Automatic Extraction of Abstract-Object-State Machines from Unit-Test Executions

Tao Xie, Evan Martin, and Hai Yuan
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu, {eemartin,hyuan3}@ncsu.edu

ABSTRACT

An automatic test-generation tool can produce a large number of test inputs to exercise the class under test. However, without specifications, developers cannot inspect the execution of each automatically generated test input practically. To address the problem, we have developed an automatic test abstraction tool, called Abstra, to extract high level object-state-transition information from unit-test executions, without requiring a priori specifications. Given a class and a set of its generated test inputs, our tool extracts object state machines (OSM): a state in an OSM represents an object state of the class and a transition in an OSM represents method calls of the class. When an object state in an OSM is concrete (being represented by the values of all fields reachable from the object), the size of the OSM could be too large to be useful for inspection. To address this issue, we have developed techniques in the tool to abstract object states based on returns of observer methods, branch coverage of methods, and individual object fields, respectively. The tool provides useful object-state-transition information for programmers to inspect unit-test executions effectively. In particular, the tool helps facilitate correctness inspection, program understanding, fault isolation, and test characterization.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Experimentation, Reliability, Verification.

Keywords: Software Testing, Program Understanding, Debugging.

1. INTRODUCTION

Given a class, automatic test-generation tools can generate a large number of test inputs, including some valuable corner or special inputs that developers often forget to include in their manually written tests. When developers write specifications for the class, some specification-based test generation tools [3, 8] automatically check the execution of generated tests against the written specifications. Without specifications, developers rely on uncaught exceptions or inspect the executions of generated test inputs in order to determine whether the program behaves as expected. However, it is limited to rely on only uncaught exceptions for exposing faulty behavior and it is impractical to for developers to inspect the executions of a large number of generated test inputs.

To help developers to inspect unit-test executions effectively, we have developed a test abstraction tool, called Abstra, to help developers to inspect object-state-transition behavior. Given a class and a set of existing tests (generated either automatically or manually), Abstra extracts a set of object state machines (OSM): a state in an OSM represents an object state of the class and a transition in an OSM represents method calls of the class. A state in an OSM can be concrete or abstract. A concrete state of an object is characterized by the values of the fields that are transitively reachable from the object. A concrete OSM is an OSM with concrete states.

Because a concrete OSM is often too complicated to be useful for inspection, we have developed three techniques in Abstra for abstracting the concrete states in an OSM to abstract states. An abstract OSM is an OSM with abstract states. These three techniques complement with each other and each of them can be effective for helping inspect specific types of programs. The *observer abstraction* technique [15] uses the return values of observers (public methods with non-void returns) invoked on a concrete object as an abstract state in an OSM. The *branch-coverage abstraction* technique [19] uses the branch coverage of methods invoked on a concrete object as an abstract state. The *state-slicing abstraction* technique [16] uses the values of each member field of the concrete object.

In this demo, we present the abstract OSMs extracted by Abstra and show how developers can inspect these OSMs to improve correctness inspection, program understanding, fault isolation, and test characterization (e.g., identifying the weakness of a test suite).

2. RELATED WORK

Ernst et al. [5] developed the Daikon tool to dynamically infer likely invariants from test executions. Invariants are in the form of axiomatic specifications. These invariants describe the observed relationships among the values of object fields, arguments, and returns of a single method in a class interface, whereas OSMs describe the observed state-transition relationships among multiple methods in a class interface and we have used three techniques to abstract concrete object states; the state-representation techniques except for the state-slicing abstraction technique do not explicitly refer to object fields.

Henkel and Diwan [9] and our previous work [17] discover algebraic specifications from the execution of automatically generated unit tests . These discovered algebraic specifications present a local view of relationships between two methods, whereas OSMs present a global view of relationships among multiple methods. In addition, Henkel and Diwan's approach cannot infer local properties that are related to indeterministic transitions in abstract OSMs and these indeterministic transitions are often useful for inspection. In summary, abstract OSMs are a useful form of behavior representation, complementing algebraic specifications or axiomatic specifications inferred from unit-test executions.

Whaley et al. [13] developed a tool to extract multiple finite state machines as component interfaces from system-test execu-

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

tions. From system-test executions, Yang and Evans [18] developed the Terracotta tool to dynamically infer a program's temporal properties, which are extensions of the response property pattern [4]. Ammons et al. [1] developed a tool to mine protocol specifications in the form of a finite state machine from system-test executions. These three existing tools infer sequencing constraints from system-test executions without taking into account the actual state information whereas the states in abstract OSMs are abstracted from concrete states exercised by unit-test executions.

3. OBJECT STATE MACHINE

To characterize program behavior related to object-state transition, we have defined an object state machine for a class [15]:

DEFINITION 1. *An* object state machine *(OSM)* M *of a class* c *is a sextuple* $M = (I, O, S, \delta, \lambda, INIT)$ *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* $\lambda(s, i)$*.*

When a method call in a class 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.

An OSM can be deterministic or nondeterministic. 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 [14]. 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 [14] 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 *non-equivalent* otherwise.

For example, we can use a test generation tool called Rostra (developed in our previous work [14]) to generate tests for a `UBStack` class, which implements a bounded stack that stores unique elements of integer type [11]. Figure 1 shows a concrete OSM exercised by generated tests, containing 41 states and 142 transitions and Figure 2 shows a detailed view of the highlighted area in Figure 1. States in the OSM are shown as circles in Figure 2 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 2 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.

We have observed that the concrete OSM is too complex to be useful in practice. Although we can zoom in to view details of

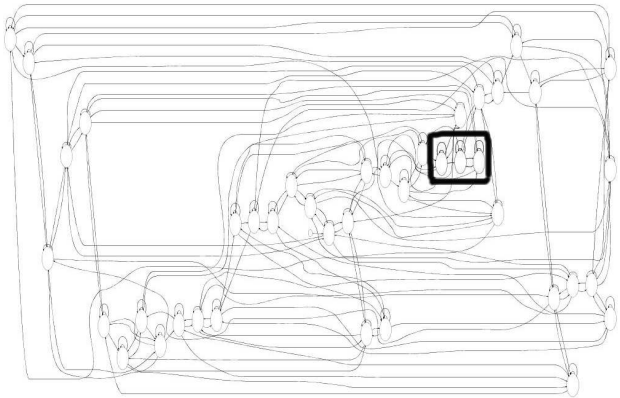

Figure 1: An overview of `UBStack` concrete OSM (containing 41 states and 142 transitions) exercised by generated tests

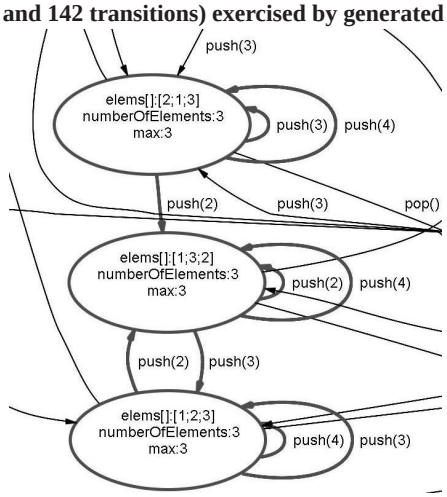

Figure 2: A detailed view of the selected area in `UBStack` concrete OSM

object states and transitions such as in the highlighted area in Figure 1, these details in such a large OSM are often not very useful for inspection.

4. TEST ABSTRACTION

Because concrete OSMs are often too complicated to be useful for inspection, we developed the Abstra tool to abstract concrete states and construct abstract OSMs. An *abstract state* of an object is defined by an *abstraction function* [10]; the abstraction function maps each concrete state to an abstract state. We developed three techniques for constructing abstraction functions based on observer methods invoked on concrete states, branch coverage of methods invoked on concrete states, and individual fields of concrete states.

4.1 Observer Abstraction

The observer abstraction technique constructs an abstraction function by using return values of methods invoked on concrete states.

A method m is characterized by its defining class c, method name, and method signature. A method call mc of a method m is a pair $\langle m, a \rangle$ where a is a vector of method-argument values.

We first define an observer method following previous work on specifying algebraic specifications for a class [9]:

DEFINITION 2. *An* observer method *of a class* c *is a method* ob *in* c*'s interface such that the return type of* ob *is not void.*

Given a class c and a set of observer-method calls $OB = \{ob_1, ob_2, ..., ob_n\}$ of c, the observer abstraction technique constructs an abstraction of c with respect to OB. In particular, a concrete state cs is mapped to an abstract state as defined by n values $OBR = \{obr_1, obr_2, ..., obr_n\}$, where each value obr_i represents the return value of method call ob_i invoked on cs.

DEFINITION 3. *Given a class c and a set of observer-method calls $OB = \{ob_1, ob_2, ..., ob_n\}$ of c, an* observer abstraction *with respect to OB is an OSM M of c such that the states in M are abstract states defined by OB.*

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

By default, each observer abstraction generated by our Abstra tool is defined based on the calls of a single observer method. When an observer method has different method calls (with different given arguments), we invoke these method calls of the observer method and use their return values to construct an observer abstraction.

4.2 Branch-Coverage Abstraction

The branch-coverage abstraction technique constructs an abstraction function by using branch coverage of methods invoked on concrete states. We first define the branch coverage we shall use in representing an abstract state of an object.

DEFINITION 4. 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.*

DEFINITION 5. *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 6. *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.

By default, each abstract OSM generated by our Abstra tool is defined based on all method calls whose methods have a non-empty conditional set.

4.3 State-Slicing Abstraction

The state-slicing abstraction technique constructs an abstraction function by using the values of specific member fields of concrete states.

DEFINITION 7. *Given an object o of class c and a set of c's member fields $MF = \{mf_1, mf_2, ..., mf_n\}$, the abstract state of o with respect to MF is represented by $\{fv_1, fv_2, ..., fv_n\}$, where fv_i is value of o's mf_i.*

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

Note that if the member field is of a reference type, the value of the member field include the values of all fields transitively reachable from the member field. But a member field of a reference type often points to a complex object graph; therefore, even if we use the value of the member field to represent an abstract state, we may still produce a complex abstract OSM. To address this issue, by default our Abstra tool conducts *structural abstraction* on a member field of a reference type by keeping only structural information among object fields but ignoring those primitive field values in a sliced state. The underlying rationale for structural abstraction is that object states sharing the same object graph structure often exhibit certain common behavior.

By default, each abstract OSM generated by our Abstra tool is defined based on a single member field.

5. DISCUSSION

The observer abstraction technique relies on the availability of (good) observer methods. The complexity of an abstract OSM with respect to observer methods depends on the characteristics of its corresponding observer methods. Observer abstractions help investigate behavior related to the return values of observers. The effectiveness of the state-slicing abstraction technique depends on the characteristics of member fields. The effectiveness of the branch-coverage abstraction technique depends on the characteristics of control flow graphs in method body.

In some cases, it might be difficult to construct a good abstraction function from the code with default configurations of these three techniques. Then human inputs can be used to improve the results. For example, developers can configure Abstra to use several observer methods for constructing an abstract OSM, rather than using a single observer method. Developers can write extra observer methods for the class under test in order to get better abstract OSMs. Developers can configure the branch-coverage abstraction to use only the branches in a specified subset of public methods or the branches that are related to specified object fields. Developers can specify the state-slicing abstraction to use a group of member fields rather than using a single observer method. Developers can construct indistinguishability properties [8] or other forms of abstraction functions to group values of specific member fields when applying the state-slicing abstraction technique.

We expect that this way of getting human inputs in Abstra 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 OSMs extracted without their upfront inputs (some OSMs could have already been useful for them to understand parts of program behavior). Second, we expect that it would be easier for programmers to formulate abstraction functions based on the crude OSMs extracted by Abstra.

6. TOOL IMPLEMENTATION

The Abstra tool consists of four components: state collector, test generator, OSM extractor, and OSM presenter.

State Collector. To collect concrete object states, Abstra uses Java reflection mechanisms [2] to recursively collect all the fields that are reachable from an object. Abstra also instruments test classes to collect method call information that is used to reproduce object states in test generation.

Test Generator. Recall that the observer abstraction and branch-coverage abstraction techniques require invoking specific methods on the concrete state to be abstracted. Because these method calls may not occur in the existing test suite, our test generator generates new tests to invoke methods on each concrete state exercised by the existing test suite. In particular, we perform combinatorial test generation on the two types of inputs: concrete states exercised by the existing test suite and method calls exercised by the existing

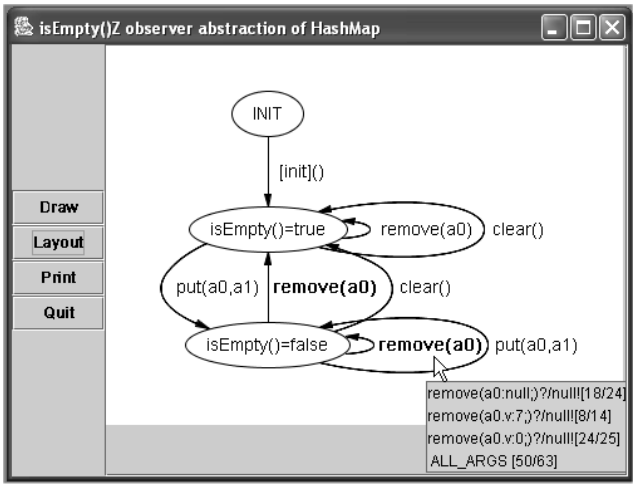

Figure 3: `isEmpty` observer abstraction of HashMap (screen snapshot)

test suite. Abstra uses Java reflection mechanisms [2] to generate and execute new tests online. Abstra exports generated tests to a JUnit [6] test class.

OSM Extractor. For observer abstraction, Abstra collects the return values of observer methods invoked on each concrete object state. For branch-coverage abstraction, Abstra collects covered branches of methods invoked on each concrete object state. For state-slicing abstraction, Abstra collects field values of each concrete object state. After collecting the information required to represent abstract states, Abstra construct transitions (method calls) among them to construct abstract OSMs.

OSM Presenter. Abstra displays extracted abstract OSMs by using the Grappa package, which is part of graphviz [7].

For example, Figure 3 shows a screen snapshot of an abstract OSM generated based on an observer method `isEmpty` of `HashMap`, which is an implementation of a map in the Java Collections Framework, being a part of the standard Java libraries [12]. We configured Abstra to display on each edge only the method name associated with the transition. When developers want to see the details of a transition, they can move the mouse cursor over the method name associated with the transition and then the details are displayed (detail notations are described in [15]).

7. 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] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[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. 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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/`.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 277–287, November 2005.

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

[19] H. Yuan and T. Xie. Automatic extraction of abstract-object-state machines based on branch coverage. In *Proc. 1st International Workshop on Reverse Engineering To Requirements at WCRE 2005 (RETR 2005)*, pages 5–11, November 2005.