Generating Component Interfaces by Integrating Static and Symbolic Analysis, Learning, and Runtime Monitoring*

Falk Howar[1], Dimitra Giannakopoulou[2], Malte Mues[3], and Jorge A. Navas[4]

[1] Dortmund University of Technology and Fraunhofer ISST, Dortmund, Germany
[2] NASA Ames Research Center, Moffett Field, CA, USA
[3] Dortmund University of Technology, Dortmund, Germany
[4] Computer Science Laboratory, SRI International, USA

Abstract. Behavioral interfaces describe the safe interactions with a component without exposing its internal variables and computation. As such, they can serve as documentation or formal contracts for black-box components in safety-critical systems. Learning-based generation of interaces relies on learning algorithms for inferring behavioral interfaces from observations, which are in turn checked for correctness by formal analysis techniques. Learning-based interface generation is therefore an interesting target when studying integration and combination of different formal analysis methods. In this paper, which accompanies an invited talk at the ISoLA 2018 track "A Broader View on Verification: From Static to Runtime and Back", we introduce interpolation and symbolic search for validating inferred interfaces. We discuss briefly how interface validation may utilize information from runtime monitoring.

1 Introduction

In the automotive and aeronautic industries, it is customary for original equipment manufacturers (OEMs) to contract the design and implementation of system components to external companies [18]. In such a scenario, the OEM specifies a set of requirements on the component that is to be developed. The contractor develops the component as a black-box: the delivered product does not typically include intermediate artifacts such as design models or source code. As a consequence, the only means of verifying external components is black-box testing, which provides no formal guarantees.

This situation can be mitigated to some extent if contractors produce high-level behavioral models which, while suitable for verification, do not expose details of the component implementation. For example, behavioral component interfaces describe the safe interactions with a component without exposing its internal variables and computation. Such interfaces can serve as assume-guarantee-style formal contracts for black-box components of flight-critical systems [18]. In

* This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4011 and the Office of Naval Research under Contract No. N68335-17-C-0558. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA, SSC Pacific, or the Office of Naval Research.

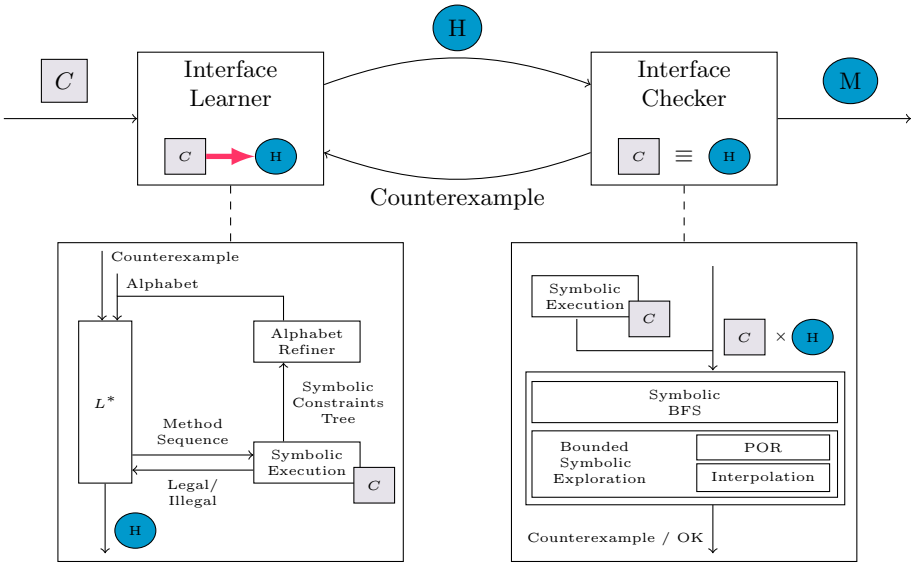

Fig. 1: Architecture of the PSYCO Tool for Interface Generation.

general, substituting component models with abstract component interfaces can significantly increase the scalability of formal analysis techniques for component-based safety-critical systems.

In order to generate such component interfaces for component implementations, the internal details of the component have to be abstracted and relevant observable behavior has to be projected to the interface. Learning-based interface-generation frameworks use learning algorithms to infer behavioral interfaces from observations; by design, the generated interfaces do not include structural or internal component information. On the other hand, learning from observations needs to be complemented by methods that check the quality (i.e., accuracy) of an inferred model, and that trigger model refinement, if necessary. The variety of formal techniques involved in learning-based interface generation makes interface learning frameworks great platforms for studying the integration and combination of different formal analysis methods.

This work uses the Java Pathfinder extension PSYCO [14, 17] for learning-based computation of behavioral interfaces of Java components. As discussed above, and as shown in Figure 1, PSYCO iterates between two modes of operation when generating interfaces: *interface learning*, the part that learns a model from observations, and *interface checking*, the part that checks the quality of the produced interfaces.

To generate candidate interfaces, PSYCO uses a combination of 1) active automata learning [2, 21] with alphabet abstraction refinement [19] and 2) dynamic symbolic execution [15]. Active automata learning algorithms infer finite state models by generating sequences of invocations to component methods and

observing the resulting output of components. Alphabet abstraction represents sets of equivalent concrete method invocations (especially with data parameters) symbolically. Symbolic predicates are recorded when executing method sequences generated by the learning algorithm with symbolic data parameters. Since automata learning infers models from observations, it is neither correct nor complete unless inferred models can be discharged against component implementations. To validate a candidate interface, PSYCO compares the interface to the component implementation up to a user-specified depth. As a consequence, PSYCO cannot provide strong guarantees on the correctness of generated interfaces.

This paper extends the *Interface Checker* of PSYCO (shown in the right half of Figure 1) with additional approaches. First, we use interpolation [10] to obtain over-approximating state predicates from paths explored in symbolic execution. These predicates allow us to skip symbolic execution for already explored parts of the state space. Second, symbolic search can be used for checking correctness of candidate interfaces. The basis for describing and implementing these extensions is a unified logic-based encoding of component interfaces and internal behavior. Finally, we briefly discuss how runtime monitoring and modern active learning algorithms can be used for "life-long" learning of interfaces at runtime. We evaluate the presented extensions on a set of Java components that have served as benchmarks in previous works on PSYCO.

The contribution of the presented work is two-fold: On the one hand, our extensions enable PSYCO to actually compute correct component interfaces — as opposed to interfaces with bounded guarantees. On the other hand, we show how quite different formal methods (at design time as well as at runtime) can be integrated and exchange information to solve the interface generation problem.

Related Work. Interface generation for white-box components has been studied extensively in the literature (e.g., [1, 13, 14, 16, 29, 6]). To the best of our knowledge, PSYCO [14, 17] is unique in that it generates interfaces that combine constraints over sequences of method invocations with constraints over values of method parameters and integrates static, dynamic, and symbolic analysis.

In this paper, we focus on the problem of checking equivalence of a conjectured interface model and a component implementation. We show that this problem can be formulated as a reachability analysis on the product of component and interface, which is a classic model-checking problem. Coudert et al. [9] present the idea of building the cross product of a component interface and its specification using Boolean formulas for verifying implementation correctness of electronic circuits. Our symbolic search is similar to this approach. We provide a state-of-the-art implementation of the search algorithm that allows solving of cross products in quantified bitvector logic.

Binary decision diagrams have been successfully used in the past as encoding for model checking algorithms to increase verifiable system size (e.g. [5]). These works are not directly applicable to our scenario as methods have data parameters which appear as quantified variables in state predicates. Another class of approaches relies on logic-based encoding and SAT- or SMT-solvers: Biere

et. al. [4] developed a logic-based Bounded Model Checker for LTL properties. McMillan [26] has shown that model checking can verify CTL properties using a SAT-solver able to solve universally quantified Boolean formulas. These works have shown that a symbolic encoding based on Boolean formulas is more space-efficient than BDDs but depends on deciding quantified formulas for reaching completeness. More recent works replaced SAT-solvers by SMT-solvers [3, 8]. Ge et al. [12] implemented complete instantiation for quantified formulas in Z3 allowing to solve quantified formulas in SMT. Wintersteiger et al. [30] extended this for the theory of quantified bitvector formulas (QBVF) and introduced symbolic quantifier instantiation to avoid complete instantiations whenever possible. Unfortunately, no effective interpolation techniques are available for bitvector formulas with or without quantifiers.

Jaffar et al. [24] were the first to tackle the path-explosion problem of symbolic execution by annotating symbolic states with learned facts (i.e., interpolants) after a search backtracks, and pruning future searches. This work was presented in the context of Constraint Logic Programming and it was extended to deal with C programs by [23, 22], but limited to intra-procedural programs over linear arithmetic and arrays. McMillan [27] generalized further Jaffar and colleagues' ideas to deal with richer SMT theories and recursive functions, naming also the idea with the term *Lazy Annotation*. Our work applies the ideas presented in these works to symbolic execution of reactive components. The main difference between our use case and symbolic execution of sequential programs is that for components there is no sensible notion of program location. The environment can call arbitrary sequences of methods. As a consequence, we can use state predicates only for component states at the same depth of exploration.

Outline. The remainder of the paper is structured as follows: We start by providing some intuition on the problem we are addressing in the next section before providing some formal preliminaries in Section 3. In Section 4, we present the technical details on the new extensions to PSYCO. Results from an evaluation can be found in Section 5. We conclude in Section 6 by discussing future research directions with a particular focus on combining analysis steps at design time and at runtime.

2 Motivating Example

We are interested in generating behavioral interfaces for components. In our context, a component comprises a finite set of primitive state variables and one or more methods with data parameters that operate on the state variables. Recursion and loops without constant bounds are not allowed in methods. This is a common restriction for embedded systems that must guarantee safety critical constraints. A formal definition of components can be found in [17, 14].

For the remainder of the paper we assume JAVA implementations for components. An example of a component is shown in Figure 2a. The `Explorer` Component exposes methods `move(int p)` and `check()`. Method `move` adds the value

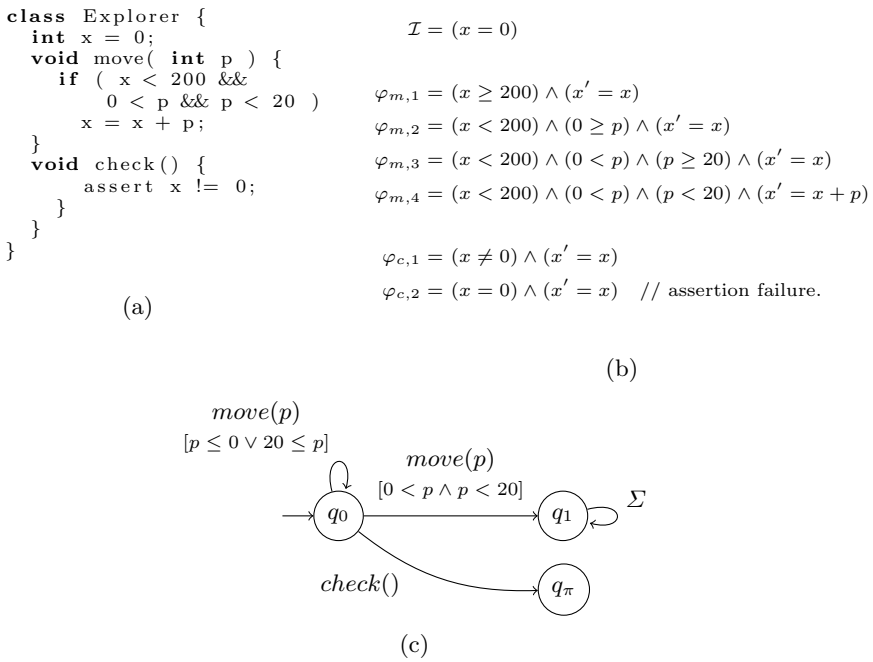

Fig. 2: JAVA Component (a), Corresponding Symbolic Method Paths (b), and Behavioral Interface (c).

of parameter p to internal state variable x if x and p are within certain bounds. Method `check` asserts that state variable x is not zero.

Behavioral interfaces are formal models of components that describe all sequences of method invocations on a component that are safe, i.e., will not lead to errors. For the `Explorer` component shown in the figure, the sequence of method invocations τ_1 = `move(1) check()` will not lead to an error while sequence τ_2 = `move(0) check()` leads to an assertion violation. The interfaces generated by PSYCO are finite-state automata whose transitions are labeled with method names and guarded with constraints on the corresponding method parameters, as shown in Figure 2c. Initially, it is not safe to call `check` (the automaton enters the error state q_π). The guards partition the input spaces of parameters: Invocations of `move` with values of p in the interval $[1, 19]$ put the component into a state (q_1 in the model) from which all method invocations are safe. Calling `move` after instantiation with value of p outside $[1, 19]$ has no effect on the state of the component.

As shown in Figure 1 and detailed in [17], PSYCO generates interfaces by iterating two modes of operation: The *interface learner* generates conjectured interfaces using a combination of active automata learning, automated alphabet abstraction, and dynamic symbolic execution. As learning is neither sound nor complete, learned interfaces have to be validated against the component implementation. The *interface checker* of PSYCO does this by executing the interface

and the component in parallel, and checking agreement of their error states. For the remainder of the paper, we focus on two extensions of the interface checker that compute correctness of interfaces on symbolic representations of component and interface.

Figure 2b shows fully symbolic execution paths (including symbolic state variables) of methods of the `Explorer` component. PSYCO uses the JDART dynamic symbolic execution engine for computing these paths. These fully symbolic method summaries can be understood as a logic-based encoding of the component's behavior as a transition system. We can respresent the interface as a symbolic transition system, too. In the next section, we explain how candidate interfaces are validated against components where symbolic representations are used for both components and interfaces.

3 Symbolic Representation of Components and Interfaces

We are interested in reactive components that manipulate a set of internal state variables. Components expose methods with data parameters that can be invoked from the outside. We abstract from output since we are only interested in safe interactions with a component, i.e., ones that do not trigger an error. For the sake of presentation we assume that all methods have the same number of parameters. We use logic-based symbolic transition systems for specifying the behavior of components and their interfaces.

Let $\mathbb{L}$ be a logic with decidable entailment $\models_{\mathbb{L}}$, and a set V of values allowed in $\mathbb{L}$. For the sake of readability we abstract from types in the presentation. For a formula φ in $\mathbb{L}$ with free variables $\mathbf{x}$, we write $\varphi[\mathbf{x}]$ when we want to emphasize that φ has free variables $\mathbf{x}$ and $\mathbb{F}_{\mathbf{x}}$ for the set of formulas with free variables $\mathbf{x}$. We write $\varphi[y/x]$ for the formula that results from replacing x by y in φ. For variables $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, a valuation is a mapping $\sigma : \mathbf{x} \mapsto V$ and $V_{\mathbf{x}}$ is the set of all valuations of $\mathbf{x}$. For a formula $\varphi[\mathbf{x}]$ and a valuation $\sigma \in V_{\mathbf{x}}$, let $\varphi[\sigma]$ denote the instantiation of $\mathbf{x}$ by σ in φ, i.e., $\varphi[\sigma(x_1)/x_1][\ldots][\sigma(x_n)/x_n]$. A valuation σ satisfies φ if $\varphi[\sigma]$ is true. A formula φ is satisfiable if there exists a valuation σ that satisfies φ. When a formula $\varphi[\mathbf{x}]$ entails a formula $\psi[\mathbf{x}]$, denoted by $\varphi \models_{\mathbb{L}} \psi$, then $\varphi[\sigma]$ being true implies that $\psi[\sigma]$ is true for any valuation σ in $V_{\mathbf{x}}$. We can test entailment by checking if $\varphi \wedge \neg\psi$ is satisfiable. If this formula is satisfiable then φ does not entail ψ. Entailment is more complex to check for formulas $\varphi[\mathbf{x}]$ and $\psi[\mathbf{y}]$, where $\mathbf{y}$ contains variables that do not belong to $\mathbf{x}$. Let $\mathbf{z} = \mathbf{y} \setminus \mathbf{x}$. In such a case, the formula testing for entailment becomes $(\varphi \wedge \forall \mathbf{z}.\neg\psi)$, where $\forall \mathbf{z}.\neg\psi$ denotes the formula in which all variables in $\mathbf{z}$ are bound by universal quantifiers for $\neg\psi$. Finally, if we have two inconsistent formulas $\varphi[\mathbf{x}]$ and $\psi[\mathbf{y}]$ (i.e., $\varphi \wedge \psi$ is not satisfiable), an interpolant [10] is a formula $i[\mathbf{x} \cap \mathbf{y}]$ over the common variables in φ and ψ that is implied by φ (i.e., $\varphi \implies i$) and inconsistent with ψ (i.e., $i \wedge \psi$ is unsatisfiable).

Definition 1 (Symbolic Transition System). *A symbolic transition system is a tuple $\mathcal{STS} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$, where M is a finite set of methods with method*

parameters $\mathbf{p}$*,* $\mathbf{x}$ *is a set of state variables,* $\mathcal{I}[\mathbf{x}]$ *is a logic formula describing a set of initial states, and* $\mathcal{T} \subseteq M \times \mathbb{F}_{\mathbf{x} \cup \mathbf{p} \cup \mathbf{x}'}$ *is a set of guarded method executions (i.e., transitions) with guards over state variables* $\mathbf{x}$ *and method parameters* $\mathbf{p}$*, and effects on updated state variables* $\mathbf{x}'$*. Finally,* $\mathcal{T}_\pi \subseteq \mathcal{T}$ *is a subset of guarded method executions that lead to errors.*

For the remainder of this paper, we focus on deterministic and input-complete symbolic transition systems, where for every method $m \in M$, every valuation $v_\mathbf{x} \in V_\mathbf{x}$ of state variables and every valuation $v_\mathbf{p} \in V_\mathbf{p}$ of method parameters, there exists exactly one transition $(m, \varphi) \in \mathcal{T}$ for which $\varphi[v_\mathbf{x}][v_\mathbf{p}]$ is satisfiable.

We define the semantics of a symbolic transition system $\mathcal{T} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$ in terms of feasible sequences of transitions. For a sequence $(m_1, \varphi_1), \ldots, (m_k, \varphi_k)$ of transitions, we define the strongest postcondition, denoted by $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_k)$, as the conjunction

$$\mathcal{I}[\mathbf{y}_1/\mathbf{x}] \ \wedge \ \varphi_1[\mathbf{y}_1/\mathbf{x}][\mathbf{p}_1/\mathbf{p}][\mathbf{y}_2/\mathbf{x}'] \ \wedge \ \ldots \ \wedge \ \varphi_k[\mathbf{y}_k/\mathbf{x}][\mathbf{p}_k/\mathbf{p}][\mathbf{x}/\mathbf{x}'],$$

where we assume sets $\mathbf{y}_i$ and $\mathbf{p}_i$ to contain uniquely named variables (especially when working with multiple post conditions). Analogously, the weakest precondition of a sequence of transitions and a state predicate $\psi[\mathbf{x}]$, denoted by $wpc(\varphi_1 \cdot \ldots \cdot \varphi_k \cdot \psi)$, is the conjunction

$$\varphi_1[\mathbf{x}/\mathbf{x}][\mathbf{p}_1/\mathbf{p}][\mathbf{y}_1/\mathbf{x}'] \ \wedge \ \ldots \ \wedge \ \varphi_k[\mathbf{y}_k/\mathbf{x}][\mathbf{p}_k/\mathbf{p}][\mathbf{y}_{k+1}/\mathbf{x}'] \ \wedge \ \psi[\mathbf{y}_{k+1}/\mathbf{x}].$$

Please note, that *wpc* and *spc* are formulas over $\mathbf{x}$ and sets of uniquely named variables.

A sequence $(m_1, \varphi_1), \ldots, (m_k, \varphi_k)$ of transitions is a (symbolic) *trace* of $\mathcal{STS}$ if the first $k-1$ transitions are from $(\mathcal{T} \setminus \mathcal{T}_\pi)$ and $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_k)$ is satisfiable. It is a *safe trace* if it ends with a transition from $\mathcal{T} \setminus \mathcal{T}_\pi$. A trace that ends with a transition from $\mathcal{T}_\pi$ is an *error trace* of $\mathcal{STS}$.

Example. Figure 2b shows path conditions for the JAVA component from Figure 2a. These paths are the basis for $\mathcal{STS}_{Ex} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$ with methods $M = \{move(p), check()\}$, variables $\mathbf{x} = \{x\}$, initial condition $\mathcal{I} = (x = 0)$, as well as transitions $\mathcal{T} = \{(move(p), \varphi_{m,1}), \ldots, (check(), \varphi_{c,2})\}$ and $\mathcal{T}_\pi = \{(check(), \varphi_{c,2})\}$. The strongest post-condition of the sequence of transitions $(move(p), \varphi_{m,2}), (check(), \varphi_{c,2})$ is

$$(y_1 = 0) \quad \wedge \quad (y_1 < 200) \ \wedge \ (0 \geq y_2) \ \wedge \ (y_3 = y_1) \quad \wedge \quad (y_3 = 0) \ \wedge \ (x = y_3).$$

The condition is satisfiable, which makes the transition sequence an error trace of $\mathcal{STS}_{Ex}$. Also, the condition describes the strongest post-condition on x after executing these method paths in sequence.

Behavioral Interfaces We fix a component with set of methods M and method parameters $\mathbf{p}$. A behavioral interface is a finite automaton that describes the safe traces of the component.

Definition 2 (Behavioral Interface). *A behavioral interface is a tuple* $\mathcal{A} = \langle \Sigma, Q, q_0, q_\pi, \Gamma \rangle$*, where* $\Sigma \subseteq M \times \mathbb{F}_{\mathbf{p}}$ *is finite set of guarded methods (and guards are only over* $\mathbf{p}$*),* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $q_\pi \in Q$ *is the error state, and* $\Gamma : (Q \setminus \{q_\pi\}) \times \Sigma \mapsto Q$ *is the transition function.*

As for symbolic transition systems, we assume deterministic automata where for every method $m \in M$, and every $v_{\mathbf{p}} \in V_{\mathbf{p}}$ there exists exactly one transition $(m, \varphi) \in \Gamma$ for which $\varphi[v_{\mathbf{p}}]$ is satisfiable.

For some fixed mapping $\kappa : Q \mapsto \mathbb{N}_0$, we define the semantics of an interface automaton $\mathcal{A} = \langle \Sigma, Q, q_0, q_\pi, \Gamma \rangle$ by means of the symbolic transition system $\mathcal{STS}_{\mathcal{A}} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$ where M has been fixed above, $\mathbf{x} = \{x_q\}$ contains a single variable for maintaining the current state, the initial condition q_0 is $(x_q = \kappa(q_0))$, and for every transition $(q, (m, \varphi), q') \in \Gamma$, we define transition $t = (m, \psi)$ in $\mathcal{T}$ with $\psi = (x_q = \kappa(q) \wedge \varphi \wedge x'_q = \kappa(q'))$. We add t to $\mathcal{T}_\pi$ if $q' = q_\pi$. A sequence of method invocations is safe in $\mathcal{A}$ if it is a safe trace of $\mathcal{STS}_{\mathcal{A}}$ and erroneous if it is an error trace of $\mathcal{STS}_{\mathcal{A}}$.

Example. Figure 2c shows the behavioral interface for the component from Figure 2. Formally, the interface is defined by the tuple $\mathcal{A}_{Ex} = \langle \Sigma, Q, q_0, q_\pi, \Gamma \rangle$ with $\Sigma = \{(move(p), (0 < p \wedge p < 20)), \ldots, (check(), true)\}$, states $Q = \{q_0, q_1, q_\pi\}$ and tranitions as in the figure. When generating a symbolic transition system from $\mathcal{A}_{Ex}$, the $(check(), true)$ transition from q_0 to q_π will be the basis for the only error transition.

Checking Components against Interfaces Since we are interested in precise interfaces, correctness means behavioral equivalence between the interface and the component, i.e., equality of sets of safe and error traces. Behavioral equivalence is checked on the composition of the corresponding transition systems, as described below.

For symbolic transition systems $\mathcal{STS}^1 = \langle M, \mathbf{x}^1, \mathcal{I}^1, \mathcal{T}^1, \mathcal{T}^1_\pi \rangle$ and $\mathcal{STS}^2 = \langle M, \mathbf{x}^2, \mathcal{I}^2, \mathcal{T}^2, \mathcal{T}^2_\pi \rangle$ over identical methods M and over disjoint sets of variables $(\mathbf{x}^1 \cap \mathbf{x}^2 = \emptyset)$, we define the product as

$$\mathcal{STS}^{1\times 2} = \langle M, \mathbf{x}^1 \cup \mathbf{x}^2, \mathcal{I}^1 \wedge \mathcal{I}^2, \mathcal{T}^{1\times 2}, \mathcal{T}^{1\times 2}_\pi \rangle$$

with

- $\mathcal{T}^{1\times 2} = \{(m, \varphi_1 \wedge \varphi_2) \ : \ ((m, \varphi_1), (m, \varphi_2)) \in \mathcal{T}^1 \times \mathcal{T}^2\}$, and
- $\mathcal{T}^{1\times 2}_\pi = \{(m, \varphi_1 \wedge \varphi_2) \ : \ ((m, \varphi_1), (m, \varphi_2)) \in ((\mathcal{T}^1 \setminus \mathcal{T}^1_\pi) \times \mathcal{T}^2_\pi) \cup (\mathcal{T}^1_\pi \times (\mathcal{T}^2 \setminus \mathcal{T}^2_\pi))\}$.

The composition $\mathcal{STS}^{1\times 2}$ has error traces if and only if symbolic transition systems $\mathcal{STS}^1$ and $\mathcal{STS}^2$ are not behaviorally equivalent.

4 Checking Component Interfaces

As discussed in the previous section, we check correctness of a candidate interface against a component implementation by searching for reachable error transitions

Algorithm 1 Bounded Exploration

Global: $\mathcal{STS} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$, depth bound k

```
1: procedure EXPAND( ((m_1, φ_1), ..., (m_i, φ_i)) )
2:     if i < k then
3:         for (m, φ) ∈ T do
4:             if spc(I · φ_1 · ... · φ_i · φ) satisfiable then
5:                 if (m, φ) ∈ T_π then
6:                     stop with error trace ((m_1, φ_1), ..., (m_i, φ_i), (m, φ))
7:                 else
8:                     EXPAND( ((m_1, φ_1), ..., (m_i, φ_i), (m, φ)) )
9:                 end if
10:            end if
11:        end for
12:    end if
13: end procedure
```

on the product of the component and the interface. For the remainder of this section, we fix $\mathcal{STS} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$ as this product. The baseline approach for validating interfaces that is implemented in PSYCO is exhaustive, depth- or time-bound symbolic exploration of $\mathcal{STS}$. In this mode, symbolic candidate traces from $\mathcal{T}^i$ are generated for growing i (up to some bound k) and checked for feasibility as shown in Algorithm 1. Exploration stops as soon as it finds a symbolic error trace. In [17], it is shown how information from a static analysis about reading and writing of state variables in methods can be used for a partial order reduction. This optimization is still available in PSYCO. In the remainder of this section, we present two newer extensions that optimize discharging conjectures.

4.1 Optimized Exploration through Interpolation

In order to expand fewer states during exploration, we over-approximate states reached by symbolic execution of method path sequences and under-approximate safe states with the help of interpolation. We give an example before presenting the modified exploration of $\mathcal{STS}$: On the component from Figure 2, after $\varphi_{m,4}$, path $\varphi_{c,2}$ of method `check` (which would lead to an error) is not executable and correspondingly $spc(\mathcal{I} \cdot \varphi_{m,1} \cdot \varphi_{c,2})$ is unsatisfiable. We can feed the pair

$$\left[\; (spc(\mathcal{I} \cdot \varphi_{m,1}) \quad , \quad wpc(\varphi_{c,2} \cdot true) \; \right]$$

to an interpolation constraint solver and will receive an interpolant that is implied by the left formula but is inconsistent with $\varphi_{c,2}$, and is only over variables shared by both formulas, e.g., $(0 < x \wedge x < 20)$. The interpolant over-approximates the states reached after $\varphi_{m,4}$ and under-approximates the safe states from which no immediate error can be triggered.

Figure 3 and Algorithm 2 show how we collect and propagate interpolants during symbolic exploration in order to stop expanding after transitions that

Algorithm 2 Optimized Exploration

Global: $\mathcal{STS} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$, bound k, safe states $\psi_0 = \ldots = \psi_{k-1} = false$

procedure ExpandItp($((m_1, \varphi_1), \ldots, (m_i, \varphi_i))$)
  if $i < k$ **then**
    if ($spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_i) \wedge \neg\psi_i$) satisfiable **then** ▷ Not in Cache?
      for $(m, \varphi) \in \mathcal{T}$ **do**
        if $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_i \cdot \varphi)$ satisfiable **then**
          if $(m, \varphi) \in \mathcal{T}_\pi$ **then**
            stop with error $((m_1, \varphi_1), \ldots, (m_i, \varphi_i), (m, \varphi))$
          else
            ExpandItp($((m_1, \varphi_1), \ldots, (m_i, \varphi_i), (m, \varphi))$)
          end if
        end if
      end for ▷ Update Cache
      if $i < k - 1$ **then**
        $\psi_i \leftarrow \psi_i \vee itp[\ spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_i),\ \bigvee_{(m,\varphi) \in (\mathcal{T} \setminus \mathcal{T}_\pi)} wpc(\varphi \cdot \neg\psi_{i+1})]$
      else
        $\psi_i \leftarrow \psi_i \vee itp[\ spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_i),\ \bigvee_{(m,\varphi) \in \mathcal{T}_\pi} (\varphi)\ ]$
      end if
    end if
  end if
end procedure

lead to known safe states: After exploring to depth k after sequence of transitions $((m_1, \varphi_1), \ldots, (m_i, \varphi_{k-1}))$ without finding error traces, we use the conditions of the error transitions for computing an interpolant $I[\mathbf{x}]$ that over-approximates $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_{k-1})$ (cf. bottom left of Figure 3). We propagate interpolants upwards in the tree by computing weakest preconditions for transitions and negated interpolants as indicated in the the figure: if $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_{k-1})$ and φ_k are inconsistent with interpolant I, then $spc(\mathcal{I} \cdot \varphi_1 \cdot \ldots \cdot \varphi_{k-2})$ is guaranteed to be inconsistent with $wpc(\varphi_{k-1} \cdot \neg I)$. Please note that, on one hand, negating I does not introduce universal quantification since I only has free variables $\mathbf{x}$. On the other hand, exploration still has to be depth-bounded since safe states are only computed w.r.t. some remaining depth of exploration.

4.2 Symbolic Search

In order to move from (bounded) exploration to search, we need to check if some sequence of paths reaches states that another sequence cannot reach. In our motivational example, e.g., in order to test if $spc(\varphi_{m,2}) \models_{\mathbb{L}} spc(\varphi_{m,4})$ as a basis for deciding if exploration can stop or if it has to expand $\varphi_{m,2}$. This can be done by checking if a model for $spc(\varphi_{m,2})$ exists that is not a model for $spc(\varphi_{m,4})$ and corresponds to checking the satisfiability of formula $spc(\varphi_{m,2}) \wedge \neg spc(\varphi_{m,4})$. Negation of $spc(\varphi_{m,4})$ requires universal quantification of variables private to

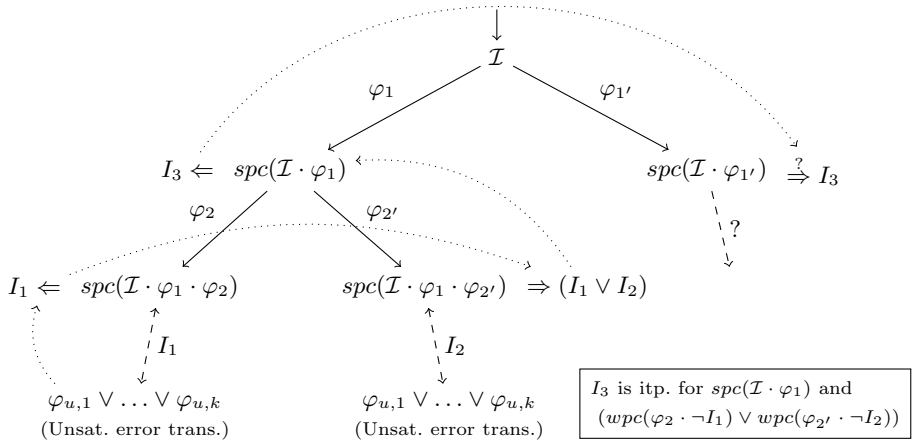

Fig. 3: Optimizing Symbolic Exploration in PSYCO based on Interpolation. Simplified schematic propagation of interpolants through symbolic exploration tree. In general, unsatisfiable error transitions occur on every level.

$spc(\varphi_{m,4})$, resulting in test

$$\big(\forall y_1 \forall p_1 \;.\; \neg(y_1 = 0 \;\wedge\; y_1 < 200 \;\wedge\; 0 < p_1 \;\wedge\; p_1 < 20 \;\wedge\; x = y_1 + p)\big) \;\wedge\; \big(y_{10} = 0 \;\wedge\; y_{10} < 200 \;\wedge\; 0 \geq p_{11} \;\wedge\; x = y_{10}\big)$$

In this particular case, it is necessary to explore after $spc(\varphi_{m,2})$: all models of $spc(\varphi_{m,2})$ imply $x = y_{10} = 0$ while there is no model for $spc(\varphi_{m,4})$ in which $x = 0$. Our symbolic search algorithm is based on checking entailment in this fashion. Algorithm 3 shows the resulting simple symbolic depth-first search algorithm. We used symbolic search for analyzing the depth of a component's state space in previous work [28]. On the product of component and interface, the algorithm enables PSYCO to discharge interfaces on component implementations.

5 Implementation and Evaluation

This section highlights some details of our implementation and presents results from an evaluation on a small set of benchmark components.

Implementation. The complete PSYCO[5] tool is implemented in Java and available as open source software on github. As pointed out in the introduction, PSYCO integrates different analysis methods with the aim of generating behavioral component interfaces. For active automata learning, PSYCO uses

[5] https://github.com/psycopaths/psyco

Algorithm 3 Symbolic Search

Global: $\mathcal{STS} = \langle M, \mathbf{x}, \mathcal{I}, \mathcal{T}, \mathcal{T}_\pi \rangle$, reached states $\psi_{Reach} \leftarrow \mathcal{I}$

```
1: procedure SEARCH( ((m_1, φ_1), ..., (m_i, φ_i)) )
2:     for (m, φ) ∈ T do
3:         ψ_Next ← spc(I · φ_1 · ... · φ_i · φ)
4:         if ψ_Next satisfiable then
5:             if (m, φ) ∈ T_π then
6:                 stop with error ((m_1, φ_1), ..., (m_i, φ_i), (m, φ))
7:             else
8:                 if (ψ_Next ∧ ∀y∀p.¬ψ_Reach) satisfiable then
9:                     ψ_Reach ← (ψ_Reach ∨ ψ_Next)
10:                    SEARCH( ((m_1, φ_1), ..., (m_i, φ_i), (m, φ)) )
11:                end if
12:            end if
13:        end if
14:    end for
15: end procedure
```

LearnLib [21] and AutomataLib[6], which also provides basic support for different automata transformations and visualization tasks. Internally, PSYCO works on a logic encoding of analyzed components, for which jConstraints[7] is used. The Z3 constraint solver [11] is used in most cases for deciding satisfiability of such encodings. The single exemption is construction of interpolants over linear integer arithmetic for state abstraction, which is done by SMTInterpol [7]. The symbolic search algorithm described in Section 4.2 is implemented in the jStateExplorer library.

Finally, PSYCO currently uses JDart [25] for symbolic execution. JDart allows conversion of execution paths through java methods into symbolic transition systems. However since verification and learning is completely language independent in PSYCO, it could be used for components in other languages than Java by replacing JDart with another tool for producing symbolic method summaries.

Evaluation. We evaluate four configurations of PSYCO. As a baseline, we use exhaustive depth-limited exploration as presented in [14]. We compare partial order reduction, as presented in [17], interpolation-based caching, and symbolic search, against the baseline. Our experiments have been conducted on a native 64 bit Linux host. The CPU is an i9-7960X combined with 128 GB RAM. The JVM was provided with 32 GB RAM. We run each experiment three times. We limit PSYCO runs regarding time and memory resources and execute different exhaustive depth checks during the run.

Results are summarized in Table 1. For all approaches, we report limits on runtime, sizes of final alphabet and model, $|\alpha M|$ and $|Q_I|$ respectively, and depth

[6] https://github.com/Learnlib/automatalib
[7] https://github.com/psycopaths/jconstraints

Example		Baseline (runtime: 1h)			Baseline + POR (runtime: 1h)			Interpolation (runtime: 1h)			Symbolic Search			
Name	$\|\mathcal{M}\|$	$\|\alpha M\|$	$\|Q_I\|$	k_{max}	$\|\alpha M\|$	$\|Q_I\|$	k_{max}	$\|\alpha M\|$	$\|Q_I\|$	k_{max}	$\|\alpha M\|$	$\|Q_I\|$	k_{max}	time
AltBit	3	7	6	160	7	6	162	-	-	n/a	8	6	256	63 min.
Stream	5	6	4	11	6	4	14	6	4	$> 272,000$	6	4	2	1 min.
Signature	6	6	5	11	6	5	12	6	5	$> 451,000$	6	5	2	1 min.
IntMath	8	11	3	10	11	3	13	-	-	n/a	11	3	1	1 min.
AccMeter	9	9	5	5	9	5	5	9	5	$> 9,200$	-	-	-	dnf
CEV	19	27	34	6	27	34	6	27	34	17^{*_1}	-	-	-	dnf
CEV V2	20	21	8	4	21	8	4	21	20	18^{*_2}	21	20	7	13 min.
Socket	49	57	42	5	57	42	5	57	42	7^{*_3}	57	42	8	19 min.
kmax1	3	4	8	11	4	8	11	4	9	109,821	4	9	12	1 min.
kmax2	4	9	7	5	9	7	5	-	-	n/a	9	8	6	1 min.

Table 1: Comparison between the existing PSYCO model checks and the newly introduces ones based on interpolation and symbolic search. n/a: no interpolants due to unsupported language features. $*_1$ This experiment hits the memory limit after 44 minutes. $*_2$ This experiment hits the memory limit after 23 minutes. $*_3$ This experiment hits the memory limit after 5 minutes.

k_{max} up to which a configuration was able to explore the product of component and model. For symbolic search we additionally report actual runtimes. Since symbolic search is the only method that terminates once the state space is explored completely, we do not impose a time limit - we let it complete or run out of resources. We report the smallest k_{max} and (for symbolic search) the maximum runtime from the three runs of an experiment.

The experiments show that interpolation and symbolic search clearly outperform the baseline and partial order reduction: symbolic search terminates within minutes in most cases and interpolation increases k_{max} by several orders of magnitude in some cases. However, it is not possible to apply both approaches to all benchmarks. One challenge for search are deep state spaces: The AltBit example, e.g., is implemented using a counter internally that is incremented with each call to this component. $k_{max} = 256$ represents a byte counter, that overflows exactly after 256 increments. While symbolic search runs to completion in this case, it requires 63 minutes. Cycling through deep state spaces becomes intractable on the bigger examples: The CEV experiment, e.g., has a bug that traverses the complete integer domain. When the bug is fixed (in CEV V2), both interpolation and search outperform the baseline approach: while bounded exploration with interpolation is able to explore up to to 2.5 times the depth of the baseline (and size of explored space grows exponentially with depth), symbolic search terminates after 13 minutes with a definitive verdict. We can observe something similar for the Socket example. Interpolation reaches one depth less than the k_{max} of the search before running out of memory but does this in half the time. Relaxing the memory limit to 64 GB, interpolation outperforms the search in depth for this example in less than 9 min.

Validation. In order to also have an external reference for assessing the efficiency of the implemented extensions, we submitted some of the smaller products of hypothesis model and component to SV-COMP[8]. Most of the SV-COMP participants in 2017 and 2018 were not able to solve these tasks, which provides some confidence that discharging interfaces on component implementations is not trivial.

6 Conclusions and Future Work

PSYCO is a tool for computing interfaces of components and to this end integrates multiple dynamic and static analysis methods. In this paper, we have presented two methods for discharging conjectured interfaces on component implementations (or vice versa). Bounded checking through symbolic exploration and optimized by interpolation-based caching of explored states is able to reach much greater depth than ordinary bounded exploration. Symbolic search terminates with a definitive verdict in most of our benchmarks. The basis for developing both extensions is a generic logic encoding of component behavior and interface. The main benefits of such an encoding are that (1) it allows to separate development of analysis techniques from generating formal representations, and (2) that it provides a lingua franca in which different analysis methods can exchange information.

The presented work brings us one step closer to the ultimate goals of using interfaces in compositional verification of properties on component-based systems and as contracts for third-party components. There are multiple intriguing directions for future research that are motivated by the work on PSYCO but also pertain to the challenge of integrating different program analysis paradigms in general:

Support for More Languages. Exchanging JDart by a symbolic execution engine for another programming language would allow to use the complete PSYCO approach without any modification on other languages as well. In theory, JDart might be replaced by a symbolic execution engine analyzing assembly or byte code of a virtual machine, enabling formal analysis of binary component realizations in domains that typically do not disclose their source code.

Adequate Representation. We have made the interesting observation when analyzing the main reason for the slow symbolic search performance on the AccMeter example: The logic encoding of states contains many math-heavy constraints. These constraints originate from parts of complex guards, but do not taint state variables. Based on this observation, we suspect that performance of symbolic search can be improved by pruning irrelevant parts of constraints. Moreover, alphabet refinement during learning and product generation before checking interfaces currently produce fairly complex logic constraints. Constraint

[8] https://sv-comp.sosy-lab.org/2018/results

simplification likely has a potential for improving performance and it will lead to models that are better comprehensible to human experts. While simplifying constraints may not be solvable efficiently in general, complex constraints arise in specific situations in PSYCO and have well-defined patterns, which may make the problem tractable.

Integration of More Analysis Paradigms. For application domains in which discharging interfaces against implementations is not possible (e.g., for thrid-party services in Web-applications) we plan to integrate runtime monitoring of interface-conformance on integrated services. This will require, among other things, the integration of a modern learning algorithm (e.g. TTT [20]) that perform well on potentially long sequences observed by a monitor. While LearnLib offers very efficient algorithms, PSYCO currently is based on the classic L^* algorithm. Integrating TTT is also expected to have a positive impact on performance in cases where interfaces can be discharged: Currently, PSYCO spends about 50% of runtime on automata learning and TTT has been demonstrated to achieve substantial savings in runtime over L^* [21].

References

1. 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.
2. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. In *International SPIN Workshop on Model Checking of Software*, pages 146–162. Springer, 2006.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.
6. G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.*, 22(3):25:1–25:46, July 2013.
7. J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, pages 248–254, 2012.
8. E. Clarke, D. Kroening, and F. Lerda. *A Tool for Checking ANSI-C Programs*, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
9. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.
10. W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

11. 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*, pages 337–340. Springer, 2008.
12. Y. Ge and L. De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*, pages 306–320. Springer, 2009.
13. 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.
14. D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In *SAS '12*, pages 248–264, 2012.
15. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.
16. 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.
17. F. Howar, D. Giannakopoulou, and Z. Rakamaric. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In *ISSTA 2013*, pages 268–279, 2013.
18. F. Howar, T. Kahsai, A. Gurfinkel, and C. Tinelli. Trusting outsourced components in flight critical systems. In *AIAA Infotech @ Aerospace*. AIAA, 2015.
19. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 263–277. Springer, 2011.
20. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - 5th International Conference, RV 2014*, pages 307–322, 2014.
21. M. Isberner, F. Howar, and B. Steffen. The open-source learnlib - A framework for active automata learning. In *CAV 2015, Part I*, pages 487–495, 2015.
22. J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 758–766, 2012.
23. J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 396–411, 2011.
24. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 454–469, 2009.
25. K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. Jdart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer, 2016.
26. K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *International Conference on Computer Aided Verification*, pages 250–264. Springer, 2002.

27. K. L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 104–118, 2010.
28. M. Mues, F. Howar, K. S. Luckow, T. Kahsai, and Z. Rakamaric. Releasing the PSYCO: using symbolic search in interface generation for java. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.
29. 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.
30. C. M. Wintersteiger, Y. Hamadi, and L. De Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.