$

set of path constraints PC and the mapping ρ, which are then interpreted by the algorithm to determine the answer to the query.

After invoking symbolic execution, the algorithm initializes three constraints (φ^{err} for error, φ^{ok} for good, and φ^{unk} for unknown paths) to false on line 2. The loop on lines 3–9 iterates over path constraints $pc \in PC$, and based on whether pc maps into error, ok, or unknown, adds pc as a disjunct to either φ^{err}, φ^{ok}, or φ^{unk}, respectively. Let $SAT : \Phi \mapsto \mathbb{B}$, where Φ is the universal set of constraints, be a predicate such that $SAT(\varphi)$ holds if and only if the constraint φ is satisfiable. In lines 10–15, the algorithm returns TRUE if all paths are good paths (i.e., if there are no error and unknown paths), FALSE if all paths are error paths, or UNKNOWN if all paths are unknown paths.

Otherwise, alphabet refinement needs to be performed; method *refine* of the *AlphabetRefiner* is invoked, which returns the new alphabet Σ_{new} (line 17). If no new symbols have been added to the alphabet, no methods have been refined. This can only happen if all potential refinements involve *mixed-parameter* constraints. Informally, a

constraint is considered mixed-parameter if it relates symbolic parameters from multiple methods. As explained in Algo. 3, dealing with mixed parameters precisely is beyond the scope of this work. Therefore, Algo. 2 returns `UNKNOWN`. Otherwise, refinement took place, and Algo. 2 returns `REFINED`.

Symbolic Alphabet Refinement. The *SymbolicInterpreter* invokes the refinement algorithm using method *refine* of the *AlphabetRefiner*. The current alphabet, mapping, and constraints φ^{err} and φ^{unk} computed by the *SymbolicInterpreter*, are passed as inputs. Method *refine* implements Algo. 3.

In Algo. 3, the new set of alphabet symbols Σ_{new} and guards Γ_{new} are initialized on line 1. The loop on lines 2–26 determines, for every alphabet symbol, whether it needs to be refined, in which case it generates the appropriate refinement. Let $\Delta(\mathtt{a}) = (m, \gamma)$. An operator Π_m is then used to project φ^{err} on the parameters of m (line 4). When applied to a path constraint pc_i, Π_m erases all conjuncts that don't refer to a symbolic parameter of m. If no conjunct remains, then the result is false. For a disjunction of path constraints $\varphi = pc_1 \vee \ldots \vee pc_n$ (such as φ^{err} or φ^{unk}), $\Pi_m(\varphi) = \Pi_m(pc_1) \vee \ldots \vee \Pi_m(pc_n)$. For example, if $m = \langle foo, \{x, y\}\rangle$, then $\Pi_m((s = t) \vee (x < y) \vee (s \leq z \wedge y = z)) \mapsto \mathsf{false} \vee (x < y) \vee (y = z)$, which simplifies to $(x < y) \vee (y = z)$.

We compute φ_m^{unk} on line 5. At that point, we check whether either φ_m^{err} or φ_m^{unk} involve mixed-parameter constraints (line 6). This is performed using a predicate $MP : \Phi \mapsto \mathbb{B}$, where Φ is the universal set of constraints, defined as follows: $MP(\varphi)$ holds if and only if $|Mthds(\varphi)| > 1$. The map $Mthds : \Phi \mapsto 2^{\mathscr{M}}$ maps a constraint $\varphi \in \Phi$ into the set of all methods that have parameters occurring in φ. Dealing with mixed-parameter constraints in a precise fashion would require more expressive automata, and is beyond the scope of this paper. Therefore, refinement proceeds for a symbol only if mixed-parameter constraints are not encountered in φ_m^{err} and φ_m^{unk}. Otherwise, the current symbol is simply added to the new alphabet (lines 24–26).

We compute φ_m^{ok} on line 7 in terms of φ_m^{err} and φ_m^{unk}, so it does not contain mixed-parameter constraints either. Therefore, when the algorithm reaches this point, all of φ_m^{err}, φ_m^{unk}, φ_m^{ok} represent potential guards for the method refinement. Note that φ_m^{err}, φ_m^{unk}, and φ_m^{ok} are computed in such a way that they partition the input space of the method m, if it gets refined. A fresh symbol is subsequently created for each guard that is satisfiable (lines 8, 13, 18), We update Σ_{new}, Γ_{new}, and Δ_{new} with the fresh symbol and its guard. In the end, the algorithm returns the new alphabet. The computed guards and mapping are stored in local fields that can be accessed through the getter method *getRefinement*() of the *AlphabetRefiner* (see Algo. 1, line 15).

6 Correctness and Guarantees

Prior to the application of our framework, loops and recursion are unrolled a bounded number of times. Consequently, our correctness arguments assume that methods have a finite number of paths. Proofs of our theorems appear in the technical memorandum [12].

We begin by showing correctness of the teacher for L*. In the following lemma, we prove that the program P_σ that we generate to answer a query σ captures all possible concrete sequences for σ. The proof follows from the structure of P_σ.

Lemma 1. (Correctness of P_σ). *Given a component $\mathscr{C}$ and a query σ on $\mathscr{C}$, the set of executions of $\mathscr{C}$ driven by P_σ is equal to the set of concrete sequences for σ.*

The following theorem shows that the teacher correctly responds to membership queries. The proof follows from the finiteness of paths taken through a component and from an analysis of Algo. 2.

Theorem 1. (Correctness of Answers to Membership Queries). *Given a component $\mathscr{C}$ and a query σ, the teacher responds* `TRUE` *(resp.* `FALSE`*,* `UNKNOWN`*) if and only if all executions of $\mathscr{C}$ for σ are legal (resp. illegal, cannot be resolved by the analysis).*

Next, we show that the teacher correctly responds to equivalence queries up to depth k. The proof follows from our reduction of equivalence queries to membership queries that represent all sequences of length $\leq k$ of the conjectured iLTS.

Theorem 2. (Correctness to Depth k of Answers to Equivalence Queries). *Let M be an LTS conjectured by the learning process for some component $\mathscr{C}$, Γ the current set of guards, and Δ the current mapping. If an equivalence query returns a counterexample, $A = \langle M, \mathscr{S}, \Gamma, \Delta \rangle$ is not a full interface for $\mathscr{C}$. Otherwise, A is k-full.*

In proving progress and termination of our framework, we use Lemma 2, which is a property of L*, and Lemma 3, which is a property of our alphabet refinement.

Lemma 2. (Termination of Learning). *If the unknown languages are regular, then L* is guaranteed to terminate.*

Lemma 3. (Alphabet Partitioning). *Algo. 3 creates partitions for the alphabet symbols it refines.*

Given that the number of paths through a method is bounded, we can have at most as many guards for the method as the number of these paths, which is bounded. Furthermore, if alphabet refinement is required, Algo. 3 always partitions at least one method. This leads us to the following theorem.

Theorem 3. (Progress and Termination of Refinement). *Alphabet refinement strictly increases the alphabet size, and the number of possible refinements is bounded.*

Finally, we characterize the overall guarantees of our framework with the following theorem, whose proof follows from Theorem 2, Theorem 3, and Lemma 2.

Theorem 4. (Guarantees of PSYCO). *If the behavior of a component $\mathscr{C}$ can be characterized by an iLTS, then* PSYCO *terminates with a k-full iLTS for $\mathscr{C}$.*

7 Implementation and Evaluation

We implemented our approach in a tool called PSYCO within the Java Pathfinder (JPF) open-source framework [20]. PSYCO consists of three new, modular JPF extensions: (1) `jpf-learn` implements both the standard and the three-valued version of

Example	#Methods	*k-max*	*k-min*	#Conjectures	#Refinements	#Alphabet	#States
SIGNATURE	5	7	2	2	0	5	4
PIPEDOUTPUTSTREAM	4	8	2	2	1	5	3
INTMATH	8	1	1	1	7	16	3
ALTBIT	2	35	4	8	3	5	5
CEV-FLIGHTRULE	3	4	3	3	2	5	3
CEV	18	3	3	10	6	24	9

Table 1: Experimental results. Time budget is set to one hour. "#Methods" is the number of component methods (and also the size of the initial alphabet); "*k-max*" the maximum value of k explored (i.e., the generated iLTS is *k-max*-full); "*k-min*" the smallest value of k for which our approach converges to the final iLTS that gets generated; "#Conjectures" the total number of conjectured iLTSs; "#Refinements" the total number of performed alphabet refinements; "#Alphabet" the size of the final alphabet; "#States" the number of states in the final iLTS.

L*; (2) `jpf-jdart` is our symbolic execution engine that performs concolic execution [13, 24]; (3) `jpf-psyco` implements the symbolic-learning framework, including the teacher for L*. For efficiency, our implementation of L* caches query results in a *MemoizedTable*, which is preserved after refinement to enable reuse of previous learning results. Programs P_σ are generated dynamically by invoking their corresponding methods using Java reflection. We evaluated our approach on the following examples:

SIGNATURE A class from the *java.security* package used in a paper by Singh et al. [25].

PIPEDOUTPUTSTREAM A class from the *java.io* package and our motivating example (see Fig. 1). Taken from a paper by Singh et al. [25].

INTMATH A class from the Google Guava repository [14]. It implements arithmetic operations on integer types.

ALTBIT Implements a communication protocol that has an alternating bit style of behavior. Howar et al. [18] use it as a case study.

CEV NASA Crew Exploration Vehicle (CEV) 1.5 EOR-LOR example modeling flight phases of a space-craft; a Java state-chart model in the JPF distribution under `examples/jpfESAS`. We translated the example from state-charts to plain Java.

CEV-FLIGHTRULE Simplified version of the CEV example that exposes a flight rule.

For all experiments, `jpf-jdart` used the Yices SMT solver [9]. The 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 application, after which PSYCO was terminated. Using a simple static analysis, PSYCO first checks whether a component is stateless. For stateless components (e.g., INTMATH), a depth of one suffices, hence we fix $k = 1$. For such components, the interface generated by PSYCO still provides useful information in terms of method guards. The resulting interface automaton for INTMATH can reach state unknown due to the presence of non-linear constraints, that cannot be solved using Yices. For all other components, the depth k for equivalence queries gets incremented whenever no counterexample is obtained after exhausting exploration of the automaton 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.

Table 1 summarizes the obtained experimental results. The generated interfaces are shown in [12]. In addition, we inspected the generated interfaces to check whether or not they correspond to our expected component behavior. For all examples, except CEV, our technique converges, within a few minutes and with a relatively small k (see column *k-min* in the table), to the expected iLTS. The iLTS do not change between *k-min* and *k-max*. Our technique guarantees they are $k\text{-}max\text{-}full$. In general, users of our framework may increase the total time budget if they require additional guarantees, or may interrupt the learning process if they are satisfied with the generated interfaces. In all of our examples the majority of the time was spent in symbolic execution.

A characteristic of the examples for which PSYCO terminated with a smaller *k-max*, such as CEV, is that they involve methods with a significant degree of branching. On the other hand, PSYCO managed to explore ALTBIT to a large depth because branching is smaller. This is not particular to our approach, but inherent in any path-sensitive program analysis technique. If n is the number of branches in each method, and a program invokes m methods in sequence, then the number of paths in this program is, in the worst case, exponential in $m*n$. As a result, symbolic analysis of queries is expensive both in branching within each method as well as in the length of the query. Memoization and reuse of learning results after refinement helps ameliorate this problem; for CEV, 7800 out of 12002 queries were answered through memoization.

8 Conclusions and Future Work

We have presented the foundations of a novel approach for generating temporal component interfaces enriched with method guards. PSYCO produces three-valued iLTS, with an unknown state reflecting component behavior that was not covered by the underlying analysis. For compositional verification, unknown states can be interpreted conservatively as errors, or optimistically as legal states, thus defining bounds for the component interface. Furthermore, alternative analyses can be applied subsequently to target these unexplored parts. The interface could also be enriched during testing or usage of the component. Reuse of previous learning results, similar to what is currently performed, could make this process incremental.

In the future, we also intend to investigate ways of addressing mixed parameters more precisely. For example, we plan to combine PSYCO with a learning algorithm for register automata [17]. This would enable us to relate parameters of different methods through equality and inequality. Moreover, we will incorporate and experiment with heuristics both in the learning and the symbolic execution components of PSYCO. Finally, we plan to investigate interface generation in the context of compositional verification.

Acknowledgements

We would like to thank Peter Mehlitz for his help with Java PathFinder and Neha Rungta for reviewing a version of this paper.

References

1. F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *ICTSS*, pages 188–204, 2010.
2. R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
4. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, pages 330–340, 2010.
5. S. Chaki and O. Strichman. Three optimizations for assume-guarantee reasoning with L*. *FMSD*, 32(3):267–284, 2008.
6. 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 *CAV*, pages 511–526, 2010.
7. Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *TACAS*, pages 31–45, 2009.
8. 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.
9. B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI International, 2006.
10. M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, pages 292–307, 2007.
11. D. Giannakopoulou and C. S. Pasareanu. Interface generation and compositional verification in JavaPathfinder. In *FASE*, pages 94–108, 2009.
12. D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. Technical report, NASA Ames Research Center, 2012.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.
14. Guava: Google core libraries. http://code.google.com/p/guava-libraries/.
15. A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. In *CAV*, pages 420–432, 2007.
16. T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE*, pages 31–40, 2005.
17. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI*, 2012.
18. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, pages 263–277, 2011.
19. S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, pages 19–30, 2012.
20. Java PathFinder (JPF). http://babelfish.arc.nasa.gov/trac/jpf.
21. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
22. 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. *FMSD*, 32(3):175–205, 2008.
23. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
24. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.
25. R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *CAV*, pages 527–542, 2010.