Releasing the PSYCO: Using Symbolic Search in Interface Generation for Java*

Malte Mues
Clausthal University of Technology, Germany
malte.mues@tu-clausthal.de

Falk Howar
Clausthal University of Technology, Germany
falk.howar@tu-clausthal.de

Kasper Luckow
Carnegie Mellon University Silicon Valley, Mountain View, CA, USA
kasper.luckow@sv.cmu.edu

Temesghen Kahsai
NASA Ames Research Center, Moffett Field, CA, USA
teme.kahsai@sv.cmu.edu

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

ABSTRACT
The Java PathFinder extension Psyco generates interfaces of Java components using a combination of dynamic symbolic execution and automata learning to explore different combinations of method invocations on a component. Such interfaces are useful in contract-based compositional verification of component-based systems. Psyco relies on testing for validating learned interfaces and currently cannot guarantee that a generated interface is correct. Instead, it simply returns the most recent learned interface once a user-defined time limit is exceeded. In this paper, we report on work that was performed during the 2016 Google Summer of Code. The aim of this work is to extend Psyco with symbolic search. During symbolic search, Psyco uses fully symbolic method summaries for exploring the state space of a component symbolically. We plan to eventually use symbolic search to compute a termination criterion for Psyco that guarantees the correctness of learned interfaces (e.g., by using symbolic search as a basis for symbolically model-checking a component against a learned interface).

CCS Concepts
•**Software and its engineering** → **Software verification; State systems;** *Search-based software engineering;* Requirements analysis; •**Theory of computation** → Verification by model checking;

General Terms
Algorithms, Verification

Keywords
Symbolic Execution, Active Learning, Model Generation, Symbolic Search

1. INTRODUCTION
In the context of a NASA-funded project we are interested in developing an automated framework for the generation of assume-guarantee-style formal contracts for components of flight-critical systems [12]. This endeavor is motivated by a need for scaling the formal analysis effort on component-based flight-critical systems to the level of complexity found in industrial-scale systems. The design and implementation of software systems used in aviation are often contracted out to external companies. For example, the FAA rarely develops its air traffic software internally; it usually acquires it from contractors who develop new systems in accordance with the FAA's requirements. The delivered products usually do not include intermediate artifacts such as design models or source code, which would allow the FAA to take advantage of advanced verification techniques (e.g., formal verification methods). As a consequence, the only means of verifying these external components is black-box testing, which provides no formal guarantees.

Technically, our approach is centered on *contract-based compositional verification*, where contracts are elicited automatically from component design models, prototype implementations, and system-level properties. The proposed approach is based on techniques developed in the area of *automata learning*, *invariant generation*, *model checking*, and *automated assume/guarantee reasoning*. One of the cornerstones of this project is the generation of component interfaces (i.e., contracts) from prototypical implementations of components. These contracts become part of the component specification and serve as a basis for testing black-box 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. A formal definition of components can be found in [11]. The Java PathFinder extension Psyco [10, 11] generates interfaces of Java components using a combination of dynamic symbolic execution and automata learning to explore different combinations of method invocations on a component. Such interfaces are useful in contract-based compositional verification of component-based systems.

The current version of Psyco iterates between two modes of operation: generating conjectured interfaces (using automata learning and dynamic symbolic execution) and validating interfaces (using model-based testing and dynamic symbolic execution). Since Psyco relies on testing for validating conjectured interfaces, it currently cannot guarantee that a generated interface is correct. Instead, it simply returns the most recent learned interface once a user-defined time limit is exceeded during testing. Concretely, Giannakopoulou et al. [10], define an interface to be k-full if it is correct (i.e., safe, permissive, and tight) for all method sequences of length up to $k \in \mathbb{N}$. Correspondingly, validation of interfaces is done by checking k-fullness for increasing k. Currently, Psyco terminates once an interface was proven full for a fix k or when runtime exceeds a predefined limit. In [11], Howar et al. were able

*This material is based upon work partially funded and sponsored by NASA Contract No. NNX14AI09G and NSF Award No. 1422705

```
class Example {
  int x = 0;
  void setX( int p ) {
    if ( 0 < p && p < 200 ) {
      x = p;
    } else {
      assert false;
    }
  }
}
```

Figure 1: Example Java class.

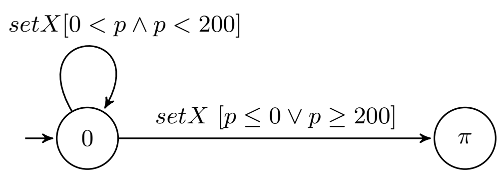

Figure 2: Interface for Class Example.

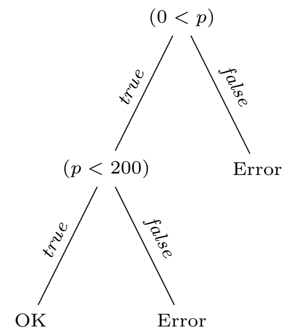

Figure 3: Constraints tree produced by symbolic execution of method setX(int p) of class Example. Leafs labeled 'Error' correspond to executions that ended with assertion violations. The path to the leaf labeled 'OK' represents a successful execution.

to define a termination criterion for a limited set of cases (based of enumerating concrete states reached during checking k-fullness).

In this paper, we report on ongoing work that was performed during the 2016 Google Summer of Code. The aim of this work is to extend PSYCO with symbolic search. During symbolic search, PSYCO explores the state space of a component symbolically in breath-first fashion until exhaustion. We use the JDART [14] dynamic symbolic execution extension of Java PathFinder to produce fully symbolic summaries of methods of an analyzed white-box component. These summaries can then be used for computing a symbolic transition system of the component. Symbolic exploration of this transition system (using a variant of the symbolic search algorithm presented in [1]) allows us to use PSYCO to symbolically model-check component implementations for errors, e.g., assertion violations. In a second step (after Google Summer of Code) we plan to use symbolic search as a basis for symbolically model-checking a component against a learned interface in order to determine the correctness of the interface and decide termination in PSYCO. We evaluate the symbolic search on a set of Java components that have served as benchmarks in previous works on PSYCO.

Please note that since PSYCO operates only on a symbolic representation of a component, it can easily be extended to white-box components in other languages and maybe even to binaries. Symbolic summaries of component methods could, e.g., be obtained by using KLEE [4] for components written in C. Replacing JDART in the processing chain by a tool that transforms compiled components into a symbolic transition system allows using PSYCO also on black-box components. BAP [3] might be a candidate that can be adapted for this.

Related Work. Describing program states symbolically by Boolean formula and using transitions between states as edges yields a graph on which Breath-first search (BFS) can be applied. This is described, for example, by Edelkamp et al. [7] or Alur [1]. Edelkamp just mentions the symbolic BFS in a few words and continues directly with binary decision diagrams (BDDs) which are an established solution solving the search on huge state spaces. Alur provides more details for the symbolic BFS and describes a variant using existential quantification to simplify state descriptions after each application of the transition system. We use Alur's version as theoretical baseline for our implementation.

Farzan et al. [9] have introduce JavaFAN a tool adapt Java source code to the maude LTL model checker [8] which applies BFS on finite state systems to explore them completely. Maude uses term rewriting to do so but does not expose the state model and guides the search to verify occurrences of a single error, so it is not suitable for PSYCO purpose. Within the Java PathFinder ecosystem JPF-Statechart has been introduced by Mehlitz [15] to verify UML state charts translating them to Java and exploring the state machines by executing them step by step until all paths are explored. Contrary to our implementation JPF-Statechart does not benefit from symbolic state descriptions and therefore has to enumerate concrete states which can lead to state explosion quickly.

De Caso et al. [5] target the generation of permissive behavioral models in their recent work. The generated models are similar to the interfaces generated by PSYCO. Their scenario is slightly different from ours: While they are interested in errors that can occur due to memory management in systems with dynamic memory allocation, we target components in embedded and safety critical systems where this usually is not an issue as dynamic allocation of memory is avoided.

A discussion of other approaches generating full interfaces can be found in the previous work regarding PSYCO [10, 11]. To the best of our knowledge, there is no other approach using symbolic search during interface generation.

2. PRELIMINARIES

In this section, we briefly describe how PSYCO generates interfaces of JAVA components and how symbolic search can help deciding when to stop PSYCO. We base our presentation on a simple example. We refer to previous work [10, 11], for a more in-depth description of the PSYCO algorithms.

PSYCO generates temporal interfaces for components that include methods with parameters. We illustrate how PSYCO works on the `Example` class shown in Figure 1. The class has one member `x`, which is initialized to 0. There is one method `setX(int p)` which will set `x` to `p` if `p` is within certain bounds and otherwise fail with an assertion violation.

The interfaces generated by PSYCO are finite-state automata whose

Algorithm 1 Symbolic Search

Input: Initial State $Init$, Transitions $Trans$
Output: Max. required symbolic exploration depth k

$Reach \leftarrow Init$
$New \leftarrow Reach$
$k \leftarrow 0$
while $New \neq \emptyset$ **do** ▷ Check for Emptiness
  $k \leftarrow k + 1$
  $Next \leftarrow Post(New,\ Trans)$
  $New \leftarrow Next \setminus Reach$ ▷ Compute Difference
  $Reach \leftarrow Reach \cup New$
end while
return k

$$Paths = \begin{cases} (\ Error,\ \ (p \leq 0)\ ) \\ (\ Error,\ \ (p \geq 200)\ ) \\ (\ OK,\ \ (0 < p) \wedge (p < 200)\ \ \wedge\ \ (x' = p)\ ) \end{cases}$$

Figure 4: Fully symbolic summary of method setX(int p) of class Example. Primed variables denote updated values. Every path is a pair of execution result and path condition including post conditions on class members.

transitions are labeled with method names and guarded with constraints on the corresponding method parameters. The guards partition the input spaces of parameters, and enable a more precise characterization of legal orderings than was previously possible in a fully automatic fashion. Figure 2 shows the interface for the `Example` class shown in Figure 1. The interface has two states, the initial state 0, and an error state π. Transitions are labeled by $setX$ (the name of the only method) and guards. In this simple example, the guard corresponds exactly to the condition of the `if`-statement in the `setX` method.

Psyco uses automata learning [2, 13] to create sequences of calls to methods of a component; these sequences are then turned into programs. JDart [14] is used to analyze these programs.[1] From the resulting path constraints, Psyco extracts guards for transitions in the generated interfaces. In our example, the most trivial example for such a sequence would be a single invocation of `setX(int p)` with a symbolic parameter p. This would yield two error paths and one successful path as shown in Figure 3.

After running a number of these generated programs, the automata learning algorithm used by Psyco produces a conjecture for the interface of the component. The learning algorithm can not decide if the conjectured interface is correct, it merely guarantees that it is the most concise interface consistent with the observations made during running the programs.

Psyco uses a combination of dynamic symbolic execution and model-based testing for validating the correctness of the interface: it uses the interface as a basis for generating sequences of guarded methods. These sequences are translated into programs with test oracles (based on the interface). The programs are then analyzed with JDart until a difference between the behavior of the program and the interface is found. Such a difference (a so-called counterexample) can be exploited by the learning algorithm for refining the conjectured interface. The method sequence $setX(p_1)$ $setX(p_2)$ with constraints $(0 < p_1 \wedge p_1 < 200)$ and $(p_2 \leq 0 \vee p_2 \geq 200)$ on symbolic values p_1 and p_2 is an example for such a test. In this case, the expectation would be that all paths explored by JDart end in assertion failures.

Test programs are generated in a structured fashion from the conjectured interface: Starting with length $k = 1$, all sequences of guarded methods of length k are generated and checked. Then, k is increased by 1 and sequences of this length are checked. In the current version of Psyco, there is no facility for deciding when enough tests were generated from the interface. Psyco instead requires the user to specify a limit on the runtime or on k. Once this limit is reached, Psyco terminates and returns the latest conjecture as interface along with values k_{min} and k_{max}. There, k_{min} is the value of k at which the last counterexample was found and k_{max} is the greatest length k for which all sequences of guarded methods were tested.

In this work, we aim at improving this last part of Psyco, the validation of the interface. The basic idea is that we can unroll the transition system of a component using symbolic search. When symbolic search terminates, we know that no new states can be reached in the component. We plan to use this symbolic exploration of the state space as a basis for symbolically checking the correctness of a learned interface. We detail our implementation of symbolic search in the next section.

3. SYMBOLIC SEARCH

In this section, we present our modified version of symbolic search on a symbolic transition system. As discussed in the previous section, Psyco relies on the dynamic symbolic execution engine JDart for extracting such symbolic transition systems from white-box Java components. Please note that the symbolic transition systems we target here model the implemented internal behavior of Java components and not the interfaces generated by Psyco.

Our implementation of symbolic search is based on the algorithm that is presented in Alur's textbook on the principles of cyber-physical systems [1]. We modified parts of the algorithm to improve performance in our concrete scenario and with the concrete constraint solver (Z3 [6]) we use. For example, we replaced quantifier elimination in one step of the original algorithm by tests over quantified formulas which are better supported in Z3 than is quantifier elimination (at least for our use cases).

Algorithm 1 shows the pseudo-code of our symbolic search at the abstract level of sets of states. The algorithm takes as input a transition system, consisting of an initial condition $Init$ and a set of transitions $Trans$. The algorithm maintains the current depth of exploration k, a symbolic characterization of the set of reachable states ($Reach$), and a symbolic representation of states that became reachable in the previous iteration (New).

In every iteration the algorithm first checks if the set of newly discovered states is empty (i.e., if New is unsatisfiable). If there are no new reachable states at some point, the algorithm terminates and returns k as the depth required for complete symbolic exploration. In case that there are new states, the algorithm increases k and then computes the set $Next$ of states that can be reached from New by applying $Trans$. Following this, the set of newly reachable states New is computed as the difference between $Next$ and $Reach$.

Finally, these states are added to the set of reachable states (as

[1] Psyco additionally provides an optimized mode in which JDart is only used for computing fully symbolic method summaries once. Subsequent calls to JDart are then simulated using these summaries.

Example		X-Psyco (runtime: 1h)						Symbolic Search		Errors
Name	$\|\mathcal{M}\|$	$\|\alpha M\|$	$\|Q_I\|$	k_{min}	k_{max}	k_{full}	k_{sym}	Emptiness of New [ms]	Runtime [ms]	Reachable/Code
AltBit	3	6	6	4	298	d/k	d/k	937	21,202	4 / 4
Stream	5	6	4	1	54	d/k	2	17	146	4 / 4
Signature	6	6	5	1	2	2	2	6	97	3 / 3
IntMath	8	9	3	1	1	1	1	0	522	263 / 263
AccMeter	9	12	8	2	7	d/k	d/k	16,394	16,893	23 / 23
CEV	19	27	34	5	16	d/k	d/k	42,846	48,245	32 / 32
CEV V2	20	21	8	2	4	d/k	7	5,215	20,120	75 / 251
Socket	55	56	42	2	4	d/k	8	132,971	162,393	52 / 60

Table 1: Preliminary experimental results. Left half of the table reports results of running X-Psyco (from [11]). $|\mathcal{M}|$ is the number of component methods (and also the size of the initial alphabet); k_{min} the value of k at which the final interface gets generated; k_{max} the maximum value of k explored (i.e., the generated interface is k_{max}-full); k_{full} is the value of k at which a correct interface was inferred provably. Right half reports result of running the symbolic search. k_{sym} is the k at which symbolic exploration was complete. Errors compares reachable error paths to error paths in the code.

disjunction of the symbolic formulas for *Reach* and New). Working with logic formulas, the computation of New requires the negation of *Reach*. Since the formulas we work with are over Integers and Reals and contain variables that encode method parameters, this step introduces quantifiers. These quantifiers can either be removed through quantifier elimination, or the underlying constraint solver has to be able to deal with quantified formulas. In our experiments, Z3 performed better on quantified formulas than with quantifier elimination.

We demonstrate a run of the algorithm on the component $Example$ from Figure 1. In a first step, we use JDart to produce fully symbolic summaries of methods of Example, shown in Figure 4. The execution paths now contain symbolic information about class member x as well.

From these summaries and the concrete initial state of Example (which can be obtained from JDart as well), we generate the transition system shown below. We introduce a Boolean variable err that encodes if an error was reached on some path.

$$\begin{aligned} Init &:= (x = 0 \ \wedge \ \neg err) \\ Trans &:= (\neg err \wedge p \leq 0 \ \wedge \ err') \ \vee \\ & \quad\ (\neg err \wedge p \geq 200 \ \wedge \ err') \ \vee \\ & \quad\ (\neg err \wedge 0 < p \wedge p < 200 \ \wedge \ x' = p \wedge \neg err') \end{aligned}$$

We initialize *Reach* and New to $Init$ (see $Init$ definition above). Since New is not empty, we enter the while-loop in line 4 of algorithm 1 and apply all possible paths for `setX(int p)` (see Figure 4) calculating $post(New, Trans)$. After renaming of updated

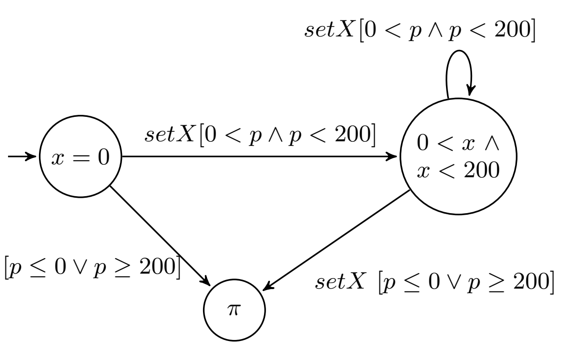

Figure 5: Symbolic State Space of Class Example.

variables, this yields the term $(err) \vee (\neg err \wedge x = p_1 \wedge 0 < p_1 \wedge p_1 < 200)$ as $Next$. The encoded states are new entirely and we extend *Reach* accordingly. In the next round of the algorithm, however, no new states are found — only the encoding of $Next$ becomes more complex. The symbolic search terminates with $k = 2$.

Figure 5 shows the symbolic state space of the component that has been explored. From the initial state ($x = 0$), two states are reachable: the error state π and the state where the value of x is between 0 and 200.

The symbolic search was implemented in Psyco as part of a 2016 Google Summer of Code project and is available as open-source software.[2]. In a next step (after Google Summer of Code), we will implement a synchronized symbolic breath-first search on component and learned interface. This will allow us to determine if a learned interface is correct and produce counterexamples for the learning algorithm in Psyco otherwise. Model checking will mark the transition from learned to generated interfaces.

4. PRELIMINARY EVALUATION

In this section, we report on preliminary findings from using symbolic search for symbolically exploring the state space of examples used in previous evaluations of Psyco. We have run our new implementation of the symbolic search for a first evaluation in a virtual Ubuntu x64 machine with 2 virtual cores and 4GB ram which is hosted on a windows 7 machine running on an Intel Core i7-2720QM CPU and 8GB ram. Each run is executed three times and the results are arithmetically averaged about this three runs for timing values. All non timing values are identical in each run.

We used the known examples from X-Psyco (see [11]) to evaluate the search and tried to find for as many example as possible a termination criterion. Our results are summarized in Table 1. In five out of our eight examples, we had success and the search found a fix point. However, for three of the examples symbolic search did not terminate before running out of resources. We then investigated the extracted transition systems of these three examples further in order to understand why they do not terminate. AltBit uses a counter internally, that is not reset. So the search will explore the complete Integer space for this counter until an overflow error occurs. AccMeter has also an infinite state space due to the internal calculation procedure of next state. The third example that is not finite is the implementation of the crew evac-

[2]`https://github.com/psycopaths/psyco`

uation vehicle (CEV). We compared the implementation to the original state machine provided by Mehlitz in [15] and discovered a bug in the Java implementation.

Regarding runtime, symbolic exploration was able to terminate within minutes on the examples where it terminated — X-Psyco in comparison ran for one hour. In most cases checking satisfiability of *New* requires a significant or even dominating fraction of the overall runtime. Overall, the measured runtimes make us confident that a synchronized exploration of component and interface will also be much more efficient than the testing done by Psyco and X-Psyco.

Finally, the evaluation shows that symbolic exploration can already be used in Psyco for checking reachability of errors in components without generating interfaces.

5. CONCLUSION AND FUTURE WORK

In this paper, we have reported on work that was performed during the 2016 Google Summer of Code. The aim of this work was to extend Psyco with symbolic search. During symbolic search, Psyco explores the state space of a component symbolically. We have implemented symbolic search and have evaluated its efficiency in a small series of experiments. In a next step we will use symbolic search as a basis for checking the correctness of interfaces generated with Psyco. Moreover, we want to investigate whether it is possible to reliably detect infinite transition systems on which search would not terminate.

6. REFERENCES

[1] R. Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] 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.

[6] 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.

[7] S. Edelkamp and S. Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[8] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[9] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of java programs in javafan. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[10] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 248–264, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 268–279, New York, NY, USA, 2013. ACM.

[12] F. Howar, T. Kahsai, A. Gurfinkel, and C. Tinelli. Trusting outsourced components in flight critical systems. In *AIAA Infotech @ Aerospace*. AIAA, 2015.

[13] 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. (Best Artifact Award).

[14] 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.

[15] P. C. Mehlitz. Trust your model - verifying aerospace system models with java pathfinder. In *In Proc IEEE Aerospace*, pages 1–11, 2008.