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

Malte Mues, Falk Howar, Kasper Luckow
Clausthal University of Technology, Germany
clausthal.de falk.howar@tu-clausthal.de kasper.luckow@sv.cmu.edu

Temesghen Kahsai, Zvonimir Rakamarić
NASA Ames Research Center, Moffett Field, CA, USA
teme.kahsai@sv.cmu.edu 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 ∈ 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
Psyco search algorithm presented in \cite{1}) allows us to use box component. These summaries can then be used for computing 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 breadth-first fashion until exhaustion. We use the JDart \cite{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 \cite{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 \cite{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 \cite{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. \cite{7} or Alur \cite{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. \cite{9} have introduce JavaFAN a tool adapt Java source code to the maude LTL model checker \cite{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 \cite{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. \cite{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 \cite{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 \cite{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
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 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...
Table 1: Preliminary experimental results. Left half of the table reports results of running X-Psyco (from [11]). \(|M|\) is the number of component methods (and also the size of the initial alphabet); \(k_{\text{min}}\) the value of \(k\) at which the final interface gets generated; \(k_{\text{max}}\) the maximum value of \(k\) explored (i.e., the generated interface is \(k_{\text{max}}\)-full); \(k_{\text{full}}\) is the value of \(k\) at which a correct interface was inferred provably. Right half reports result of running the symbolic search. \(k_{\text{sym}}\) is the \(k\) at which symbolic exploration was complete. Errors compares reachable error paths to error paths in the code.

| Example   | \(|M|\) | \(|M|\) | \(k_{\text{min}}\) | \(k_{\text{max}}\) | \(k_{\text{full}}\) | \(k_{\text{sym}}\) | Symbolic Search | Errors |
|-----------|--------|--------|-------------------|-------------------|-------------------|----------------|-----------------|-------|
| AltBit    | 3      | 6      | 6                 | 298               | d/k               | d/k            | 937            | 21,202 | 4 / 4          |
| Stream    | 5      | 6      | 4                 | 54                | d/k               | 2              | 17             | 146   | 4 / 4          |
| Signature | 6      | 6      | 5                 | 1                 | 2                 | 2              | 6              | 97    | 3 / 3          |
| IntMath   | 8      | 9      | 3                 | 1                 | 1                 | 1              | 0              | 522   | 263 / 263      |
| AccMeter  | 9      | 12     | 8                 | 2                 | d/k               | d/k            | 16,394         | 16,893 | 23 / 23        |
| CEV       | 19     | 27     | 34                | 5                 | 16                | d/k            | 42,846         | 48,245 | 32 / 32        |
| CEV V2    | 20     | 21     | 8                 | 2                 | 4                 | d/k            | 5,215          | 20,120 | 75 / 251       |
| Socket    | 55     | 56     | 42                | 2                 | 4                 | d/k            | 132,971        | 162,393 | 52 / 60        |

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-
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