Appears in *Proceedings of the 3rd Workshop on Specification and Verification of Component-Based Systems at FSE-12 (SAVCBS 04)*

Automatic Extraction of Sliced Object State Machines for Component Interfaces

Tao Xie David Notkin

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195, USA
{taoxie,notkin}@cs.washington.edu

ABSTRACT

Component-based software development has increasingly gained popularity in industry. Although correct component-interface usage is critical for successful understanding, testing, and reuse of components, interface usage is rarely specified formally in practice. To tackle this problem, we automatically extract sliced object state machines (OSM) for component interfaces from the execution of generated tests. Given a component such as a Java class, we generate a set of tests to exercise the component and collect the concrete object states exercised by the tests. Because the number of exercised concrete object states and transitions among these states could be too large to be useful for inspection, we slice concrete object states by each member field of the component and use sliced states to construct a set of sliced OSM's. These sliced OSM's provide useful state-transition information for helping understand behavior of component interfaces and also have potential for being used in component verification and testing.

1. INTRODUCTION

Component-based software development has become an emerging discipline that manages the growing complexity of software systems [18]. In component-based software development, software components are the building blocks of a software system. When component users try to reuse an existing component in their applications, they need to understand behavior of the component's interface, such as usage rules that they are required to obey or expected results of some component usage scenarios. When component developers or users test their components before being released or reused, they need to know whether their components behave correctly against some usage rules or expectations. However, in practice, component-interface-usage rules or behavioral specifications are usually not equipped for many components. Even if usage rules or behavioral specifications are provided, they are often informally written in interface documentation such as Java API documentation [17], being prone to errors or difficult to be understood.

In this work, among a variety of specifications, we propose to use the form of *object state machines* (OSM) to characterize behavior of component interfaces and dynamically extract OSMs from automatically generated tests for component interfaces. We have proposed OSM in our previous work [27]. A state in an OSM represents the state that a component object is in at runtime. A transition in an OSM represents method calls invoked through the component interface transiting the component object from one state to another. States in an OSM can be concrete or abstract. A concrete state of a component object is characterized by the values of all transitively reachable fields of the component object. A concrete OSM is an OSM with concrete states. Given a component, we generate a set of tests for the component and then collect all exercised concrete states of component objects and transitions (method calls through component interfaces) among states. These collected states and transitions are used to construct a concrete OSM; however, the concrete OSM is often too complicated to be useful for understanding. To address this problem, our previous work has proposed the *observer abstraction* approach [27]; the approach uses the return values of observers (interface methods with non-void returns) invoked on a component object as an abstract state in an OSM. This paper proposes a new supplementary approach of slicing a concrete state by each member field of the component[1]. Different from our previous observer abstraction approach [27], our new approach is not affected by the availability or complexity of observers in component interfaces. Our state slicing technique is inspired by Whaley et al's model slicing by member fields in dynamically extracting component interfaces [22]; however, our new approach is more accurate in characterizing component behavior and does not require a good set of existing system tests for exercising component interfaces. In this work, we focus on components in the form of Java classes and component interfaces in the form of public methods in classes; however, we expect the approach could be easily extended to components in other forms.

The rest of this paper is organized as follows. Section 2 describes a nontrivial illustrative example. Section 3 introduces the formal definition of an OSM. Section 4 illustrates the automatic approach of extracting sliced OSM's. Section 5 discusses main issues of the approach and proposes future work. Section 6 presents related work and Section 7 concludes.

[1]We define state slicing or OSM slicing following the definition of model slicing by Whaley et al. [22]. The use of *slicing* in these definitions differs from the one in a more common definition: program slicing [21], which is closely related to some notion of dependence.

```
public class LinkedList extends AbstractSequentialList
     implements List, Cloneable, java.io.Serializable {
  private transient Entry header
                        = new Entry(null, null, null);
  private transient int size = 0;
  private static final long serialVersionUID
                   = 876323262645176354L;

  public LinkedList() {...}
  public void add(int index, MyInput element) {...}
  public boolean add(MyInput o) {...}
  public boolean addAll(int index, Collection c) {...}
  public void addFirst(MyInput o) {...}
  public void addLast(MyInput o) {...}
  public void clear() {...}
  public Object remove(int index) {...}
  public boolean remove(MyInput o) {...}
  public Object removeFirst() {...}
  public Object removeLast() {...}
  public Object set(int index, MyInput element) {...}
  public Object get(int index) {...}
  public ListIterator listIterator(intindex) {...}
  public Object getFirst() {...}
   ...
}
```

Figure 1: A LinkedList implementation

2. ILLUSTRATIVE EXAMPLE

As an illustrative example, we use a nontrivial data structure: a LinkedList class, which is the implementation of linked lists in the Java Collections Framework, being a part of the standard Java libraries [17]. Figure 1 shows declarations of LinkedList's fields and some public methods that we shall refer to in the rest of this paper (these public methods either modify object states or throw uncaught exceptions).[2] This implementation uses doubly-linked, circular lists that have a `size` field and a `header` field, which acts as a sentinel node. In addition, it also has a static `serialVersionUID` field, which is used during serialization. It inherits a `modCount` field from a super class `AbstractList`; this field records the number of times the list has been structurally modified. LinkedList has 25 public methods, 321 noncomment, non-blank lines of code, and 708 lines of code including comments and blank lines.

3. OBJECT STATE MACHINE

We have defined an object state machine for a component in our previous work [27]:

DEFINITION 1. *An* object state machine *(OSM) M of a component c is a sextuple M = (I, O, S, δ, λ, $INIT$) where I, O, and S are nonempty sets of method calls in c's interface, returns of these method calls, and states of c's objects, respectively. $INIT \in S$ is the initial state that the machine is in before calling any constructor method of c. $\delta : S \times I \to P(S)$ is the state transition function and $\lambda : S \times I \to P(O)$ is the output function where $P(S)$ and $P(O)$ are the power sets of S and O, respectively. When the machine is in a current state s and receives a method call i from I, it moves to one of the next states specified by $\delta(s, i)$ and produces one of the method returns given by $\lambda(s, i)$.*

When a method call in a component interface is executed, an uncaught exception might be thrown. To represent the state where an object is in after an exception-throwing method call, we introduce a special type of states in an OSM: *exception states*. After a method call on an object throws an uncaught exception, the object is in an exception state represented by the type name of the exception. The exception-throwing method call transits the object from the object state before the method call to the exception state.

[2]We change those `Object` argument types to `MyInput` so that we can guide ParaSoft Jtest 5.1 [15] (being used in our test generation described in Section 4.1) to generate better arguments; `MyInput` is a class that contains an integer field `v`.

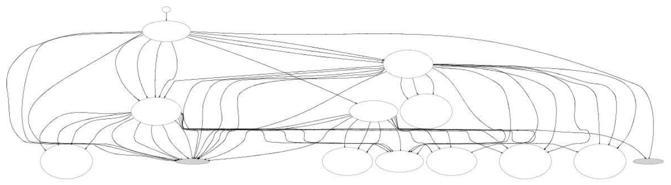

Figure 2: An overview of LinkedList concrete OSM (containing only state-modifying transitions) exercised by generated tests

An OSM can be deterministic or indeterministic. To help characterize indeterministic transitions, we have defined two statistics in a dynamically extracted OSM: transition counts and emission counts [27]. Assume a transition t transits state s to s', the *transition count* associated with t is the number of concrete states enclosed in s that are transited to s' by t. Assume m is the method call associated with t, the *emission count* associated with s and m is the number of concrete states enclosed in s and being at entries of m (but not necessarily being transited to s'). If the transition count of a transition is equal to the associated emission count, the transition is deterministic and indeterministic otherwise.

The object states in an OSM can be concrete or abstract. A concrete OSM is an OSM where all states are concrete object states. We have proposed several techniques to represent object states in our previous work [24]; we use the WholeState technique to represent concrete object states in this work. Given an object, the WholeState technique collects the values of all fields reachable from the object and uses these field values to represent the concrete state of the object. When we encounter a reference-type field with a non-null value during field-value collection, we use a linearization algorithm [24] to collect the field value as the field name of the earliest collected aliased field; if we cannot find any earlier collected aliased field for the field, we collect its value as "not_null". Two concrete object states are nonequivalent if their representations are different. A set of nonequivalent concrete object states contain concrete object states any two of which are nonequivalent.

For example, there are 11 nonequivalent concrete object states of LinkedList exercised by tests generated in our test generation step (Section 4.1). There are 161 transitions among these states (including both state-modifying and state-preserving transitions). There are two exception states: `IndexOutOfBoundsException` and `NoSuchElementException`. Figure 2 shows a concrete OSM (containing only state-modifying transitions) exercised by generated tests. [3] We have observed that the concrete OSM is too complex to be useful for inspection.

To reduce the complexity of an OSM, we shall extract an abstract OSM containing abstract states instead of concrete states. An *abstract state* of an object is defined by an *abstraction function* [14]; the abstraction function maps each concrete state to an abstract state. In this work, for each member field of a component, we define an abstraction function that maps each concrete state to an abstract state characterized by the values of those fields reachable from the member field. The next section describes the details of the state slicing approach.

[3]We display OSM's by using the Grappa package, which is part of graphviz [9].

4. SLICED-OSM EXTRACTION

Given a Java class, we automatically generate a set of tests for extensively exercising object states within a (small) scope (Section 4.1). During the execution of the generated tests, we slice each exercised concrete object state by member fields and construct abstract OSM's (Section 4.2). For a member field with a reference type, we additionally conduct structural abstraction on the sliced state to further abstract primitive field values reachable from the member field (Section 4.3).

4.1 Test Generation

Given a Java class, we first use Parasoft Jtest 5.1 [15] (a commercial Java testing tool) to generate method arguments for each public method of the class. Jtest generates a small set of method arguments and invoke public methods with these arguments after invoking class constructors. For example, Jtest 5.1 generates two tests for exercising `add(MyInput element)`:

```
Test 1:
        MyInput t0 = new MyInput(0);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
Test 2:
        MyInput t0 = new MyInput(7);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
```

Jtest also allows the user to configure whether to generate null values as method arguments. For the sake of simplicity in illustrative results, we configure Jtest 5.1 not to generate null argument values for LinkedList.

A list of arguments for a method consists of all arguments required for invoking the method. Two lists of arguments for a method are equivalent if the concrete state of each argument in the first list is equivalent to the concrete state of the corresponding argument in the second list. If an argument is of a primitive type, its concrete state is represented by its primitive values. If an argument is of Java built-in `String`, `Integer`, or another primitive-type wrapper, the concrete state of the argument is represented by its character strings or corresponding primitive value. If arguments are of other reference types, we use the WholeState technique (described in Section 3) for comparing their state equivalence.

We use the Rostra tool (developed in our previous work [23, 24]) to monitor the execution of the test class generated by Jtest and generate new tests based on collected method arguments. The pseudo-code of our test-generation algorithm is presented in Figure 3 (adapted from our previous work [23]). The test generation algorithm receives a set of third-party generated tests (e.g. Jtest-generated tests) and a maximum iteration number that specifies how many iterations we shall use to grow concrete object states. We first run these third-party generated tests and collect run time information from their execution; the collected runtime information includes the set of all nonequivalent non-constructor-method argument lists and nonequivalent object states exercised during the execution.

Then in the first iteration, the frontier set (containing the object states to be fully exercised) includes those nonequivalent states at exits of constructors exercised by the third-party tests. We iterate each object state in the frontier set and each argument list in the set of nonequivalent non-constructor-method argument lists exercised by the third-party tests. For each combination of an object state and an argument list, we construct a test by invoking the corresponding method with the argument list on the object state. We execute all constructed tests and collect runtime information. In the subsequent iteration, the frontier set includes those nonequivalent states exercised by the new tests but not exercised by any test in previous iterations. We continue the iterations until we have reached the maximum iteration number or the frontier set contains no object states.

```
Set testgen(Set thirdPartyTests, int maxIterNum) {
  Set newTests = new Set();
  RuntimeInfo runtimeInfo = runAndCollect(thirdPartyTests);
  Set nonEqArgLists = runtimeInfo.getNonEqArgsLists();
  Set frontiers = runtimeInfo.getAfterInitNonEqObjStates();
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set newTestsForCurIter = new Set();
      foreach (objState in frontiers) {
        foreach (args in nonEqArgLists) {
          Test newTest = makeTest(objState, args);
          newTestsForCurIter.add(newTest);
          newTests.add(newTest);
        }
      }
      runtimeInfo = runAndCollect(newTestsForCurIter);
      frontiers = runtimeInfo.getNewNonEqObjStates().
  }
  return newTests;
}
```

Figure 3: Pseudo-code of the test-generation algorithm.

For the LinkedList example, we configure the maximum iteration number as two. For illustration purpose, let us assume here that third-party tests contain only two tests (Tests 1 and 2) that we have shown in the beginning of this section. Then in the first iteration, we generate Tests 1 and 2; in the second iteration, we generate Tests 3 and 4 shown as below:

```
Test 3:
        MyInput t0 = new MyInput(0);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
        MyInput t1 = new MyInput(7);
        boolean RETVAL1 = THIS.add(t1);
Test 4:
        MyInput t0 = new MyInput(7);
        LinkedList THIS = new LinkedList();
        boolean RETVAL = THIS.add(t0);
        MyInput t1 = new MyInput(0);
        boolean RETVAL1 = THIS.add(t1);
```

4.2 State Slicing

Given a concrete state and a member field of the class, we produce an abstract state represented by the value of the member field and the values of all those fields reachable from the member field if the member field is of a reference type. For example, in the end of Tests 1 and 2, the `THIS` object's concrete states are represented by the following object-field values:

```
Concrete object state at the end of Test 1:
 size=1;
 modCount=1;
 serialVersionUID=876323262645176354;
 header.element=null;
 header.next.element.v=0;
 header.next.next=header;
 header.next.previous=header;
 header.previous=header.next;

Concrete object state at the end of Test 2:
 size=1;
 modCount=1;
 serialVersionUID=876323262645176354;
 header.element=null;
 header.next.element.v=7;
 header.next.next=header;
 header.next.previous=header;
 header.previous=header.next;
```

When we slice these concrete object states by the `size` field, both abstract-state representations are "`size=1;`" and these two nonequivalent concrete states are mapped to the same abstract state. After we generate abstract states at the entry and exit of a method call, we generate a transition (characterized by the method call) from

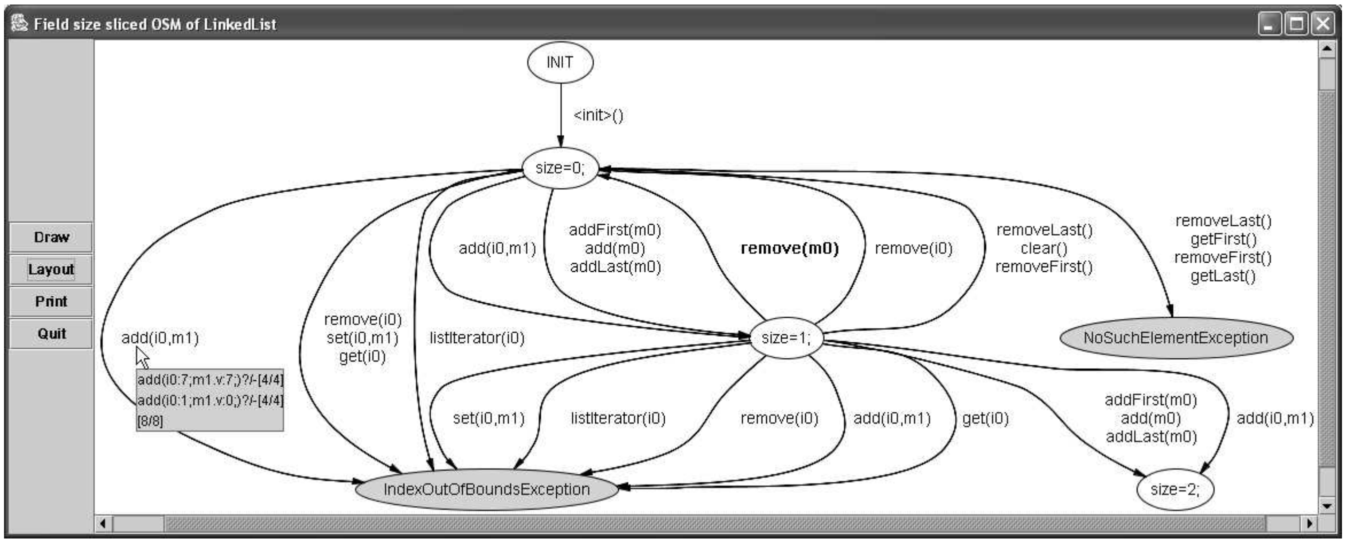

Figure 4: A LinkedList OSM sliced by the `size` field

the abstract state at the method entry to the abstract state at the method exit. Then we can construct an abstract OSM from test executions. Figure 4 shows a LinkedList OSM sliced by the `size` field (displaying also exception states and transitions to them). Figure 5 shows a LinkedList OSM sliced by the `modcount` field (without displaying exception states or transitions to them). [4] We allow the user to configure whether to display exception states and transitions to them in a sliced OSM. By default, we do not display state-preserving transitions in a sliced OSM in order to present a succinct view. In Figure 4, the transition starting from the top "`INIT`" state is marked with `<init>()`, which represents a constructor call. In general, each transition edge in an OSM is marked with a simplified representation of the method name and signature that correspond to the method calls of the transition. When there are multiple nonequivalent argument lists of the same method transiting one state to another, we group them into one single transition edge. This grouping mechanism can be viewed as a form of abstraction on transitions. When the user move the mouse cursor over the edge, the details of method calls are displayed. For example, the leftmost edge in Figure 4 shows the simplified method name and signature for `add(`**`int`**` index, MyInput element)`: `add(i0, m1)`, where each parameter is represented as the combination of the first letter of its type name and its parameter order (starting from 0). The details of method calls in this left-most transition are:

```
add(i0:7;m1.v:7;)?/-[4/4]
add(i0:1;m1.v:0;)?/-[4/4]
[8/8]
```

where `m1.v` represents the `v` field of the second argument, argument values or argument's field values are shown following their argument names or argument's field names separated by "`:`", and different arguments or fields are separated by "`;`". For succinctness, we do not display the "not_null" value for a non-null reference-type field ("not_null" assignments are described in Section 3). A line of description for method calls is in the form of $m?/mr![tc/ec]$ where m is the method call name and argument values, mr is the return value if any (if a return is void or the method call throws an exception, we display the return value as "–" and we do not display "!"), tc is the transition count, and ec is the emission count (the descriptions of transition counts and emission counts are described in Section 3). In the bottom line of the detailed description, we summarize the total number of transition counts and emission counts for all the method calls in the transition. When the method calls in the transition exercise all existing argument lists for the method, we additionally display "ALL_ARG", such as in the details for a `remove(m0)` in Figure 5. To present a more succinct view, we group calls of different methods with the same starting state and ending state into a single transition edge if these method calls satisfy the following two properties: (1) the calls of each method exercise all existing argument lists for the method (displayed with "ALL_ARG"); (2) the calls of each method are deterministic (their transition counts are equal to their emission counts). For indeterministic transitions, we highlight their simplified method names and signatures in bold font. For example, one edge of `remove(m0)` is highlighted in central Figure 4. This indeterminism indicates that invoking `remove(m0)` on a linked list containing one element does not necessarily make the linked list empty. For example, one such case is to remove an element with the value of 0 from a linked list containing an element with the value of 7.

Extracted sliced OSM's provide succinct views for summarizing interesting state-transition behavior exhibited by a component. For example, by inspecting and exploring Figure 4, we can conveniently understand the conditions of throwing uncaught exceptions, which often indicate the sequencing constraints of using a component. For example, an `IndexOutOfBoundsException` is thrown when invoking `get(i0)` immediately after invoking a constructor. Previous research in inferring sequencing constraints [1, 22, 28] could be effective in inferring this simple constraint but might not be able to infer more complex constraints extracted by our approach. One such a complex constraint is that if we invoke a constructor, `add(m0)`, `removeLast()`, and finally `get(i0)`, an `IndexOutOfBoundsException` is thrown. The reasons are that previous research in inferring sequencing constraints does not consider the internal states of a component but only the sequence order among method calls invoked through a component interface.

By looking into the details of those transitions leading to the `IndexOutOfBoundsException` state, we can understand that if a method argument is an integer index to a linked list, it shall generally fall into the scope between zero and the size of the list. But

[4]We do not show the LinkedList OSM sliced by the `serialVersionUID` field in this paper because the class does not modify `serialVersionUID` and the extracted OSM is trivial.

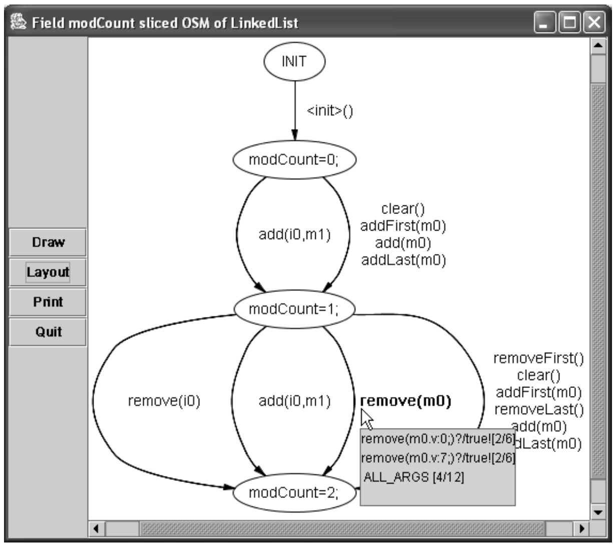

Figure 5: A LinkedList OSM sliced by the `modCount` field

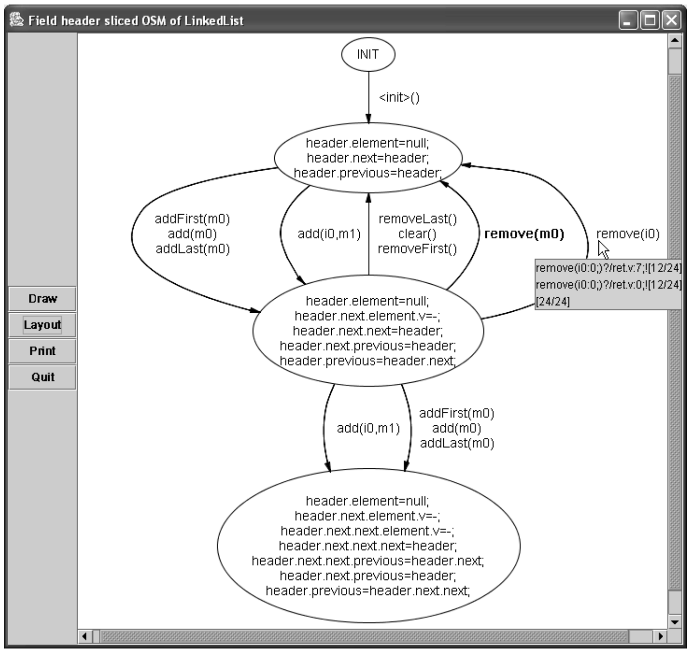

Figure 6: A LinkedList OSM sliced by the `header` field after structural abstraction

one difference has caught our attention: `add(i0, m1)` in the left-most of Figure 4 is not grouped with other method calls with index arguments on the second-to-leftmost edge of Figure 4, such as `remove(i0)` and `set(i0, m1)`; this indicates that all argument lists for methods on the second-to-leftmost edge lead the "`size=0;`" state to the "`IndexOutOfBoundsException`" state, but not all argument lists for `add(i0, m1)` lead to the exception state. By inspecting their details, we found that, to avoid the exception, the `i0` argument for `add(i0, m1)` should satisfy `(0 <= i0 && i0 <= size())` but the `i0` argument for the methods on the second-to-leftmost edge should satisfy `(0 <= i0 && i0 < size())`. We also found that `listIterator(i0)` needs to satisfy the same constraint as `add(i0, m1)`. We have confirmed these small distinctions among exception-throwing conditions by browsing Java API documentation [17].

4.3 Structural Abstraction

When we slice two concrete object states in the end of Tests 1 and 2 by the `header` field, these two nonequivalent concrete object states are still mapped to two different abstract states. After we slice all exercised concrete object states by the `header` field, we reduce 11 concrete object states to 7 abstract states, whose corresponding OSM is still complex. Inspired by Korat's object graph isomorphism [3], we conduct *structural abstraction* by keeping only structural information among object fields but ignoring those primitive field values in a sliced state. The underlying rationale for this technique is that object states sharing the same object graph structure often exhibit certain common behavior. For example, after we apply structural abstraction on `header`-sliced states in the end of Tests 1 and 2, we produce the same abstract state as below:

```
header.element=null;
header.next.element.v=-;
header.next.next=header;
header.next.previous=header;
header.previous=header.next;
```

In the representation of abstract states, we replace all field values of primitive types with "–". In fact, we have found that the generated abstract states have a one-to-one correspondence with the states sliced by the `size` field. For example, the `header`-sliced state after structural abstraction in the end of Tests 1 and 2 corresponds to the "`size=1;`" state. Figure 6 shows a LinkedList OSM sliced by the `header` field after structural abstraction (without displaying exception states or transitions to them). This OSM is especially useful for another implementation of a linked list that does not have a `size` field but computes the size on the fly from the `header` field when the size's value is needed. For other data structures such as a binary tree, one `size`-sliced abstract state might map to more than one sentinel-node-sliced abstract states after structural abstraction.

5. DISCUSSION AND FUTURE WORK

There are two main factors that affect our approach's usability in practice: member fields and generated tests. In our approach, member fields take the role of abstraction functions [14], which are used to specify state abstractions. In addition, like other dynamic inference techniques [1, 7, 11, 22, 27, 28], the quality or complexity of an extracted sliced OSM depends on the executed tests besides the characteristics of the used member field. Section 5.1 and 5.2 further discuss the factors of member fields and generated tests, respectively. Section 5.3 discusses other potential applications of our approach than the task of understanding component behavior.

5.1 Member Fields

Our approach uses a single member field as an abstraction function: different concrete states with the same value for the member field are abstracted to the same abstract state. Although we construct a sliced OSM for each member field, we might abstract away some aspects of concrete states that are central in understanding the behavior of a method in a sliced OSM. For example, in some classes, some member fields might be closely coupled and we might prefer to slice states by multiple member fields instead of a single member field. To provide tool supports for these cases, we can categorize member fields into groups based on field-access patterns by member methods using concept analysis [5]. Then we can slice states by these field groups and use sliced states to construct sliced OSM's. On the other hand, the state abstraction based on state slicing might not be high level enough; therefore, the resulting OSM's might be still too complicated for inspection.

In some cases, it might be difficult to infer a good abstraction function from the code itself by using various heuristics. Then in order to get satisfactory OSM's, we might need human inputs for defining indistinguishability properties [10] or other forms of abstraction functions to further abstract states. We expect that this way of getting human inputs in our approach shall be better for many types of programs than requiring upfront human inputs in traditional formal methods. First, we expect that programmers would be more willing to provide their inputs of abstraction functions after they have already seen OSM's extracted without their upfront inputs (some OSM's could have already been useful for them to understand parts of the component behavior). Second, we expect that it would be easier for programmers to formulate abstraction functions based on the crude OSM's extracted by our approach.

5.2 Generated Tests

There are two controllable configurations on the tests generated by our approach: method arguments and the maximum iteration number. When we use another third-party tool to generate more method arguments for a method but keep the same maximum iteration number as two, the sliced OSM's for LinkedList in Figure 4, 5, and 6 would be kept mostly the same (details associated with transitions might grow though) but the `header`-sliced OSM before structural abstraction would grow rapidly. When we keep the same method arguments but increase the maximum iteration number, the sliced OSM's in Figure 4, 5, and 6 would grow linearly. For example, in Figure 4, there will be new transitions starting from the bottom-right "`size=2;`" state similar to the ones starting from the "`size=1;`" state. In general, when there are more method arguments or higher maximum iteration numbers, the space of both concrete states and sliced states could grow. To address the scalability of the approach, programmers can configure fewer method arguments or lower maximum iteration numbers, or specify user-defined abstraction functions to further abstract states (discussed in Section 5.1).

If the generated tests used for OSM extraction are not of good quality, the quality of extracted sliced OSM's can be compromised. Static analysis techniques can be used to identify some insufficiency of generated tests for extracting sliced OSM's. For example, because Jtest 5.1 generates only an empty collection argument for `addAll(int index, Collection c)`, the `addAll` method is dynamically identified as a state-preserving method for all extracted sliced OSM's. Existing static techniques for method-purity analysis [2,16] can identify `addAll` not to be state preserving; then we can augment Jtest-generated tests with non-empty-collection arguments for `addAll`.

5.3 Other Applications

Although in this paper we primarily investigate the extraction of sliced OSM's to help understand component behavior, there are other promising applications of extracted OSM's. For example, we can extract sliced OSM's from existing generated tests to ease the task of test inspection. We can use extracted OSM's to guide test generation using existing finite-state-machine-based testing techniques [13], use new generated tests to further improve extracted OSM's, and then use new improved OSM's to generate more new tests and so forth. During iterations, any new generated tests violating existing inferred properties (e.g. OSM's) can be selected for inspection [26]. These iterations form a feedback loop between test generation and specification inference proposed in our previous work [25].

We can apply sliced OSM's in testing and verification by extrapolating unseen states and transitions based on observed states and transitions. Then the prescribed component behavior is not limited to observed one. For example, in Figure 4, we can predict the structure of transitions around the unseen "`size=3;`" state or other unseen states.

After we have extrapolated initial sliced OSM's, we can perform conformance checking between OSM's and the implementation, which is similar to conformance checking between abstract state machines and an implementation [8]. We can also explore ways of translating properties captured by OSM's to the forms understood by existing software model checking tools [4, 20] and use existing tools to verify programs against their extracted OSM's. Note that finding counterexamples does not necessarily expose bugs in programs but might expose insufficiency of originally generated tests for OSM extraction. These counterexamples can help generate new tests to augment existing generated tests.

Because we extract sliced OSM's from an implementation, if the implementation is faulty and the initial sliced OSM's exhibit wrong behavior, we might not expose faults by performing conformance checking between OSM's and the implementation. Therefore, before we extrapolate initial sliced OSM's, we might prefer human inspection on the initial sliced OSM's to make sure that the initial sliced OSM's exhibit expected behavior.

6. RELATED WORK

Our previous work develops the observer abstraction approach for extracting OSM's (called observer abstractions) from unit-test executions [27]. The observer abstraction approach uses the return values of observers invoked on a concrete object state as abstract state representation, whereas our new approach in this paper uses the values of a member field in a concrete object state as abstract state representation. Unlike the observer abstraction approach, our new approach does not require the availability of (good) observers. The complexity of an observer abstraction depends on the characteristics of its corresponding observers, whereas the complexity of a sliced OSM depends on the characteristics of its corresponding member field. Observer abstractions help investigate behavior related to the return values of observers and this type of behavior is not explored in our new approach. In the LinkedList example, in contrast to four sliced OSM's generated by our new approach, the observer abstraction approach generates 18 observer abstractions. One observer is `int size()`; therefore, the extracted `size()` observer abstraction is exactly the same as our `size`-sliced OSM.

From system-test executions, Whaley et al. dynamically extract Java component-interface models, each of which accesses the same field [22]. They statically determine whether a method is a state-modifying one. In their extracted models, they assume that the same state-modifying method transits an object to the same abstract state. This assumption makes the extracted models less accurate than our approach. Ammons et al. mine protocol specifications in the form of a finite state machine from system-test executions [1]. Although their approach uses data dependence to extract relevant API method calls, it does not use component internal states but use the sequence order among API method calls for learning models. Both Whaley et al. and Ammons et al.'s approaches usually require a good set of system tests for exercising component interfaces, whereas our approach receives a given component and generates a set of tests to exercise component's object states in a small scope. Because their approaches do not consider object state information but just sequence order among API method calls, applying Whaley et al.'s approach on our generated unit tests would yield a complete graph of methods that modify the same object field and applying Ammons et al.'s approach on our generated unit tests would yield a complete graph of all methods in the component interface.

Yang and Evans infer temporal properties in the form of the strictest pattern any two methods can have in execution traces [28]. Similar to Whaley et al. and Ammons et al.'s approaches, their approach considers only sequence order among method calls without considering internal states of a component, whereas our approach use sliced states to construct OSM's, which encoded more accurate sequencing constraints. In addition, their approach considers sequencing relationship between two methods, whereas our approach considers state-transition relationship among multiple methods.

Ernst et al. develop Daikon to dynamically infer likely invariants from test executions [7]. These invariants describe the observed relationships among the values of object fields, arguments, and returns of a single method in a component interface, whereas our sliced OSM's describe state-transition relationships among multiple methods in a component interface and use the values of fields reachable from a member field to represent object states. Henkel and Diwan discover algebraic specifications from the execution of automatically generated unit tests [11]. Their discovered algebraic specifications usually present a local view of relationships between two methods, whereas our sliced OSM's present a global view of relationships among multiple methods.

Corbett et al. develop Bandera to extract finite-state models from Java source code for model checking [4]. Given a property, Bandera's slicing component removes control points, variables, and data structures that are irrelevant for checking the property. For each member field of a component, our approach dynamically slices object states that are reachable from the member field and constructs a sliced OSM. Given a definition of an abstraction, Bandera's abstraction-based specializer transforms the source code into a specialized version by replacing concrete operations and tests on relevant concrete data with abstracted versions on abstract values. Our approach conducts structural abstraction on a sliced state by mapping all primitive values in the state to the same abstract value.

Grieskamp et al. allow the user to define indistinguishability properties to group infinite states in abstract state machines into equivalence classes, called hyperstates [10]. Their tool incrementally produces finite state machines by executing abstract state machines. Our approach use the values of a member field to group concrete object states into abstract states in a sliced OSM.

Kung et al. statically extract object state models from class source code and use them to guide test generation [12]. An object state model is in the form of a finite state machine: the states are defined by value intervals over object fields, which are derived from path conditions of method source; the transitions are derived by symbolically executing methods. Our approach dynamically extracts sliced OSM's from test executions and supports a much wider range of classes than Kung et al's approach. For example, Kung et al.'s approach could not extract any state models for the `header` field because `header`'s values cannot be characterized by value intervals, which are usually applicable for primitive numeric fields. Their approach could not extract any model for the `modeCount` field because there is no usable path condition for this integer field in the source code. Because of the code complexity, their approach would have difficulties in symbolically deriving transitions for the states extracted from the only path condition usable for their approach: `(size==0)`.

Turner and Robson use finite state machines to specify the behavior of a class [19]. The states in a state machine are defined by the values of a subset or complete set of object fields. The transitions are method names. Although both their specified finite state machines and our sliced OSM's are in a similar form, we automatically extract state machines from test executions, whereas they manually specify state machines for a class. Edwards develops an approach of generating tests based on flowgraphs extracted from a component's specifications [6]. A flowgraph is a directed graph where each node represents one method provided by the component and a directed edge from a node `n` to node `n'` represents the possibility that control may flow from `n` to `n'`. Our approach automatically extracts OSM's from test executions without requiring a priori specifications and our OSM's capture actual-state transition.

7. CONCLUSION

Lack of specifications for a component has posed the barrier to the reuse of the component in component-based software development. In this paper, we have proposed a new approach for automatically extracting sliced OSM's for component interfaces. Given a component such as a Java class, we generate a set of tests for the component. Then we slice exercised concrete object states by each member field of the component and construct OSM's based on the sliced states. These sliced OSM's provide useful state-transition information for inspection. These OSM's also have potential for component verification and testing.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work was supported in part by the National Science Foundation under grant ITR 0086003. We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

8. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *Proc. 6th Workshop on Formal Techniques for Java-like Programs*, June 2004.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. the 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[5] U. Dekel and Y. Gil. Revealing class structure with concept lattices. In *Proc. 10th IEEE Working Conference on Reverse Engineering*, pages 353–365, 2003.

[6] S. H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, 10(4):249–262, 2000.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[8] Foundations of Software Engineering, Microsoft Research. Abstract state machine language. `http://research.microsoft.com/fse/AsmL`.

[9] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering.

Software: Practice and Experience, 30(11):1203–1233, Sept. 2000.

[10] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[12] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proc. 18th International Computer Software and Applications Conference*, pages 222–227, 1994.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. The IEEE*, volume 84, pages 1090–1123, Aug. 1996.

[14] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[15] Parasoft. Jtest manuals version 5.1. Online manual, July 2004. `http://www.parasoft.com/`.

[16] A. Rountev. Precise identification of side-effect-free methods in Java. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 82–91, Sept. 2004.

[17] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[19] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proc. International Conference on Software Maintenance*, pages 302–310, 1993.

[20] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, 2000.

[21] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering*, pages 439–449, 1981.

[22] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. the International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[23] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.

[24] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[25] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

[26] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[27] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, Nov. 2004.

[28] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proc. the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 23–28, 2004.