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

Ivo Krka†, Yuriy 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-

ICSE '10, May 2-8 2010, Cape Town, South Africa

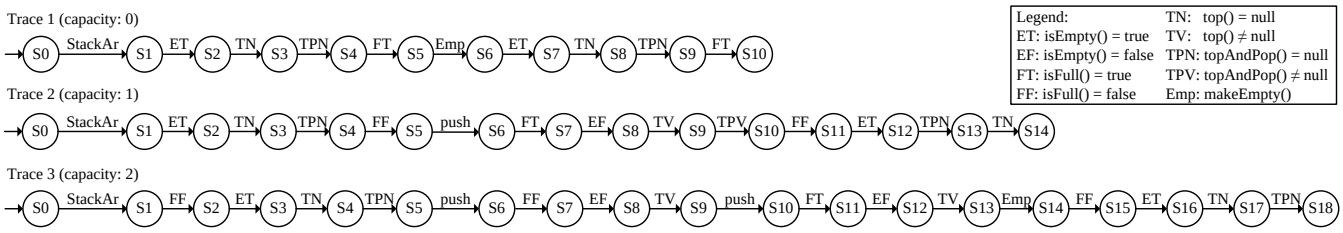

Figure 1: Three example StackAr method invocation traces.

ods: `void push(x)`, `Object top()`, `Object topAndPop()`, `void makeEmpty()`, `boolean isEmpty()`, and `boolean isFull()`. Internally, `StackAr` represents its stack as an array `Object[] theArray` and the top of the stack as `int topOfStack`. Whenever a `push` is performed on a full stack, an exception is thrown, while a `topAndPop()` on an empty stack returns `null`. A sample implementation of `StackAr` is available with the Daikon download.

The Daikon dynamic invariant detector [4] is a dynamic program analysis tool. Daikon observes data values of program executions and reports invariants that hold over all executions. Each invariant is a mathematical description of observed relationships among data values that the program computes. Together, invariants form an operational abstraction that contains preconditions, postconditions, and object-level properties. For example, Daikon may report that in observed executions, an object variable *capacity* was always greater than 0. The invariants are sound over the observed executions but are not guaranteed to be true in general.

We now illustrate Daikon's invariants with the `StackAr` example. After observing some executions, Daikon can report several object-level invariants: `theArray` is never `null`; the `topOfStack` index is never smaller than -1 and is always less than the length of `theArray`; the elements of `theArray` are null for indices larger than `topOfStack` and non-`null` otherwise. Daikon generates pre- and postconditions for `StackAr`'s methods in a similar manner.

Daikon produces operational specifications to facilitate understanding of internal class logic and postulate legal static "snapshots" at particular execution points. However, more interaction-oriented behavioral descriptions that describe the legal invocation sequences are useful for the development tasks discussed in Section 1. The most direct description of legal invocation sequences are the recorded invocation traces. On the downside, such traces are verbose, and fail to convey information that is sufficiently general to facilitate object behavior understanding. The *kTail* algorithm [1] generalizes the behavior from the observed sequences and has been recognized as a promising solution to behavioral model inference [11, 13]. Specifically, the *kTail* algorithm merges those states in the traces whose next k invocations are identical — i.e., identical k-*tails*. Selecting an appropriate k in the *kTail* method involves an innate tradeoff between precision (smaller k implies more spurious merges) and generalization (larger k implies fewer merges). Very similar to the *kTail* method is the *kHead* method that merges states based on k preceding invocations. In this paper, we focus on applying the *kTail* method; parallel results can be drawn for *kHead*.

To illustrate the *kTail* method, we will utilize three example `StackAr` invocation traces: creating and using a stack of capacity zero, one, and two (depicted in Figure 1). Each node represents a program state and each transition represents a method invocation, labeled with the method name and the return values. Let us now consider how *kTail* works on the given traces for $k = 3$. The algorithm correctly merges states S_1 and S_6 in Trace 1 because the three following invocations from each state are `isEmpty()=false`, `top()=null`, and `topAndPop()=null`. However, the algorithm also merges states S_1 in Trace 1 and S_{15} in Trace 3. This merge represents an incorrect generalization; it allows a non-zero-capacity stack to change capacity to zero after `isFull()` invocation returns `false`. The state-of-the-art *kTail*-based FSA generation techniques (e.g., [11, 13]) all suffer from similar overgeneralization flaws. In the following section, we demonstrate how our approach uses program invariants to avert such model overgeneralization.

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-

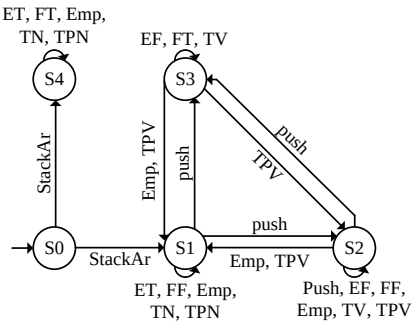

Figure 2: The invariant-based StackAr FSA.

thesizes behavioral models from pre- and postcondition specifications and execution scenarios. Daikon's invariants specify how each method's invocation affects the predicates, allowing us to generate all legal FSA transitions. A transition, labeled with a method invocation, exists between two states if that method's precondition is satisfied in the source state and the postcondition is satisfied in the destination state. The resulting object-level FSA contains a state for each legal object state and a transition for each legal method invocation within the object state. Figure 2 depicts the FSA for `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 $S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} \ldots \xrightarrow{I_{n-1}} S_n \xrightarrow{I_n} S_{n+1}$ simulates a trace $T = M_0 \rightarrow M_1 \rightarrow \ldots \rightarrow M_n$ if $I_i = M_i$ 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_1; 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_l 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_l 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_l and for all the outgoing transitions that remain identical to the original

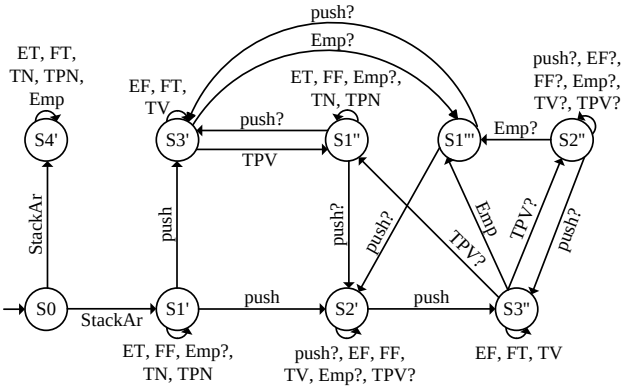

Figure 3: The StackAr FSA refined with invocation traces.

state's transitions. For example, the first `push` invocation in Trace 3 refines `StackAr` invariant-based FSA state S_2 into two new states, S_2' and S_2'', as shown in Figure 3. S_2' has incoming transitions on `push`, while S_2'' has incoming transitions on `topAndPop`. Note that maybe transitions in Figure 3 are marked with '?'. The beginning of Trace 3 is captured in the refined automaton with the following FSA sequence: $S_0 \overset{StackAr}{\rightarrow} S_1' \overset{isFull}{\rightarrow} S_1' \overset{isEmpty}{\rightarrow} \ldots$

The proposed refinement method may seem to potentially result in a large FSA due to the involved state splitting. However, the current version of our refinement algorithm [10] bounds the number of possible refinements by allowing a single state with incoming transitions labeled with I per predicate value combination. For larger objects, we may utilize a smaller number of state predicates to more efficiently obtain an invariant-based FSA; in such cases increasing this refinement bound could benefit the quality of the obtained automata. Additionally, we plan to assess the utility of statistics-based reasoning about state refinements (similar to [2, 16]). For example, trace statistics, annotated on FSA transitions, may uncover some invocation causalities that can be effected in additional state refinements.

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

[1] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, 21(6), 1972.

[2] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3), 1998.

[3] C. Csallner et al. DySy: Dynamic symbolic execution for invariant inference. In *Proc. of ICSE*, 2008.

[4] M. Ernst et al. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1), 2007.

[5] A. Fox. Addressing software dependability with statistical and machine learning techniques. In *Proc. of ICSE*, 2005.

[6] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of ICSE*, 2002.

[7] M. Harder et al. Improving test suites via operational abstraction. In *Proc. of ICSE*, 2003.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[9] J. Hatcliff and M. Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. In *Proc. of CONCUR*, 2001.

[10] I. Krka et al. Synthesizing component-level partial behavioral models from system specifications. In *Proc. of ESEC/FSE*, 2009.

[11] D. Lo et al. Automatic steering of behavioral model inference. In *Proc. of ESEC/FSE*, 2009.

[12] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proc. of WCRE*, 2006.

[13] D. Lorenzoli et al. Automatic generation of software behavioral models. In *Proc. of ICSE*, 2008.

[14] L. Mariani et al. Compatibility and regression testing of COTS-component-based software. In *Proc. of ICSE*, 2007.

[15] S. Reiss and M. Renieris. Encoding program executions. In *Proc. of ICSE*, 2001.

[16] S. Shoham et al. Static Specification Mining Using Automata-Based Abstractions. *IEEE TSE*, 34(5), 2008.

[17] H. van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, 2008.

[18] J. Yang et al. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. of ICSE*, 2006.

[19] Yices SMT Solver. `http://yices.csl.sri.com`, 2009.