Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference

Ivo Krka†, Yury Brun†, Daniel Popescu†, Joshua Garcia‡, and Nenad Medvidovic†
†Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{krka, dpopescu, joshuaga, neno}@usc.edu
‡Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
brun@cs.washington.edu

ABSTRACT

Software behavioral models have proven useful for design, validation, verification, and maintenance. However, existing approaches for deriving such models sometimes overgeneralize what behavior is legal. We outline a novel approach that utilizes inferred likely program invariants and method invocation sequences to obtain an object-level model that describes legal execution sequences. The key insight is using program invariants to identify similar states in the sequences. We exemplify how our approach improves upon certain aspects of the state-of-the-art FSA-inference techniques.

Categories and Subject Descriptors
D.2 [Software Engineering]: Testing and Debugging

General Terms
Algorithms, Verification

Keywords
dynamic analysis, likely invariants, specification mining

1. INTRODUCTION

Behavioral software specifications play an integral role in software design, validation, verification, testing, and maintenance [17]. Behavioral models, such as statecharts [8], allow early system specification, design-decision evaluation, and prototype development. Inferred behavioral models can help verify the implementation [9] and guide test-case generation to improve test coverage [7, 14]. In software maintenance, behavioral models help engineers understand complex objects and their interactions [13], as well as support debugging, fault detection, and detecting other anomalies [6, 18].

The outlined usage scenarios have motivated the development of a variety of methods for inferring behavioral specifications from system executions. The most relevant existing methods can be classified into two major categories: (1) inferring likely state invariants (e.g., [3, 4]) and (2) inferring finite-state automata (FSA)-based behavioral models (e.g., [2, 15]). The state invariant methods suggest likely values of system variables at particular execution points (e.g., they provide a method’s pre- and postconditions). These invariants illustrate “snapshots” of internal system state and do not explicitly capture the interactions and sequences of invocations. The FSA approaches aim to comprehensively describe method execution sequences from a set of dynamic method invocation traces. A major obstacle in this task is that the execution traces provide a highly detailed but limited subset of the possibly infinite set of legal executions. Hence, these FSA approaches generalize from the observed executions by merging states that exhibit similar behavior. Most existing approaches for FSA inference are based on the kTail algorithm [1], which merges states based on the similarity of the next k invocations in the trace. While automating such generalizations is highly desirable for complex software systems [5], FSA inference for such systems can produce imprecise, overgeneralized models containing many spurious behaviors [12].

Recently, Lo et al. [11] enhanced the kTail algorithm by incorporating state information. They proposed first inferring temporal properties that generally hold for the dynamic traces, and then merging states, while ensuring that the merge does not violate the inferred properties. However, this approach only considers the observed execution sequences and does not consider internal state information. We posit that the quality of the generated behavioral models can be further improved by considering information about the state of internal class variables. In this paper, we propose a novel approach that uses not only execution traces but also dynamically inferred program invariants. This enables us to overcome the aforementioned shortcomings of the two classes of behavioral specification-inference techniques. Our approach is fully automated and requires only compiled executables.

2. BACKGROUND

In this section, we describe StackAr, a data structure we will use as a running example throughout this paper, and two program analysis artifacts: invariants generated by Daikon [4] and method invocation traces. We then describe an FSA generated using today’s state-of-the-art techniques and highlight the flaws in that FSA that our approach corrects by leveraging program invariants.

StackAr is an array implementation of a stack. Created with a certain capacity, StackAr has seven public meth-
Figure 1: Three example StackAr method invocation traces.

3. APPROACH

In this section, we outline our technique, which overcomes the shortcomings of both, program invariant inference and FSA-based dynamic behavioral model synthesis. Our proposed technique has two phases. In phase 1, we derive an FSA that captures legal invocation sequences of an object's public interface based on the inferred data-value invariants. The primary idea behind phase 1 is that such an FSA can simulate the dynamic method invocation traces, while allowing us to gain more knowledge about the internal object states at particular points in those traces. In phase 2, we use the collected dynamic invocation traces to refine the invariant-based FSA into an object-level FSA that is both more general than the dynamic invocation traces and more precise than the invariant-based FSA.

3.1 Generating an Invariant-Based FSA

The goal of the first phase of our approach is to create an object-level FSA that describes all legal invocation sequences — those sequences that do not violate the methods' inferred likely invariants. We first create the FSA states based on a set of $p$ first-order predicates observed by Daikon (we only use the predicates defined on boolean and numerical variables). For example, for StackAr, Daikon reports $p = 4$ predicates: $P_1 = (topOfStack = -1), P_2 = (topOfStack \geq 0), P_3 = (topOfStack < size(theArray) - 1)$, and $P_4 = (topOfStack = size(theArray) - 1)$. The set of FSA states includes a special initial state and up to $2^p$ possible combinations of the $p$ predicates. While the largest theoretical state space for StackAr is $1 + 2^4 = 17$ states, our FSA has only four states (plus the initial state) due to predicate interdependence (e.g., $P_1$ and $P_2$ cannot be simultaneously true). We summarize our solution for dealing with such predicate interdependence below.

We next determine the legal transitions among the states, based on the inferred pre- and postconditions. To that end, we leverage our previous efforts [10]: an algorithm that syn-
we hypothesize that modern SMT solvers can efficiently off-the-shelf Satisfiability Modulo Theories (SMT) solvers. To enable construction of the invariant-based FSA, we utilize StackAr. Note that while the largest theoretical number of non-constructor transitions with 4 states and 6 invocations is 96, some such transitions are illegal (e.g., push on a full stack), so our FSA contains only 27 legal transitions.

For complex objects, the state space can get rather large. To enable construction of the invariant-based FSA, we utilize off-the-shelf Satisfiability Modulo Theories (SMT) solvers. We hypothesize that modern SMT solvers can efficiently decide whether a candidate state's predicate value assignment is satisfiable, and whether an invocation's invariants are satisfiable for the (source, destination) state pair. To test this hypothesis, we extended the implementation of our synthesis algorithm [10] to use the Yices SMT solver [19] when checking formula satisfiability. Our initial experience demonstrated that we can construct an invariant-based FSA almost instantaneously from StackAr invariants, as well as slightly more-complex object invariants. In our future work, we plan to further test the performance of FSA construction, and explore additional opportunities for optimizing and reducing the number of performed satisfiability checks.

3.2 Adding Invocation Traces

The second phase of our approach is concerned with combining the information about (1) inferred data invariants captured with the invariant-based FSA, and (2) the collected dynamic invocation traces. Since the invariant-based FSA can be overly general and imprecise, and the dynamic traces can be overly specific, we argue that combining the two in an intelligent manner can result in a general yet precise behavioral model. We suggest two promising strategies for coupling these different types of information. In Section 3.2.1, we propose enhancing the kTail algorithm by simulating the dynamic trace on the invariant-based FSA and applying an additional merging criterion involving the simulation results. In Section 3.2.2, we propose refining the FSA with the behavior captured in the dynamic traces. The latter strategy is based on our previous algorithm that synthesizes a partial-behavior model from a component’s method invariants and positive execution scenarios [10].

3.2.1 Enhancing the kTail Algorithm

In Section 2, we showed how the popular kTail algorithm can produce imprecise models due to spurious state merges. Lo et al. [11] demonstrated that including some state information can help produce more-precise models by avoiding undesirable merges. Our work is based on the premise that meaningful state information can be obtained from the inferred program invariants, which we accumulate with the invariant-based FSA. We propose simulating the dynamic traces on the invariant-based FSA and recording the traversed states. Formally, a sequence of invariant-based FSA state transitions $S_0 \xrightarrow{l_0} S_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} S_n \xrightarrow{l_n} S_{n+1}$ simulates a trace $T = M_0 \rightarrow M_1 \rightarrow \ldots \rightarrow M_n$, if $l_i = M_i$ for each $i \in [0, n]$. We then use the simulation results as an additional state-merging criterion: two states in a dynamic invocation trace can be merged when they have identical $k$-tails and are simulated with the same state (set of states) in the invariant-based FSA. For example, the simulation of Trace 1 from Figure 1 on the StackAr's invariant-based FSA from Figure 2 exposes that Trace 1’s state $S_0$ corresponds to StackAr invariant-based FSA’s $S_0$, while Trace 1’s states $S_1$–$S_{10}$ pertain to the FSA state $S_4$. Furthermore, Trace 3’s state $S_{15}$ corresponds to the invariant-based FSA’s $S_{15}$; consequently, the spurious merge of Trace 1 and Trace 3 states, performed by the other kTail algorithms, would be avoided.

The availability of the likely program invariants drives additional optimizations, such as identification of side-effect-free invocations (e.g., StackAr’s isEmpty and top). Hence, the states connected with transitions corresponding to side-effect-free invocations can be merged, turning the transitions into self-transitions. For example, states $S_1$, $S_2$, and $S_3$ in Trace 3 can be merged into a state with self-transitions on isEmpty and top. Yet another postulated benefit of using state information in the kTail algorithm is achieving high-quality results while using a smaller $k$ (in the basic kTail, a smaller $k$ results in more spurious merges). Note that our approach can similarly enhance the kHead algorithm.

3.2.2 Refining the Invariant-based FSA

As an alternative to model generalization from dynamic traces, we propose refining the derived invariant-based FSA and making it more specific to the traces at hand. The refinement process is similar to the algorithm we proposed in [10], where we refine an initial partial-behavior model, which is based on the method pre- and postconditions, with the behavior depicted in positive example executions (scenarios). The high-level idea is to take an initial FSA model (from Section 3.1), which contains only maybe transitions (i.e., behavior that does not violate the invariants, but has also not yet been confirmed with an execution sequence), and to refine the maybe transitions traversed when simulating the invocation traces on the FSA into required transitions.

To achieve this, we would first simulate all of the dynamic traces on the invariant-based FSA. Next, we refine a state traversed with the trace into two new states. The first state would have incoming transitions labeled with the invocation $I_t$ that lead into the state being refined, while the outgoing transitions would remain the same as those of the original state. Additionally, the transition corresponding to invocation $I_t$ would be refined to a required transition. The second state would account for all of the original state’s incoming transitions that are not labeled with invocation $I_t$ and for all the outgoing transitions that remain identical to the original.
Another benefit of refining the invariant-based FSA in the described manner is identification of transitions that are legal according to the inferred invariants, but are not executed in the system executions. Hence, such behavior is a strong candidate for further testing and refinement of the object specification. For example, the refinement process identifies that the looping transition labeled makeEmpty on state $S_2$ is not traversed with the dynamic traces since it is a maybe transition. A detailed inspection of the implementation reveals this particular transition to be impossible and a result of overgeneralized program invariants.

4. DISCUSSION

In this paper, we presented an approach to improve the state-of-the-art in generating object-level behavioral models from dynamically observed executions. It does so by combining the information about likely data-value invariants and dynamic invocation sequences. The scalability of our approach can be sensitive to the complexity and number of objects, which can be a shortcoming compared to the lightweight kTail-based algorithms. However, the invariant-based FSA is an abstraction and we can consider invariants for only a subset of the object variables, trading scalability for precision. In addition, future generations of SMT solvers should be able to efficiently manage even more complex specifications. Another concern regarding our approach is sufficient coverage of interesting executions, but this is a general concern for dynamic behavior analysis. Finally, we believe that our approach can be straightforwardly applied to the context of static analysis enhanced with dynamic traces for improving static invariant detectors and contract specifications (a similar idea has already appeared in [16]).

Acknowledgments

This work was sponsored in part by the National Science Foundation, award numbers 0720612, 0820170, and CIF-288.

5. REFERENCES