Determining legal method call sequences in object interfaces

Samir V. Meghani Michael D. Ernst

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{smeghani,mernst}@lcs.mit.edu

Abstract

The permitted sequences of method calls in an object-oriented component interface summarize how to correctly use the component. Many components lack such documentation: even if the documentation specifies the behavior of each of the component's methods, it may not state the order in which the methods should be invoked. This paper presents a dynamic technique for automatically extracting the legal method call sequences in a component interface, expressed as a finite state machine. Compared to previous techniques, it increases accuracy and reduces dependence on the test suite. It also identifies certain programming errors.

1 Introduction

Software systems typically contain independently developed components. The system integrator needs a thorough understanding of the component interfaces. Particularly for components that implement or participate in protocols, or that maintain state, the order in which methods are invoked is crucial to the component's correct operation. Specification of methods in isolation may be inconvenient or insufficient: correctness of a method invocation often depends on the order in which the component's methods were invoked previously.

For example, consider a generic file server (Figure 1). The server first handles a USER message, which can be followed immediately by a PASSWD or another USER message if the first username was not accepted. If the password is rejected, only another PASSWD message may come next. Otherwise, the PASSWD message may be followed by a RETRIEVE, SEND, or QUIT message. An arbitrary sequence of RETRIEVE or SEND messages can occur. Once the QUIT message is received, no other methods may follow. Extracting such ordering information is the goal of this research.

This paper presents a dynamic technique for extracting

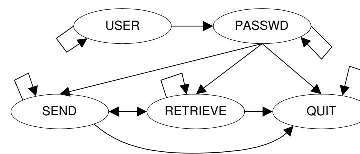

Figure 1. A generic file server. The FSM summarizes the order in which the methods should be called.

the legal method call sequences in an interface. Even when sequencing information is implicitly or explicitly present, automated tools can refine or double-check existing documentation. The technique uses dynamic invariant detection [ECGN01] to determine the likely pre/post-conditions of each of the component's methods, then uses logical comparison of pre/post-conditions to indicate which sequences of calls are likely to be legal. The technique outputs the legal call sequences in the form of a finite state machine (FSM), with states representing methods and transitions between states indicating what the next method call may be. Such an FSM is an approximation to a set of call sequences: transitions indicate that, under some circumstance, the method may be invoked safely, not that it can *always* be invoked safely.

The FSM indicating legal call sequences is useful for a number of purposes, including the following.

1. The FSM identifies the critical constraints on the order in which methods may be invoked. For example, as shown in Figure 2, a Java Thread's start() method may not be invoked twice in a row. Section 3.1 illustrates this point.
2. Missing transitions in the FSM can indicate holes in a test suite. Furthermore, the technique indicates the reason a transition is disallowed, in the form of incompatible post- and pre-conditions. Hence, the programmer is aided in determining the exact test case to add. Section 3.2 presents an example.
3. When used in combination with a call trace, our tech-

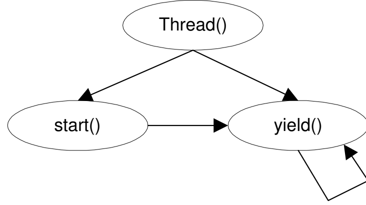

Figure 2. An FSM representation of the legal method call sequences in a Java Thread object. The states correspond to methods, and transitions indicate what the next method call may be. In a Thread, the programmer can call yield() immediately after the constructor or start(). Calling start() immediately after itself is illegal.

nique can help identify representation exposure problems. Section 3.3 contains examples.

This paper is organized as follows. Section 2 explains our technique for extracting the legal method call sequences, including an automatic technique for determining pre/post-conditions. Section 3 experimentally demonstrates the effectiveness of the technique. Section 4 compares our technique to previous work by Whaley et al. [WML02], which our research builds upon. Section 5 discusses additional related work, and section 6 concludes.

2 Technique

This section details our technique for extracting legal method call sequences. Section 2.1 presents the algorithm, which makes use of method pre/post-conditions. Section 2.2 describes an automatic technique for determining pre/post-conditions when they have not been supplied by the original component programmer. Section 2.3 discusses two potential shortcomings of the technique that may yield non-ideal FSMs.

2.1 Obtaining FSMs from pre/post-conditions

Method pre/post-conditions indicate much about legal method call sequences. If the post-conditions of method A are mutually exclusive with the pre-conditions of method B, then B may never be called immediately after method A (from the point of view of the client). Otherwise, there is some circumstance in which a call to method B may immediately follow a call to method A. We consider a method call sequence *legal* if the post-conditions of each method do not conflict with the pre-conditions of the immediately following method.

We represent the set of all legal call sequences as a finite state machine (FSM), with methods corresponding to states and transitions between states denoting the methods that may be called immediately after another method (Figure 2). A transition from method A to B indicates that under some circumstance, method B may follow method A, not that it must be allowed in all cases. For example, in Figure 3, the transition in StringTokenizer's FSM from next() to next() indicates that next() may be called immediately after next() under some circumstance. The transition is not always legal — it is only legal when there are actually more tokens available.

Our technique creates the FSM as follows:

1. Determine the likely pre/post-conditions of each method. Because of a lack of *a priori* specifications, our experiments use the Daikon invariant detector to identify these likely invariants, as discussed in Section 2.2. If available, we could use the pre/post-conditions provided by the programmer in specifications.

2. Compare the pre-conditions of each method to the post-conditions of every method (including itself). If they are not mutually exclusive, add the corresponding transition to the FSM.

The resulting FSM summarizes the legal method call sequences. Our prototype tool can present this FSM to the user as text or as a graphical representation.

We follow other researchers [CW98a, WML02] in summarizing call sequences by means of an *optimistic* FSM in which a transition between methods A and B indicates that in some circumstance, a call to method A may be immediately followed by a call to method B. Alternately, in a *pessimistic* FSM, the transition indicates that after any call to method A, a call to method B is permitted, and call sequence pairs represented by missing transitions may sometimes be possible. We could create a pessimistic FSM by changing the algorithm to add transitions only when one method's preconditions imply another's postconditions. We found such FSMs both smaller and (in our opinion) less useful, but they might be appropriate in some circumstances.

2.2 Determining pre/post-conditions

The technique of Section 2.1 uses method pre/post-conditions in order to determine which call sequences are illegal. Preferably, a programmer would supply exhaustive pre/post-conditions, and would ensure their correctness via theorem-proving or other verification techniques. (Even more ideally, the programmer would also document the legal method call sequences, removing the necessity to infer them. That would remove the primary need for our technique, though it could still be useful for refining or double-checking the documentation.) Unfortunately, such specifications are typically absent and, even when present, are rarely exhaustive. In particular, the experiments of Sections 3 and 4 used programs that lacked such specifications. Therefore, we automatically generated pre/post-conditions for use in the experiments.

Various techniques exist for determining method pre/post-conditions. Previous work [WML02] obtained these invariants from a sound and conservative, but weak, static analysis. We use dynamic invariant detection [ECGN01], which is much more precise and is accurate in practice [NE02] but is not guaranteed to be sound.

Dynamic invariant detection is a run-time technique for determining likely program properties; a set of likely program properties is also called an operational abstraction [HME03]. Our experiments use the Daikon implementation of invariant detection. Daikon operates on Java, C, and Perl programs, among others, and reports representation invariants and method pre/post-conditions; in this research, we applied it to Java programs and ignored the representation invariants.

Dynamic invariant detection monitors program execution (over whatever runs a user chooses, such as those from a test suite), observes the values the program computes, and generalizes from those values. In particular, at method entry and exit, the Daikon tool compares each variable in scope (including parameters, return values, class variables, and globals) to each other variable in scope. The generalization step uses an efficient generate-and-check algorithm that postulates many potential invariants and eliminates those that are ever falsified by observed values. It also uses static analysis, statistical analysis, and other techniques to enhance its output and to avoid reporting false positives [ECGN00].

Dynamic invariant detection is neither complete nor sound. It is incomplete because the grammar of properties that it checks is necessarily finite; that grammar was designed to be simple, broad, and generally useful, with uncomplicated invariants that can be applied in a number of situations. We did not modify the grammar for the experiments reported in this paper. However, users can assuage completeness problems by adding additional invariants to the Daikon invariant detector; doing so requires only writing a Java class that implements an interface containing four methods.

Dynamic invariant detection is unsound because the properties are likely, but not guaranteed, to hold in general. As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. If the observed executions do not fully characterize all possible execution environments, a candidate invariant that could have been falsified by an additional test case may be reported.

2.3 Accuracy of the technique

Given perfect pre/post-conditions, our technique creates an ideal FSM. (An FSM is a concise approximation of a set of call sequences, so some information may be lost, but the FSM would be as good as possible.) In the absence of programmer-supplied specifications, we use likely invariants produced by the Daikon tool, as noted in Section 2.2. Dynamic invariant detection is neither sound nor complete, and these inaccuracies have the potential to result in an FSM with missing or extraneous transitions. (Both problems also occur with non-perfect human-supplied specifications, and extraneous transitions can result from a conservative static analysis.)

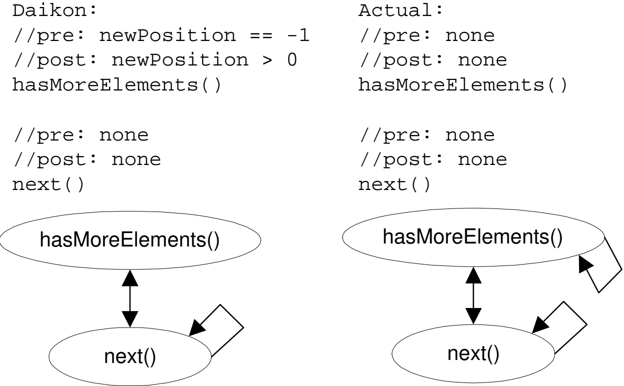

Figure 3. If the test suite is incomplete, our technique misses transitions. Suppose that while testing StringTokenizer, newPosition is always set to -1 when hasMoreElements() is invoked. Then invariant detection incorrectly reports a pre-condition of newPosition == -1 for hasMoreElements(), and our technique misses the transition from hasMoreElements() to hasMoreElements().

Incorrect invariants result in missing transitions. Dynamic analysis is inherently dependent on the quality of the test suite. With an incomplete test suite, the invariant detector may report over-specific pre/post-conditions — they held during the execution of the test suite, but will not necessarily hold for all executions. Figure 3 presents such an example. A component programmer comparing his or her mental model to the extracted model will notice the incompleteness, and improve the test suite accordingly (see Section 3.2). When the component programmer has not verified the extracted FSM, it still provides useful information to others, as illustrated in Section 4.1. (Missing transitions can indicate not only poor test suites, but also code errors; see Section 3.3.)

Missing invariants result in extraneous transitions. The Daikon invariant detector reports many commonly used types of invariants, but not all possible invariants; for example, it does not report the invariant `x is prime`. If such an invariant is missing from a pre- or post-condition, then pre/post-conditions that are actually mutually exclusive may not be identified as mutually exclusive, and our technique would infer incorrect transitions. This never occurred in our experiments (including ones not reported in this paper), but Figure 4 presents an example of how such a problem might occur.

```
Daikon:                   Actual:
//pre: none               //pre: none
//post: x == 0            //post: x == 0
PrimeStream()             PrimeStream()

//pre: none               //pre: x is prime
//post: none              //post: x is prime
next()                    next()
```

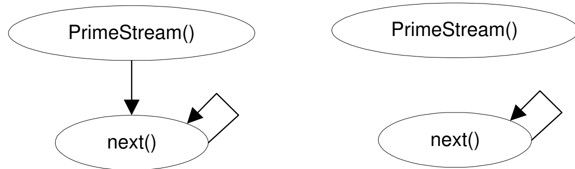

Figure 4. When invariant detection cannot detect all of the invariants present in the program, our technique may infer incorrect transitions. In this example, suppose the invariant detector cannot detect the `x is prime` invariant. Then our technique incorrectly infers a transition from PrimeStream() to next(), even though it is not possible in reality.

3 Experiments

This section presents experiments that illustrate the effectiveness of our technique: it extracts important constraints on call sequences, helps locate test suite deficiencies, and can aid in identifying representation exposure problems.

3.1 Identifying critical illegal call sequences

We extracted FSMs representing the legal method call sequences for several Java library classes. Figure 5 lists some components we tested our program on, a brief description of the component, the test suite used, and an English description of the critical constraints automatically identified as illegal. The programs shown include all those from Whaley et al [WML02] for which interesting results were presented. We extracted FSMs from a variety of other programs as well, but the results were of little value: the true FSMs permitted all or most transitions and so did not yield insight regarding component use.

3.2 Identifying test suite deficiencies

A programmer can use our technique to identify test suite deficiencies. To illustrate this point, consider the StringTokenizer class. We built our own test suite to extract an FSM. In building the test suite, we found that our technique provided useful information on how to improve the test suite.

We first built a black-box test suite of the component, which we believed to be of good quality. A programmer might make a similar test suite in practice. We then extracted an FSM for the StringTokenizer using this test suite, and compared the FSM to the actual, known FSM, which contains all 54 possible transitions. (The lack of interesting constraints is why StringTokenizer does not appear in Figure 5.) A programmer might make a similar comparison, between a mental model and the tool's output. Missing transitions in the extracted FSM indicate test suite deficiencies. The Daikon extracted FSM contained only 36 transitions, indicating significant deficiencies in the initial test suite. While a simple call trace would also indicate holes in the test suite, our technique provides useful information on how to improve the test suite beyond the sequences of method invocations which were not explicitly tested. Furthermore, the Whaley dynamic technique of Section 4.2 produced an FSM containing only 6 transitions from this test suite.

In order to determine which test cases to add, we examined the conflicting invariants that caused transitions to be eliminated using our technique. All of the missing transitions involved the invariant `newPosition == -1`. For example, the invariant detector reported a pre-condition of `newPosition == -1` and a post-condition of `newPosition > 0` for the hasMoreElements() method. In developing the test suite, we had assumed that hasMoreElements() did not change the state of the object, and therefore, the test suite did not need to make consecutive calls to it. However, hasMoreElements() does change the state of the object, as it updates the `newPosition` field which indicates where in the String the next token begins. The technique gave us an indication of the state to put the object into before making the corresponding method invocations, to ensure the extracted FSM contains the desired transition. In essence, it pinpointed the exact test case to add.

Alternately, for transitions that are properly missing, the pre/post-conditions indicate exactly why the invariant is excluded from the extracted FSM, aiding in understanding of the impossibility of a call sequence. This abstraction of behavior is more useful to programmers than simply the information that a particular call sequence has not yet been covered by a test suite.

3.3 Locating representation exposure problems

The data abstraction paradigm requires programmers to hide the implementation details of a software component from the outside world. Components that adhere to this principle are easier to maintain and reuse [LG86]. *Representation exposure* occurs when an implementation fails to fully encapsulate all implementation details. For instance, clients might be able to observe or, more seriously, to modify, concrete data structures held in variables or fields. Representation exposure of mutable data structures can allow client code to push a component into an illegal state. Combining our technique with a call trace (an ordered list of observed calls from some program execution) can sometimes locate such problems. One attractive property of our technique is that it identifies representation exposures before they cause faults, by noticing inconsistencies between the inferred FSMs and the actual call traces. Thus, if a client modifies an implementation's data structures directly, but in

Component	Description	Test Suite Executed	Key Constraints Identified
Vector, LinkedList, ArrayList	Data structures in java.util	Nimmer [NE02]	Data must be added to a data structure before it can be accessed or removed.
FileInputStream, FileOutputStream, FileReader, FileWriter	Streams in java.io	Copy program from Java Tutorial	A stream cannot be read from after it is closed.
PlainSocketImpl, ServerSocket, DatagramSocket, MulticastSocket	Networking components in java.net	HttpTest program from JCSL	A connection cannot be read/written after it is closed.
ThreadGroup, Thread	Thread components in java.util	HttpTest program from JCSL (uses multi-threaded server)	A thread cannot be started twice.
Signature	Security component in java.security	SimpleSignatureTest from JCSL	Cannot `verify` an object that is in `sign` state, and vice versa.

Figure 5. Some components we extracted FSMs for. The table contains the name of the component, a general description of its function, the test suite we used to extract the FSM, and the method call sequences that our technique identified as illegal.

a safe way that permits the program to continue running, our technique can identify the representation exposure nonetheless.

Suppose a client makes two method calls on an object, with no other calls in between. However, in between the two calls, the client somehow changes the state of the object without calling a method, i.e., the component programmer exposed the representation. The call trace shows that the two methods were called consecutively. However, the FSM extracted using our technique will not necessarily show this transition, if the state of the object is changed between the two method invocations in a manner such that the pre/post-conditions detected become mutually exclusive. Therefore, if the call trace contains consecutive method invocations that do not appear in the FSM extracted using our technique, there is probably a representation exposure problem, because invariant detection cannot possibly report mutually exclusive pre/post-conditions between consecutive method calls unless the state is changed between the two calls, without calling a method. We present two examples where this technique successfully identifies representation exposure.

We first consider the PlainSocketImpl class, the default implementation of a network socket, in the java.net package. We used as a test suite the HttpTest program, part of a larger test suite for many of the standard Java networking components. The call trace contained PlainSocketImpl()–getOutputStream() pairs (that is, sometimes getOutputStream() was called immediately after the constructor), but such sequences were not permitted by the inferred FSM. The inferred FSM forbade such sequences because getOutputStream() requires that the `address` field not be null, but the constructor does not initialize the `address` field. When we read the code, we found that the ServerSocket class directly sets the `address` field. External code can modify the state of the PlainSocketImpl component without calling a method because the `address` field has protected access. This makes for a fragile system — if the programmer of PlainSocketImpl renames the `address` field, components that depend on PlainSocketImpl will break. Also, external code could assign the `address` field to null, causing the PlainSocketImpl to malfunction.

Another variety of representation exposure problems results from returning an internal data structure, such as an array, instead of a copy of the array. If the exposed data structure is mutable, external code can modify it, and thereby push the component into an unexpected state. Figure 6 shows a simplified example of such an error. The getElements() method returns the `elements` Vector, instead of a copy of it. If client code uses the class as indicated in the main method, the program continues without error; however, the CharSet contains String objects instead of Character objects, which is not the intended behavior. We can identify such a problem by extracting the FSM, and comparing the results to a call trace as described above. The extracted FSM does not allow calling remove(int) after getElements(), even though it occurs in the call trace. This indicates that the object has changed between the calls to getElements() and remove(int), leading a programmer to the representation exposure in getElements().

4 Comparison to other techniques

This research represents an improvement on Whaley et al.'s "Automatic extraction of object-oriented component interfaces" [WML02]. Whaley presents a technique for extracting the legal method call sequences in a component interface, and presents them as a FSM. (Whaley refers to this FSM as the interface, but we call it the legal call sequences

```
public class CharSet {
  private Vector elements = new Vector();

  public void add(char c) {
    Character ch = new Character(c);
    if (!elements.contains(ch)) {
      elements.add(ch);
    }
  }
  // Representation exposure: should return a copy
  public Vector getElements() {
    return elements;
  }
  public void remove(int i) {
     elements.remove(i);
  }
  ...
}

public class CharSetClient {
  public static void main(String[] args) {
    CharSet s = new CharSet();
    s.add('a');
    // Directly modify the elements vector.
    s.getElements().set(0, "string");
    // No run-time error
    s.remove(0);
  }
}
```

Figure 6. A simple example of representation exposure that causes no run-time error (for this client), but is detected by our technique. The class incorrectly returns the elements Vector, instead of a copy. The CharSetClient class illustrates how to modify the state of the CharSet without calling its methods.

to avoid confusion over better-known meanings for "interface".) Whaley extracts two such FSMs, using two distinct strategies. The first strategy conservatively identifies illegal transitions using a static analysis of the program text. The second approach, a dynamic analysis, conservatively identifies legal transitions. These two FSMs bound the actual, unknown FSM in number of transitions — the static FSM is an upper bound while the dynamic FSM is a lower bound.

4.1 Whaley static analysis

Whaley's static analysis operates exactly like our dynamic analysis presented in Section 2.1, except that the pre/post-conditions are obtained via a conservative static analysis rather than via a dynamic analysis.

Preconditions are the conjunctions of the negations of the conditions under which a method is guaranteed to throw an error. Whaley's static analysis assumes that programmers check method pre-conditions and explicitly throw exceptions to indicate violations.

More specifically, the static analyzer creates a control flow graph for each method, finds the statements that throw exceptions, and identifies the path from the start node in the graph to these statements. The logical `nand` of all the conditional expressions on the path correspond to one pre-condition of the method that is statically checked. Only condition of the form `variable == constant` or `variable != constant` are considered; all other conditions are ignored. For example, suppose the conditional expressions on two separate paths are [`state == 0` and `var == 2`] and [`state == 2` and `var == state`]. Whaley's analyzer classifies `not(state == 0 && var == 2)` as a pre-condition.

Each pre-condition found by the static analyzer is guaranteed to be correct, but the list of pre-conditions is not necessarily complete. In some cases, programmers practicing defensive programming may check pre-conditions; but by definition, a precondition is something that a method is permitted to assume without checking, and programmers may omit checks for performance reasons. Furthermore, the method may throw a runtime exception implicitly that does not have to be thrown explicitly [Blo01], or the pre-condition check may be of another syntactic form, causing the analysis to miss the pre-condition.

Postconditions are properties of the form `variable == constant` that are guaranteed to hold at method exit. The analyzer identifies assignments of variables to constants that are made on every possible path through the method, and are not later changed on the path.

The FSM remaining after eliminating conflicting pre/post-conditions (Section 2.1) is an upper bound on the actual FSM. The pre/post-conditions identified are definitely accurate, so the transitions removed from the FSM are definitely not allowed.

4.2 Whaley dynamic analysis

Whaley's dynamic analysis runs a program and creates an FSM from the observed call sequences, adding a transition for each successive pair of observed method calls. Whaley notes two shortcomings of this technique. First, as with any dynamic analysis, the result is dependent on the quality of the test suite. Our technique is less prone than Whaley's to poor test suites (see Section 4.3.2).

Second, accessor methods can cause the technique to yield an incorrect model. For example, in a data structure, size() and elementAt(int) operations do not change the state of the object. However, if the test suite always makes calls in the sequence add–size–elementAt, no transition from size() to add() is recorded. To avoid this problem, Whaley statically identifies the state-preserving methods by searching the method for assignments to class or instance variables. If there are no such assignments, the method is considered state-preserving. For each state preserving method invocation, Whaley adds a transition from the last state-modifying method invoked. Additionally, transitions are added so that all of the state-preserving methods invoked consecutively form cliques in the FSM.

By contrast, our technique does not need such an analysis, which is prone to conservatism in identifying state-preserving methods. If a method is state-preserving, then the invariant detector will report identical preconditions and postconditions for it. Furthermore, the invariant detector will report preconditions for every other method that are independent of the methods invoked previously. Thus, our technique permits method call sequences that were never taken to be identified as legal, based on method pre/post-conditions inferred from observed executions.

Section 3.3 noted a third problem with Whaley's dynamic technique: representation exposure errors can lead to incorrect FSMs. As noted in Section 3.3, our technique permits detection of such errors.

4.2.1 Comparison with our technique

Our technique extracts FSMs that are at least as complete as those of the Whaley dynamic technique. Except in cases where the representation is exposed, the pre/post-conditions detected for methods that were invoked consecutively cannot conflict. This property ensures that the extracted FSM contains as many transitions as a simple call trace analysis.

The Whaley dynamic technique adds the additional insight of separating state-preserving methods. Our technique inherently separates state-preserving methods — the invariants detected at method entry and exit are identical for these methods. Hence, the extracted FSM is guaranteed to contain all of the transitions that are in the Whaley dynamically extracted FSM.

Our technique achieves greater completeness by using invariants to infer transitions that were not seen at runtime. The technique can infer these transitions because the dynamically detected invariants generalize to future runs. If the pre/post-conditions of two methods do not conflict, then it is likely that consecutive invocation of the two methods is legal. Even if such consecutive invocation did not occur in the specific test suite, our technique infers it from the conditions that hold when each of the methods is invoked in other circumstances.

4.3 Experimental comparison

The Whaley techniques and our technique have the same goal: producing an FSM that approximates legal call sequences. Our experiments suggest that our technique has several advantages over Whaley's techniques. (We expect that combining the techniques, rather than using either in isolation, will prove most advantageous in the long run.) First, our technique produces an estimate that is closer to the correct FSM than either of Whaley's techniques (Section 4.3.1). Second, our technique is less sensitive to test suite than Whaley's dynamic technique: it produces a good estimate of the true FSM even from a poor test suite (Section 4.3.2). Third, our technique can identify representation errors (Section 3.3). Fourth, our technique can indicate how to improve test suites in terms of data values as well as call sequences (Section 3.2).

The major disadvantage of using our technique is that the extracted FSM may contain incorrect transitions, as suggested in Section 2.3. Given a test suite that executes every feasible pair of method sequences, Whaley's dynamic technique would produce the true FSM; our technique would have no missing transitions but might suffer from extra transitions. This problem never occurred in our experiments over large portions of the Java standard libraries, but makes a case for using the FSMs extracted using each technique in combination.

4.3.1 Comparison of extracted FSMs

We implemented the three techniques and evaluated them on a collection of programs. The results of several of these experiments are shown in Figure 7. For the dynamic analyses, we started with test suites in the Java source package available under the Java Community Source License [Sun]. When none were available or they were unrealistically small, we developed our own test suites by hand, an easy task taking only a few minutes per program.

The test subjects of Figure 7 are StackAr, QueueAr, Vector, StringTokenizer, PlainSocketImpl, and Signature. StackAr and QueueAr are array-based implementations of simple data structures, found in a Java data structures book [Wei99]. Vector is the implementation of a vector data structure from the java.util package in the Java Standard Development Kit. The StringTokenizer class, a utility class for parsing a string, also originates from this source. The PlainSocketImpl class is the default implementation of network sockets in Java. Signature, from the java.security package, provides authentication.

Because extracting call sequence FSMs is an information retrieval task, we report our results in terms of the standard precision and recall measures [Sal68, vR79]. Precision, a measure of correctness, is the fraction of reported transitions that appear in the goal: $\frac{\text{correct}}{\text{reported}}$. Recall, a measure of completeness, is the fraction of goal transitions that are reported: $\frac{\text{correct}}{\text{goal}}$. Both measures are always between 0 and 1.

As expected, the Whaley static FSM was an upper bound on the actual FSM, and the Whaley dynamic FSM was a lower bound. Our technique induced the actual FSM in every case.

We briefly give an example of the differences. In the case of Vector, the static analysis identified that calling firstElement() or lastElement() immediately after removeAllElements() or the constructor is illegal. The Whaley dynamic analysis also classified such transitions as illegal, but concluded that the only method that could be called immediately after two of the constructors is addElement(Object). Our technique extracted the correct model.

Class	Total possible	Actual	Whaley static			Daikon dynamic			Whaley dynamic			Suite size
			trans.	rec.	prec.	trans.	rec.	prec.	trans.	rec.	prec.	
Vector	648	628	640	1.00	.981	628	1.00	1.00	112	.178	1.00	76
StackAr	63	63	63	1.00	1.00	63	1.00	1.00	42	.666	1.00	18
QueueAr	48	45	48	1.00	1.00	45	1.00	1.00	32	.711	1.00	12
Signature	289	275	276	1.00	.938	275	1.00	1.00	177	.644	1.00	177
StringTokenizer	54	54	54	1.00	1.00	54	1.00	1.00	24	.444	1.00	21
PlainSocketImpl	324	316	324	1.00	.975	316	1.00	1.00	215	.664	1.00	215
Average	238	230	234	1.00	.982	230	1.00	1.00	117	.551	1.00	87

Figure 7. For each class listed in the first column, we compared the FSMs extracted using each technique. The chart shows the number of legal transitions, precision, and recall of each technique compared to the actual FSM. Test suite size is number of consecutive invocations (number of calls in the call trace); we made no effort to reduce the size of the test suites.

4.3.2 Dependence on test suite

A test suite that tests every possible sequence of method invocations in every possible state of the component yields identical models in our experiments using the Daikon based technique or the Whaley dynamic technique; however, such test suites are burdensome to write. The Whaley dynamic technique relies more heavily on the quality of the test suite — poor test suites yield poor results. By contrast, our technique both gives better results and improves its results more rapidly as the test suite improves.

For example, we ran both dynamic techniques using the (bare-bones) Sun-provided test suites for the PlainSocketImpl, Signature, and StringTokenizer classes. Both dynamic approaches yielded 100% precision. The Whaley dynamic approach gave average recall of 11%, and the Daikon dynamic approach had average recall of 47%. This differs from the numbers show in Figure 7, which benefited from a modest amount of work (far less than one hour per class) to augment poor test suites.

To further quantify the effect of test suite quality, we performed an experiment using the Vector and Signature classes in the java.security package. We chose Vector because it is fairly easy to write a test suite for. We selected Signature because it has a model more complex than the simple put/get relations of Vector, and the documentation explicitly states the legal call sequences. For each class, we built a thorough test suite that would yield the correct results using the Whaley dynamic analyzer. However, we limited the methods we tested for each class, because we cared only about the methods that have interesting constraints on the call sequences. For Vector, we tested the constructor and the elementAt, addElement, removeElement, and size methods. In the Signature test suite, we called only the nine methods that are important in the model: initVerify, initSign_1, initSign_2, sign_1, sign_2, sign_3, update_1, update_2, and verify, where subscripts indicate overloaded method names.

We randomly selected test cases from these suites, and extracted FSMs using both dynamic analyses. Some of the test cases exercised more than just two methods, because testing a pair of consecutive methods requires some setup. For example, in a Vector, to test the transitions from add(Object) to remove(int), the Vector() to add(Object) transition must also be tested. For uniformity, we always ran a set of simple test that have the same effect as this setup code. The simple test for Signature is the one described in Section 3, which accounts for six transitions in the Whaley dynamic FSM and 38 in the Daikon FSM. For the Vector test, we simply created a new Vector, added an element, checked the size, and removed the element. This accounts for three transitions in the Whaley dynamic FSM and eight in the Daikon FSM.

Figure 8 gives the results of these experiments. Our technique's FSM approaches a final value far more quickly than the Whaley dynamic FSM. The Daikon curve quickly approaches its final value, while the Whaley dynamic technique takes significantly longer to stabilize. This is further indication that our technique is more tolerant of poor test suites than the Whaley dynamic technique.

5 Related work

Automatically extracting legal method call sequences is far from a new concept. Parnas [BP78, PW89] first suggested using legal method call sequences to specify interfaces. He introduced a language for specifying the legal sequences. However, the constraints become difficult to express on large components because the number of possible legal sequences becomes too large. Classifying each possible sequence as legal or illegal manually is impractical, and places too large a burden on the programmer.

Koskimies [KMST96] proposed using dynamic program traces to extract state charts from legacy code, but the tools were not automated. The extracted information is equivalent to a call trace of the program. Whaley's dynamic analyzer is an automation of this technique, with the additional insight on state-preserving transitions.

Cook and Wolf [CW98a] generalize from event traces to finite state machines, in the domain of software change processes. Their FSMs capture sequence, selection, and iteration, and they evaluate three different techniques: one

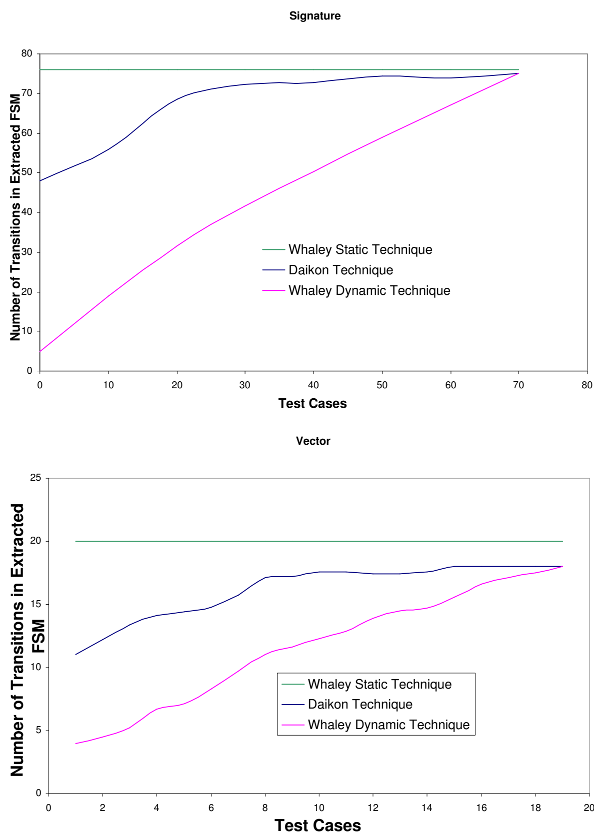

Figure 8. For the Signature and Vector classes, we compared three FSM extraction techniques over test suites of varying size. This figure graphs the number of transitions versus the number of test cases executed, averaged over ten runs at each test suite size. Our technique extracts an FSM that converges more quickly to the true FSM than the Whaley dynamically extracted FSM; the Whaley statically extracted FSM never converges to the true FSM, but does provide an upper bound.

statistical, one algorithmic, and one mixed. They also discuss how to detect patterns of concurrent behavior from event traces, inferring system behavior from statistics over number, frequency and regularity of event occurrences [CW98b].

Much work has been devoted to extracting and verifying temporal properties of components. Ammons et al [ABL02] used machine learning to extract temporal constraints on interfaces. Like our technique, traces of program execution are recorded. The Ammons technique inputs these traces to machine learning software, which attempts to identify the important temporal specifications. One difference is that Ammons's technique uses a machine learning technique that handles noise; it can infer sequences that do not perfectly match the observed executions. By contrast, our technique is not resilient to noise: its output is a faithful abstraction of its input (up to the limits of its accuracy). The two approaches are complementary, and each may be more appropriate for certain applications.

Recently, the Daikon system has been augmented to detect temporal invariants in addition to pre/post-conditions. The system analyzes program traces to identify ordering constraints on various types of inputs, such as the assignment of a variable or the invocation of a method. The system can identify put/get and open/close constraints such as the ones we have extracted, but will most likely prove to be more dependent on test suite quality, because it does not make use of pre/post-conditions.

6 Conclusion

This paper presents a novel method for extracting legal method call sequences from software components. As others have done, we extract a finite state machine representing the sequences, with methods as states and transitions between states indicating what the next method called may be.

Our technique uses dynamically detected likely method pre/post-conditions, to identify illegal transitions. The technique can thus use execution traces to infer the legality of transitions that were never taken during the observed executions. The technique may theoretically result in missing or extraneous transitions, but we found the former to be both relatively infrequent and easy to fix (and the technique aids in such corrections), and the latter never occurred in practice.

Our technique improves over previous techniques in several respects: the extracted FSM is more accurate (is closer to the true FSM), is less dependent on the quality of the test suite for the component, is useful in identifying test suite deficiencies, and can identify representation errors.

Acknowledgments

We are indebted to Jeremy Nimmer for recognizing that dynamic invariant detection could be used to extract legal call sequences in component interfaces.

References

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison Wesley, 2001.

[BP78] Wolfram Bartussek and David L. Parnas. Using traces to write abstract specifications for software modules. In *Proc. 2nd Conference of European Cooperation in Informatics*, pages 211–236, October 10–12, 1978.

[CW98a] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[CW98b] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *FSE '98, Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45, Lake Buena Vista, FL, USA, November 1998.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[KMST96] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, University of Tampere, Dept. of Computer Science, Finland, July 1996.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[PW89] David Lorge Parnas and Yabo Wang. The trace assertion method of module interface specification. Technical Report 89-261, Department of Computing and Information Science, Queen's University at Kingston, Kingston, Ontario, Canada, October 1989.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[Sun] Sun Java community source license website. `http://www.sun.com/software/java2/`.

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.

[WML02] John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, Rome, Italy, July 22–24, 2002.