Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage

Hai Yuan
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
hyuan3@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract

Some requirement modelling languages such as UML's statechart diagrams allow developers to specify requirements of state-transition behavior in a visual way. These requirement specifications are useful in many ways, including helping program understanding and specification-based testing. However, there are a large number of legacy systems that are not equipped with these requirement specifications. This paper proposes a new approach, called Brastra, for extracting object state machines (OSM) from unit-test executions. An OSM describes how a method call transits an object from one state to another. When the state of an object is represented with concrete-state information (the values of fields transitively reachable from the object), the extracted OSMs are simply too complex to be useful. Our Brastra approach abstracts an object's concrete state to an abstract state based on the branch coverage information exercised by methods invoked on the object. We have prototyped our Brastra approach and shown the utility of the approach with an illustrating example. Our initial experience shows that Brastra can extract compact OSMs that provide useful information for understanding state-transition behavior.

1 Introduction

The Unified Modelling Languages (UML) [15] provides a set of notations for describing requirements of artifacts in software systems. Among these notations, statechart diagrams capture state-transition behavior of a class or multiple classes. After requirements specifications are specified, developers can write source code to implement the specified behavior. Later when developers want to understand and maintain the source code, they can refer to requirements specifications besides directly inspecting the source code. In addition, developers can use specification-based testing tools [6, 8, 12, 17] to generate test inputs from the specifications and check the behavior of implementation with the behavior specified in requirements specifications. However, a number of legacy systems are not equipped with specifications. Understanding and testing these legacy systems present a challenge for developers. To address this challenge, researchers have developed various reverse engineering techniques [11] to infer various types of information from legacy systems.

This paper proposes Brastra, a new approach for automatically extracting object state machines (OSM) [21] for a class from unit-test executions. These OSMs describe state-transition behavior exhibited by invoking methods on objects of a class. An OSM is similar to a UML statechart diagram. In an OSM for a class, a state represents the state of an object at runtime. A transition represents method calls invoked on an object, transiting the object from one state to another. States in an OSM can be concrete or abstract. A concrete state of an object is characterized by the values of object fields transitively reachable from the object. Because a concrete OSM is often too complicated to be useful, our previous work [21, 22] has developed techniques to abstract concrete states to abstract states, which are used to construct abstract OSMs. Our previous observer-abstraction approach [21] represents an abstract state of an object with the return values of observer methods (methods whose returns are not void) invoked on the object. Our previous sliced-OSM-extraction approach [22] represents an abstract state of an object with the values of a specific field. In this paper, we have developed the new Brastra approach that does not require appropriate observer methods in class interface (required by our previous observer abstraction approach [21]) or appropriate object-field structure (required by our previous sliced OSM extraction approach [22]).

The Brastra approach represents an abstract state of an object with the branch coverage information produced by methods invoked on the object. OSMs produced by Brastra capture program behavior exhibited by branching points in method body, complementing program behavior exhibited by observer methods or specific fields (captured by our previous approaches). We have implemented the Brastra approach and demonstrated its utility by applying it on an illustrating example. Our initial experience shows that OSMs extracted by Brastra are compact and useful for providing insights to state-transition behavior.

The rest of this paper is organized as follows. Section 2 presents an illustrating example. Section 3 introduces the formal definition of an OSM. Section 4 illustrates our new approach for extracting OSMs based on branch coverage information. Section 5 introduces our implementation of the approach. Section 6 discusses issues of the approach and lays out future directions. Section 7 reviews related work, and Section 8 concludes.

2 Example

As an illustrating example, we use a data structure: a `UBStack` class, which is the implementation of a bounded stack that stores unique elements of integer type. Figure 1 shows the class including two standard stack operations: `push` and `pop`. Stotts et al. coded this Java implementation to experiment with their algebraic-specification-based approach for systematically creating unit tests [16]. In the class implementation, the `max` is the capacity of the stack, the array `elems` contains the elements stored in the stack, and `numberOfElements` is the number of the elements and the index of the first free location in the stack.

The `push` method first checks whether the element to be pushed exists already in the stack. If the same element already exists in the stack, the method moves the element to the top of the stack. Otherwise, the method increases `numberOfElements` after writing the element into the `elems` array if `numberOfElements` does not exceed the stack capacity `max`. If the stack capacity is exceeded, the method prints an error message and makes no changes on the stack. The `pop` method first checks whether `numberOfElements` is greater than zero. If so, it retrieves the top element of the stack, decreases `numberOfElements`, and returns the retrieved element; otherwise, the method prints an error message and returns `-1` as an error indicator.

To generate tests for `UBStack`, we first manually configure `push`'s arguments to be 1, 2, 3, or 4.[1]. Given the bytecode of `UBStack` our previously developed Rostra tool [19] automatically generates 263 tests; these generated tests exercise 41 non-equivalent concrete object states (two concrete object states are non-equivalent if their concrete state representations are different).

[1]We can use some existing test generation tools such as Parasoft Jtest [13] or JCrasher [2] to automatically generate method arguments for `UBStack`, but these tools may not generate relevant method arguments.

```java
public class UBStack {
  private int max;
  private int[] elems;
  private int numberOfElements;

  public UBStack() {
    numberOfElements = 0;
    max = 3;
    elems = new int[max];
  }
  public void push(int k) {
    int index;
    boolean alreadyMember = false;
    for(index=0; index<numberOfElements; index++) {
       if (k==elems[index]) {
          alreadyMember = true;
          break;
       }
    }
    if (alreadyMember) {
       for (int j=index; j<numberOfElements-1; j++)
           elems[j] = elems[j+1];
       elems[numberOfElements-1] = k;
    } else {
       if (numberOfElements < max) {
          elems[numberOfElements] = k;
          numberOfElements++;
          return;
       } else {
          System.out.println("Stack full, cannot push");
          return;
       }
    }
  }
  public int pop(){
    int ret = -1;
    if (numberOfElements > 0) {
         ret = elems[numberOfElements-1];
         elems[numberOfElements-1] = 0;
         numberOfElements --;
    } else {
       System.out.println("Stack empty, cannot pop");
    }
    return ret;
  }
}
```

Figure 1. A bounded-stack implementation that accommodates unique integer elements

3 Object State Machine

In our previous work [21], We have defined an object state machine for a class:

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. $\delta : S \times I \rightarrow P(S)$ is the state transition function and $\lambda : S \times I \rightarrow P(O)$ 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 $\lambda(s, i)$.*

The object states in an OSM can be concrete or abstract. In a concrete OSM, states of an object are represented by its concrete-state representation. An object's concrete-state representation is characterized by the values of all the field transitively reachable from the object [19]. Because some object fields may be reference types and their values point to memory addresses (which can be different in different runs of the same test), we use a linearization algorithm [19] to collect the values of these reference-type fields so that comparing state representations takes into account comparing object-graph shapes but without directly comparing memory addresses. Two states are *equivalent* if their state representations are the same, and are *nonequivalent* otherwise.

For example, the generated tests for `UBStack` exercise 41 nonequivalent concrete object states. There are 142 transitions among these states. Figure 2 shows a concrete OSM exercised by generated tests and Figure 3 shows a detailed view of the highlighted area in Figure 2. The OSM is displayed by using the Grappa package, which is part of graphviz [5]. States in the OSM are shown as circles in Figure 3 and the labels inside these circles are the state representations, which include field names followed by ":" and corresponding field values (array-element values are separated by ";"). The three states in Figure 3 represent three full stacks. Although they have the same set of stack elements, these elements are stored in three stacks in different orders. Transitions in the OSM are shown as directed edges that connect circles (states). These edges are labelled with method names and arguments (for brevity, we do not show method return values in the edge labels).

We have observed that the concrete OSM is too complex to be useful in practice. Although we can zoom in to view details of object states and transitions among them, these details in such a large OSM are often not very useful for program understanding or test-result inspection.

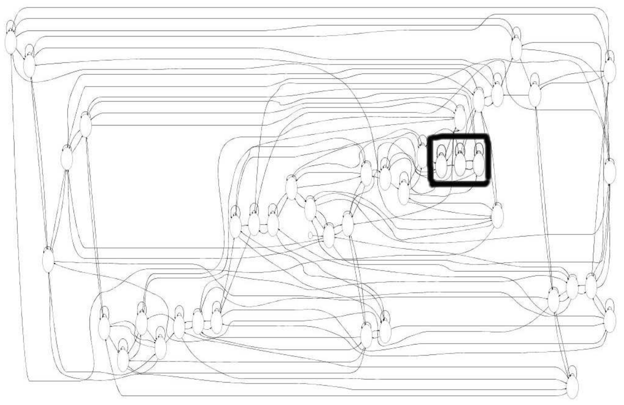

Figure 2. An overview of `UBStack` concrete OSM (containing 41 states and 142 transitions) exercised by generated tests

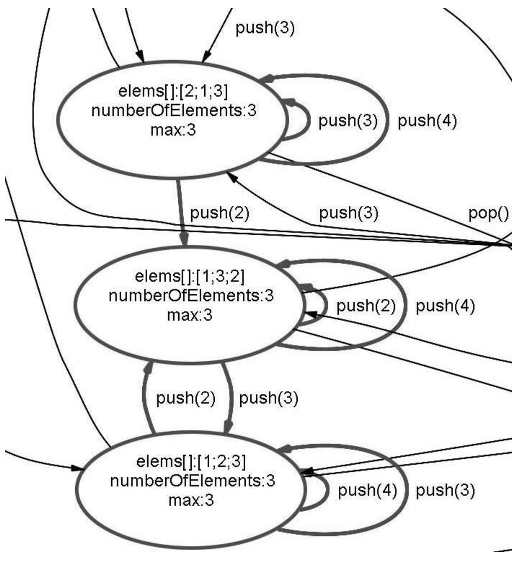

Figure 3. A detailed view of the selected area in `UBStack` concrete OSM

4 Approach

To reduce the complexity of an OSM, we can construct an *abstraction function* [10] to map concrete states to abstract states. Our Brastra approach constructs such an abstraction function by using branch coverage information. We first define the branch coverage we shall use in representing an abstract state of an object.

A method m is characterized by its defining class c, method name and method signature. Then we define conditional set for a method m.

Definition 2 Conditional set CS *of a method* m *are a set of strings, including all the conditional strings (together with their source-code-line numbers) that appear in the body of* m*,* m*'s direct and indirect callees.*

A method call mc is a pair $\langle m, a\rangle$ where m is a method and a is a vector of method-argument values.

Definition 3 *Given an object* o *of class* c *and a method call* mc:$\langle m, a\rangle$ *of* c*, assume* CS *is the conditional set of* m*,* branch coverage BC *of* mc *on* o *is a map from* CS *to* $\{true, false, both, n/a\}$*, where the map is defined based on whether a conditional's false branch, true branch, both branches, or neither branch is covered during the execution of* mc *on* o.

Definition 4 *Given an object* o *of class* c *and a set of* c*'s method calls* $MC = \{mc_1, mc_2, ..., mc_n\}$*, the abstract state of* o *with respect to* MC *is represented by* $\{BC_1, BC_2, ..., BC_n\}$*, where* BC_i *is branch coverage of* mc_i *on* o.

Then we construct an abstract OSM where all states are abstract states with respect to MC.

For example, assume MC for `UBStack` is {`pop(), push(1), push(2), push(3), push(4)`} and consider the following tests:

```
Example Test:
   UBStack s = new UBStack();
   s.push(1);
   s.pop();
   s.push(2);
   s.push(3);
   s.push(4);
```

After the end of the constructor call, if we invoke `pop()` on s, the `pop` method execution covers the false branch of the following conditional: `(numberOfElements > 0)`. We represent the map of `(numberOfElements>0)` $\rightarrow$ `false` as `!(numberOfElements>0)`.

To simplify illustration, we do not display source-code line numbers for conditional strings. When a conditional `c` is mapped to `both`, which indicates both branches of the conditional are covered, we simply represent the mapping with two entries `c` and `!c`.

After the end of the constructor call, if we invoke any of `push(1), push(2), push(3), and push(4)` on s, the `push` method execution covers the following branches following the preceding notations:

```
!(index<numberOfElements)
!(alreadyMember)
numberOfElements < max
```

Figure 4 shows the abstract OSM extracted by Brastra based on branch coverage information. The top state is labelled as `INIT`, which indicates no state before invoking a constructor. Then we represent the abstract state after the constructor call as the second to top state of the abstract OSM shown in Figure 4. On the top part of the state, we display the object field values that are common to all the concrete states represented by the the abstract state. Then we display the branch coverage for `pop` (we put method name `pop` before the first line of branch coverage). Finally we display the branch coverage for `push`. To simplify the view, we do not display the method arguments or returns on the transitions in the OSM.

Interesting behavior occurs when we abstract the concrete states resulting from invoking `push(1)` or `push(2)` on an empty stack (note that in the example test, push(2) is actually invoked on an empty stack because its preceding method call `pop()` counteracts the effect of `push(1)`, transiting the state to an empty stack). On a concrete state resulting from `push(1)`, invoking `push(1)` again follows a path different from invoking `push(2)`, because the stack stores only unique elements. Therefore, we can observe in the middle state of Figure 4 there are two different branch coverage for `push`: one representing the case where the `push`'s argument has already existed in the stack and the other representing the case where the `push`'s argument does

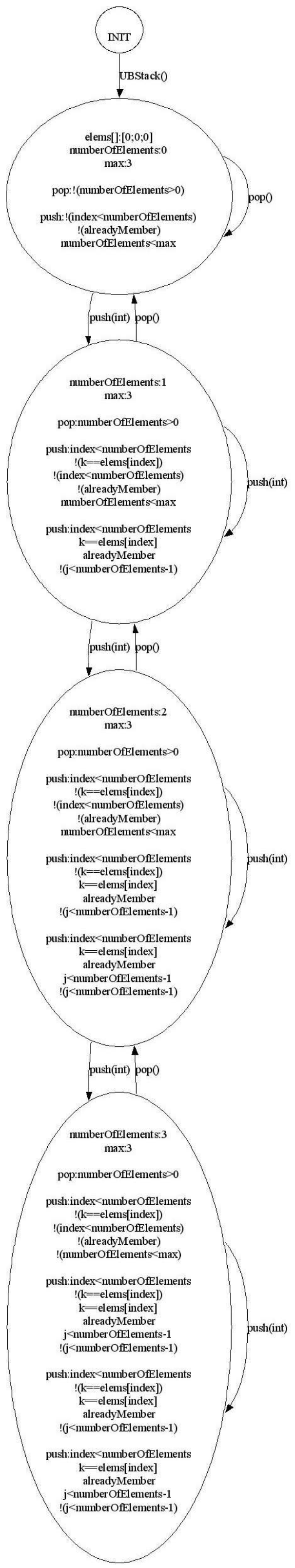

Figure 4. An overview of `UBStack` abstract OSM based on branch coverage (containing 4 states and 11 transitions) exercised by generated tests

not exist in the stack. The branches of `alreadyMember` and `!alreadyMember` from two sets of `push` branch coverage give us hints on these two cases.

The second to the bottom state has three sets of different `push` branch coverage, in addition to one set of `pop` coverage. The first set represents the case where the `push`'s argument does not exist in the stack, the second set represents the case where the `push`'s argument exists in the stack and the existing element is on the top of the stack (therefore, the element is not required to be moved to the top), and the third set represents the case where the `push`'s argument exists in the stack and the existing element is not on the top of the stack (therefore, the element is required to be moved to the top). In the example test, the concrete state of `s` after invoking `push(3)` falls into this abstract state.

The bottom state indicates a full stack; therefore, no `push` method call can further change the object state. Note that because a full stack with different concrete states can contain different elements; therefore, unlike in the second to top state, we do not display the values of the `elems[]` field. In the example test, the concrete state of `s` after invoking `push(4)` falls into this abstract state.

5 Implementation

Given a class, our Rostra tool [19] generates test inputs to exhaustively exercise object states iteratively. In particular, if users provide some sample method arguments, Rostra can use them; otherwise, Rostra uses Parasoft Jtest [13] or JCrasher [2] to generate method arguments. Then Rostra uses these method arguments to explore the object state space iteratively. Tool users can configure the maximum iteration number for Rostra to explore the state space. For `UBStack`, which has capacity of three, four iterations are sufficient to explore all possible states with the method arguments of `pop`, `push(1)`, `push(2)`, `push(3)`, and `push(4)`. Note that the Rostra's bounded-exhaustive test generation enables a better inspection of OSMs extracted from generated-test executions. For example, when tool users find out that an expected transition is missing in OSMs, it can have two reasons: a test that is required to produce that transition is missing or there is a bug in the program. Rostra's bounded-exhaustive test generation reduces the chance of the former case. In addition, Rostra's bounded-exhaustive test generation also facilitates our abstraction based on branch coverage. In order to abstract a concrete state, we need specific method calls to be invoked on the concrete state; these method calls are generated by Rostra. Note that when we invoke a method call on a concrete state in order to abstract the concrete state, the method call could modify the concrete state and later method calls on this concrete state need a reproduction of the concrete state; reproductions of concrete states are also supported by Rostra.

After Rostra generates test inputs and exports them into JUnit [4] test classes, we run these test classes with our previously developed Jusc [23] tool, a Java unit-test selection tool based on residual structural coverage [14], to output a path trace file after program executions terminate. We developed a tool to postprocess the collected path trace file to collect branch coverage information. Note that we collect branch hit coverage; therefore, when there are loop iterations during program executions, we do not count how many times a branch is hit nor collect execution orders among branches. This design decision provides further abstraction of states.

In addition, we also use the Daikon [3] Java frontend to run these test classes and collect object states exercised by these tests. Daikon [3] is a tool that dynamically detects likely program invariants in the program executions. It can collect object-field values during program executions, and reports properties that hold true on these fields during the executions. In our approach, we use Daikon to collect object states during program executions and later use these states to extract common field values among concrete states represented by an abstract state and then display the common field values in the state as an annotation.

6 Discussion and Future Work

Two main factors may affect our approach's usability in practice: methods' control flow graphs and generated test inputs. Branching points in control flow graphs take the role of abstraction functions [10]. Although different implementations of the same program behavior can have different control flow graphs, their implied behavior can be similar across different implementations. As is discussed in Section 7, we found that branch coverage information seems to more faithfully reflect interesting program behavior than our previous observer-abstraction approach [21] or sliced-OSM-extraction approach [22].

Besides the characteristics of control flow graphs, the executed test inputs can also affect the quality or complexity of an extracted OSM. Rostra's test generation has two controllable configurations: method arguments and the maximum iteration number. But comparing to previous approaches based on object-field values [22] or return values of observers [21], our new approach is less affected by the actual argument values in the generated tests inputs. But at the same time, choosing right argument values are also important. For example, if we choose only two different method arguments for `push` of `UBStack`, we can never reach a full stack state for `UBStack`. The maximum iteration number can have an effect if some boundary states are not exercised by a low maximum iteration number. For example, if we specify the maximum iteration number as three, we cannot exercise a full stack state for UBStack.

There are several future directions for us to extend the Brastra approach. First, we plan to adapt the existing finite-state-machine-based testing techniques [9] or testing techniques based on UML statechart diagrams [6, 8, 12, 17]. Extracted OSMs can guide further test generation to improve OSM extraction. These iterations form a feedback loop between test generation and specification inference proposed in our previous work [20].

Second, we plan to extend our specification inference for multiple classes instead of a single class. This may require adaptations of our diagram representations as well as inference algorithms.

Finally, we plan to investigate how human inputs can be used to improve the effectiveness of Brastra, which is currently developed as a totally automated tool. For example, when a Brastra-generated OSM is still too complicated to be understandable, developers can configure the state abstraction to be based on only the branches in a specified subset of public methods or the branches that are related to specified object fields. In addition, our Brastra approach is currently a dynamic analysis approach that focuses on functional behavior exercised by a class. There exists research on recovering non-functional requirements from legacy code such as the static analysis approach developed by Yu et al. [24]. In order to identify non-functional requirements, their approach requires some human manipulations of legacy code such as program refactoring. We plan to investigate how human inputs as well as static analysis can guide Brastra to extract non-functional behavior.

7 Related Work

The observer-abstraction approach was developed in our previous work [21]. The observer abstraction approach represents a state of an object by using the return values of observers invoked on the object. When we applied the observer abstraction on UBStack, we could invoke pop, the only observer, on an object and uses pop's return value to abstract the state of the object. By considering pop's semantic, we basically used the element on the top of the stack to abstract the whole stack. This abstraction is not helpful for us to understand UBStack's behavior. The sliced-OSM-extraction approach was developed in our previous work [22]. It uses the values of an object's single field to represent the state of the object. For example, we can use the values of the numberOfElements field to represent states and the resulting OSM is similar to the OSM extracted by Brastra. But when we set the capacity of UBStack to be a large number such as 10, the size of the OSM extracted by using numberOfElements would grow linearly with iteration numbers, whereas the OSM extracted by Brastra keeps the original shape because loop iterations have been abstracted away by our mechanism of considering only branch hit coverage without considering how many times loop iterations are executed.

Kung et al. [7] statically extract object state models from a class's source code and use them to guide test generation. States in a object state model 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. Both their approach and our approach consider branches in method body, but their approach can exploit a limited types of conditionals (e.g., conditionals that compare an object field with a constant) and their approach statically extract state models with a limited capability.

From system-test executions, both Whaley et al. [18] and Ammons et al. [1] mine protocol specifications for component interfaces. They use sequencing order among method calls in the interfaces without using internal object-field values or structural coverage information. Both approaches usually require a good set of system tests for exercising component interfaces, whereas our approach generates test inputs to exercise component's object states in a small scope. Applying their approaches on our generated unit tests for UBStack would yield a circle connecting push and pop.

8 Conclusion

We have proposed Brastra, a new approach for automatically extracting object state machines (OSM) from unit-test executions. Because a concrete OSM extracted based on concrete states is often too complicated to be useful, Brastra abstracts the concrete state of an object by using the branch coverage exercised by methods invoked on the object. We have implemented the Brastra approach and demonstrated its utility on an illustrating example. Our initial experience has shown an OSM extracted by Brastra provides succinct information for understanding key program behavior of a class.

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] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[3] 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.

[4] E. Gamma and K. Beck. JUnit, 2003. http://www.junit.org.

[5] 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.

[6] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEEE Proceedings- Software*, 146(4):197–192, 1999.

[7] 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.

[8] Y. L. L.C. Briand, M. Di Penta. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering*, 30(11), 2003.

[9] 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.

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

[11] H. A. Mueller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse Engineering: A Roadmap. In *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, 2000.

[12] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. 2nd International Conference on the Unified Modeling Language*, pages 416–429, October 1999.

[13] Parasoft Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[14] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. 21st International Conference on Software Engineering*, pages 277–284, 1999.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[16] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.

[17] M. Vieira, M. Dias, and D. Richardson. Object-oriented specification-based testing using UML state-chart diagrams. In *Proc. Workshop on Automated Program Analysis, Testing, and Verification at ICSE 2000*, June 2000.

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

[19] 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.

[20] 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.

[21] 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.

[22] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems at ACM SIGSOFT 2004/FSE-12 (SAVCBS 2004)*, pages 39–46, October 2004.

[23] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[24] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proc. International Conference on Requirements Engineering*, pages 363–372, October 2005.