Appears in *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 04)*, Seattle, WA

Automatic Extraction of Object-Oriented Observer Abstractions from Unit-Test Executions

Tao Xie and David Notkin

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

Abstract. Unit testing has become a common step in software development. Although manually created unit tests are valuable, they are often insufficient; therefore, programmers can use an automatic unit-test-generation tool to produce a large number of additional tests for a class. However, without a priori specifications, programmers cannot practically inspect the execution of each automatically generated test. In this paper, we develop the observer abstraction approach for automatically extracting object-state-transition information of a class from unit-test executions, without requiring a priori specifications. Given a class and a set of its initial tests generated by a third-party tool, we generate new tests to augment the initial tests and produce the abstract state of an object based on the return values of a set of observers (public methods with non-void returns) invoked on the object. From the executions of both the new and initial tests, we automatically extract observer abstractions, each of which is an object state machine (OSM): a state in the OSM represents an abstract state and a transition in the OSM represents method calls. We have implemented the Obstra tool for the approach and have applied the approach on complex data structures; our experiences suggest that this approach provides useful object-state-transition information for programmers to inspect unit-test executions effectively.

1 Introduction

Automatic test-generation tools are powerful; given a class, these tools can generate a large number of tests, including some valuable corner cases or special inputs that programmers often forget to include in their manual tests. When programmers write specifications, some specification-based test generation tools [4, 10, 18] automatically generate tests and check execution correctness against the written specifications. Without a prior specifications, some automatic test-generation tools [11] perform structural testing by generating tests to increase structural coverage. Some other tools [5] perform random testing by generating random inputs. Without a prior specifications, programmers rely on uncaught exceptions or inspect the executions of generated tests for determining whether the program behaves as expected. However, relying on only uncaught exceptions for catching bugs is limited and inspecting the executions of a large number of generated tests is impractical.

To help programmers to inspect unit-test executions effectively, we develop the *observer abstraction* approach, a novel black-box approach for summarizing and presenting object-state-transition information from unit-test executions. The approach is automated and does not require a priori specifications. Instead of inspecting the execution of each single test, programmers can inspect the summarized object-state-transition information for various purposes. For example, programmers can inspect the information to determine whether the class under test exhibits expected behavior. Programmers can also inspect the information to investigate causes of the failures exhibited by uncaught exceptions. Programmers can inspect the information for achieving better understanding of the class under test or even its tests.

A *concrete object state* of an object is characterized by the values of all the fields reachable from the object. An *observer* is a public method with a non-void return.[1] The observer abstraction approach abstracts a concrete object state exercised by a test suite based on the return values of a set of observers that are invoked on the concrete object state. An *observer abstraction* is an object state machine (OSM): a state in the OSM represents an abstract state and a transition in the OSM represents method calls. We have implemented a tool, called Obstra, for the observer abstraction approach. Given a Java class and its initial unit test, Obstra identifies concrete object states exercised by the tests and generates new tests to augment these initial tests. Based on the return values of a set of observers, Obstra maps each concrete object state to an abstract state and constructs an OSM.

This paper makes the following main contributions:

- We propose a new program abstraction, called observer abstraction.
- We present and implement an automatic approach for dynamically extracting observer abstractions from unit-test executions.
- We apply the approach on complex data structures and their automatically generated tests; our experiences show that extracted observer abstractions provide useful object-state-transition information for programmers to inspect.

2 Observer Abstraction Approach

We first discuss two techniques (developed in our previous work [22, 23]) that enable the dynamic extraction of observer abstractions. We next describe object state machines, being the representations of observer abstractions. We then define observer abstractions and illustrate dynamic extraction of them. We finally describe the implementation and present an example of dynamically extracted observer abstractions.

2.1 Concrete-State Representation and Test Augmentation

In previous work, we have developed the Rostra framework and five automatic techniques to represent and detect equivalent object states [23]. This work focuses on using

[1] We follow the definition by Henkel and Diwan [12]. The definition differs from the more common definition that limits an observer to methods that do not change any state. We have found that state-modifying observers also provide value in our technique and state modification does not harm our technique.

one of the techniques for state representation: the WholeState technique. The technique represents a *concrete object state* of an object as a sequence of the values of the fields reachable from the object. We use a linearization algorithm [23] to avoid putting those field values with reference addresses in the sequence[2] but still keep the referencing relationship among fields. A set of *nonequivalent concrete object states* contain those concrete object states any two of which do not have the same state representation.

A unit test suite consists of a set of unit tests. Each execution of a unit test creates several objects and invokes methods on these objects. Behavior of a method invocation depends on the state of the receiver object and method arguments at the beginning of the invocation. A *method call* is characterized by the actual class of the receiver object, the method name, the method signature, and the method-argument values. When argument values are not primitive values, we represent them using their state representations. We determine whether two method calls are equivalent by checking the equivalence of their corresponding characteristic entities, including the receiver-object class, method name, method signature, and method-argument values. A set of *nonequivalent method calls* contain those method calls any two of which are not equivalent.

After we execute an initial test suite, the concrete-state representation technique identifies all nonequivalent object states and nonequivalent method calls that were exercised by the test suite. The test augmentation technique generates new tests to exercise each possible combination of nonequivalent object states and nonequivalent non-constructor method calls [22]. We augment an initial test suite because the test suite might not invoke each observer on all nonequivalent object states; invoking observers on a concrete object state is necessary for us to know the abstract state enclosing the concrete object state. The augmented test suite guarantees that each nonequivalent object state is exercised by each nonequivalent non-constructor method call at least once. In addition, the observer abstractions extracted from the augmented test suite can better help programmers to inspect object-state-transition behavior. The complexity of the test augmentation algorithm is $O(|CS| \times |MC|)$, where CS is the set of the nonequivalent concrete states exercised by an initial test suite T for the class under test and MC is the set of the nonequivalent method calls exercised by T.

2.2 Object State Machine

We define an object state machine for a class:[3]

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*

[2] Running the same test twice might produce different reference addresses for those fields of non-primitive types.

[3] The definition is adapted from the definition of finite state machine [15]; however, an object state machine is not necessarily finite.

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 $\lambda(s, i)$.

An OSM can be deterministic or nondeterministic.

2.3 Observer Abstractions

The object states in an OSM can be concrete or abstract. An *abstract state* of an object is defined by an *abstraction function* [16]; the abstraction function maps each concrete state to an abstract state. The observer abstraction approach constructs abstraction functions to map concrete states to abstract states in an OSM.

We first define an observer following previous work on specifying algebraic specifications for a class [12]:

Definition 2. *An* observer *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 observers $OB = \{ob_1, ob_2, ..., ob_n\}$ of c, the observer abstraction approach 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 observers $OB = \{ob_1, ob_2, ..., ob_n\}$ 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.*

2.4 Dynamic Extraction of Observer Abstractions

We dynamically extract observer abstractions of a class from unit-test executions. The number of the concrete states exercised by an augmented test suite is finite and the execution of the test suite is assumed to terminate; therefore, the dynamically extracted observer abstractions are also finite.

In a dynamically extracted observer abstraction M, we add additional statistical information: the transition count and the emission count. The *transition count* for a nonequivalent method call mc transiting from a state s to a state s' is the number of nonequivalent concrete object states in s that transit to s' after mc is invoked. The *emission count* for s and mc is the number of nonequivalent concrete object states in s where mc is invoked.

Given an initial test suite T for a class c, we first identify the nonequivalent concrete states CS and method calls MC exercised by T. We then augment T with new tests to exercise CS with MC exhaustively, producing an augmented test suite T'. We have described these steps in Section 2.1. T' exercises each nonequivalent concrete state in CS with each method call in MC; therefore, each nonequivalent observer call in MC is guaranteed to be invoked on each nonequivalent concrete state in CS at least once. We then collect the return values of observer calls in MC for each nonequivalent concrete state in CS. We use this test-generation mechanism to collect return values of

observers, instead of inserting observer method calls before and after any call site to c in T, because the latter does not work for state-modifying observers, which change the functional behavior of T.

Given an augmented test suite T' and a set of observers $OB = \{ob_1, ob_2, ..., ob_n\}$, we go through the following steps to produce an observer abstraction M in the form of OSM. Initially M is empty. During the execution of T', we collect the following tuple for each method execution in c's interface: (cs_s, m, mr, cs_e), where cs_s, m, mr, and cs_e are the concrete object state at the method entry, method call, return value, and concrete object state at the method exit, respectively. If m's return type is void, we assign "$-$" to mr. If m's execution throws an uncaught exception, we also assign "$-$" to mr and assign the name of the exception type to cs_e, called an *exception state*. The concrete object state at a constructor's entry is $INIT$, called an *initial state*.

After the test executions terminate, we iterate on each distinct tuple (cs_s, m, mr, cs_e) to produce a new tuple (as_s, m, mr, as_e), where as_s and as_e are the abstract states mapped from cs_s and cs_e based on OB, respectively. If cs_e is an exception state, its mapped abstract state is the same as cs_e, whose value is the name of the thrown-exception type. If cs_s is an initial state, its mapped abstract state is still $INIT$. If cs_e is not exercised by the initial tests before test augmentation but exercised by new tests, we map cs_e to a special abstract state denoted as N/A, because we have not invoked OB on cs_e yet and do not have a known abstract state for cs_e.

After we produce (as_s, m, mr, as_e) from (cs_s, m, mr, cs_e), we then add as_s and as_e to M as states, and put a transition from as_s to as_e in M. The transition is denoted by a triple $(as_s, m?/mr!, as_e)$. If as_s, as_e, or $(as_s, m?/mr!, as_e)$ is already present in M, we do not add it. In addition, we increase the transition count for $(as_s, m?/mr!, as_e)$, denoted as $C_{(as_s, m?/mr!, as_e)}$, which is initialized to one when $(as_s, m?/mr!, as_e)$ is added to M at the first time. We also increase the emission count for as_s and m, denoted as $C_{(as_s, m)}$. After we finish processing all distinct tuples (cs_s, m, mr, cs_e), we postfix the label of each transition $(as_s, m?/mr!, as_e)$ with $[C_{(as_s, m?/mr!, as_e)}/C_{(as_s, m)}]$. The complexity of the extraction algorithm for an observer abstraction is $O(|CS| \times |OB|)$, where CS is the set of the nonequivalent concrete states exercised by an initial test suite T and OB is the given set of observers.

2.5 Implementation

We have developed a tool, called Obstra, for the observer abstraction approach. Obstra is implemented based on the Rostra framework developed in our previous work [23]. We use the Byte Code Engineering Library (BCEL) [6] to rewrite the bytecodes of a class at class-loading time. Objects of the class under test are referred as *candidate objects*. We collect concrete object states at the entry and exit of each method call invoked from a test suite to a candidate object; these method calls are referred as *candidate method calls*. We do not collect object states for those method calls that are internal to candidate objects.

To collect concrete object states, we use Java reflection mechanisms [2] to recursively collect all the fields that are reachable from a candidate object, an argument object, or a return object of candidate method calls. We also instrument test classes to

collect method call information that is used to reproduce object states in test augmentation. We also use Java reflection mechanisms [2] to generate and execute new tests online. We export a selected subset of tests in the augmented test suite to a JUnit [13] test class using JCrasher's functionality of test-code generation [5]; we select and export a test if it exercises at least one previously uncovered transition in an observer abstraction. Each exported test is annotated with its exercised transitions as comments. We display extracted observer abstractions by using the Grappa package, which is part of graphviz [8].

By default, Obstra generates one OSM for each observer (in addition to one OSM for all observers) and outputs a default grouping configuration file; programmers can manipulate the configurations in the file to generate OSM's based on multiple observers.

2.6 Example

We use a class of Binary Search Tree (named as `BSTree`) as an example of illustrating observer abstractions. This class was used in evaluating Korat [4] and the Rostra framework in our previous work [23]. Parasoft Jtest (a commercial tool for Java) [18] generates 277 tests for the class. The class has 246 non-comment, non-blank lines of code and its interface includes eight public methods (five observers), some of which are a constructor (denoted as `[init]()`), `boolean contains(MyInput info)`, `void add(MyInput info)`, and `boolean remove (MyInput info)` where `MyInput`[4] is a class that contains an integer field `v`.

Figure 1 shows the observer abstraction of BSTree with respect to an observer `contains(MyInput info)` (including two observer instances: `add(a0.v:7;)`[5] and `add(a0:null;)`) and augmented Jtest-generated tests. The top state in the figure is marked with `INIT`, indicating the object state before invoking a constructor. The second-to-top state is marked with the observer instances and their `false` return values. This abstract state encloses those concrete states such that when we invoke these two observer instances on those concrete states, their return values are `false`. In the central state, the observers throw uncaught exceptions and we put the exception-type name `NullPointerException` in the positions of their return values. The bottom state is marked with the exception-type name `NullPointerException`. An object is in such a state after a method call on the object throws the `NullPointerException`.

Each transition from a starting abstract state to an ending abstract state is marked with method calls, their return values, and some statistics. For example, the generated test suite contains two tests:

```
Test 1 (T1):                        Test 2 (T2):
   BSTree b1 = new BSTree();           BSTree b1 = new BSTree();
   MyInput m1 = new MyInput(0);        b1.remove(null);
   b1.add(m1);
   b1.remove(null);
```

[4] The original argument type is `Object`; we change the type to `MyInput` so that Jtest can be guided to generate better arguments.

[5] `a`i represents the $(i+1)$th argument and `a`i`.v` represents the `v` field of the $(i+1)$th argument. Argument values are specified following their argument names separated by `:` and different arguments are separated by `;`.

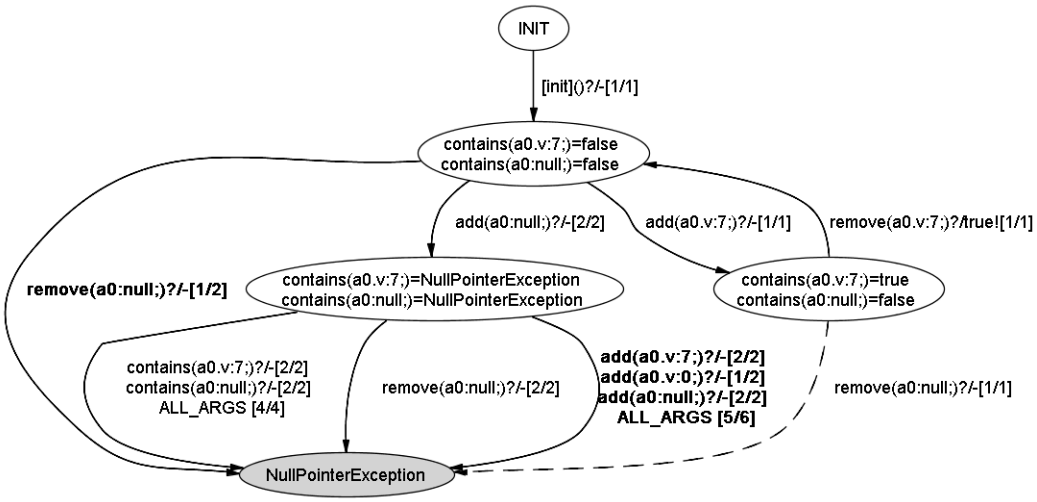

Fig. 1. `contains` observer abstraction of BSTree

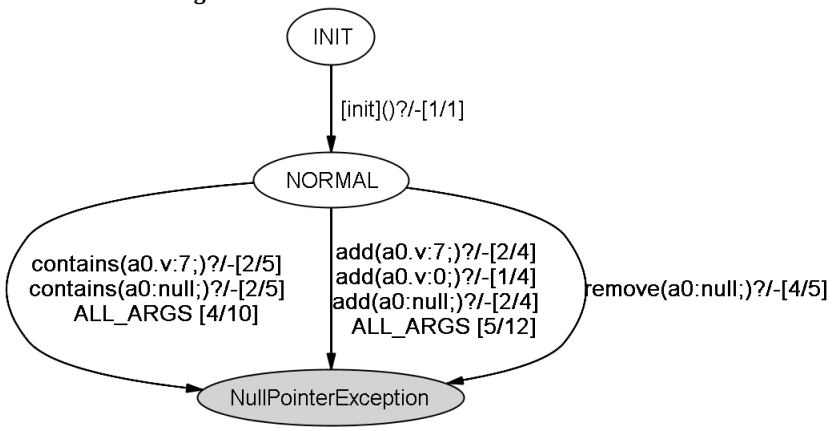

Fig. 2. `exception` observer abstraction of BSTree

The execution of `b1.remove(null)` in T1 does not throw any exception. Both before and after invoking `b1.remove(null)` in T1, if we invoke the two observer instances, their return values are `false`; therefore, there is a state-preserving transition on the second-to-top state. (To present a succinct view, by default we do not show state-preserving transitions.) The execution of `b1.remove(null)` in T2 throws a `NullPointerException`. If we invoke the two observer instances before invoking `b1.remove(null)` in T2, their return values are `false`; therefore, given the method execution of `b1.remove(null)` in T2, we extract the transition from the second-to-top state to the bottom state and the transition is marked with `remove(a0:null;)?/-`. In the mark of a transition, when return values are `void` or method calls throw uncaught exceptions, we put `-` in the position of their return values. We put `?` after the

method calls and `!` after return values if return values are not `-`. We also attach two numbers for each transition in the form of `[N/M]`, where `N` is the transition count and `M` is the emission count. If these two numbers are equal, the transition is deterministic, and is indeterministic otherwise. Because there are two different transitions from the second-to-top state with the same method call `remove(a0:null;)` (one transition is state-preserving being extracted from T1), the transition `remove(a0:null;)` from the second-to-top state to the bottom state is indeterministic, being attached with `[1/2]`. We display thicker edges and bold-font texts for nondeterministic transitions so that programmers can easily identify them based on visual effect.

To present a succinct view, we do not display N/A states and the transitions leading to N/A states. In addition, we combine multiple transitions that have the same starting and ending abstract states, and whose method calls have the same method names and signatures. When we combine multiple transitions, we calculate the transition count and emission count of the combined transitions and show them in the bottom line of the transition label. When a combined transition contains all nonequivalent method calls of the same method name and signature, we add ALL_ARGS in the bottom line of the transition label. When a transition contains only method calls that are exercised by new generated tests but not exercised by the initial tests, we display a dotted edge for the transition.

To focus on understanding uncaught exceptions, we create a special *exception observer* and construct an observer abstraction based on it. Figure 2 shows the exception-observer abstraction of BSTree extracted from augmented Jtest-generated tests. The exception observer maps all concrete states except for $INIT$ and exception states to an abstract state called `NORMAL`. The mapped abstract state of an initial state is still $INIT$ and the mapped abstract state of an exception state is still the same as the exception-type name.

3 Experiences

We have used Obstra to extract observer abstractions from a variety of programs, most of which were used to evaluate our previous work in test selection [25], test minimization [23], and test generation [22]. Many of these programs manipulate nontrivial data structures. Because of the space limit, in this section, we illustrate how we applied Obstra on two complex data structures and their automatically generated tests. We applied Obstra on these examples on a MS Windows machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 Mb allocated memory.

3.1 Binary Search Tree Example

We have described the BSTree in Section 2.6 and two of its extracted observer abstractions in Figure 1 and 2. Jtest generates 277 tests for BSTree. These tests exercise five nonequivalent concrete object states in addition to the initial state and one exception state, 12 nonequivalent non-constructor method calls in addition to one constructor call, and 33 nonequivalent method executions. Obstra augments the test suite to exercise 61 nonequivalent method executions. The elapsed real time for test augmentation and abstraction extraction is 0.4 and 4.9 seconds, respectively.

Figure 2 shows that `NullPointerException` is thrown by three nondeterministic transitions. During test inspection, we want to know in what conditions the exception is thrown. If the exception is thrown because of illegal inputs, we can add necessary preconditions to guard against the illegal inputs. Alternatively, we can perform defensive programming: we can add input checking at method entries and throw more informative exceptions if the checking fails. However, we do not want to add over-constrained preconditions, which prevent legal inputs from being processed. For example, after inspecting the exception OSM in Figure 2, we should not consider all arguments for `add`, the `null` argument for `remove`, or all arguments for `contains` as illegal arguments, although doing so indeed prevents the exceptions from being thrown. After we inspected the `contains` OSM in Figure 1, we gained more information about the exceptions and found that calling `add(a0:null;)` after calling the constructor leads to an undesirable state: calling `contains` on this state deterministically throws the exception. In addition, calling `remove(a0:null;)` also deterministically throws the exception and calling `add` throws the exception with a high probability of 5/6. Therefore, we had more confidence on considering `null` as an illegal argument for `add` and preventing it from being processed. After we prevented `add(a0:null;)`, two `remove(a0:null;)` transitions still throw the exception: one is deterministic and the other is with 1/2 probability. We then considered `null` as an illegal argument for `remove` and prevented it from being processed. We did not need to impose any restriction on the argument of `contains`.

We found that there are three different arguments for `add` but only two different arguments for `contains`, although these two methods have the same signatures. We could add a method call of `contain(a0.v:0;)` to the Jtest-generated test suite; therefore, we could have three observer instances for the `contains` OSM in Figure 1. In the new OSM, the second-to-top state includes one more observer instance `contains(a0.v:0)=false` and the indeterministic transition of `remove(a0:null;)?/-[1/2]` from the second-to-top state to the bottom state is turned into a deterministic transition `remove(a0:null;)?/-[1/1]`. In general, when we add new tests to a test suite and these new tests exercise new observer instances in an OSM, the states in the OSM can be refined, thus possibly turning some indeterministic transitions into deterministic ones. On the other hand, adding new tests can possibly turn some deterministic transitions into indeterministic ones.

3.2 Hash Map Example

A HashMap class was given in `java.util.HashMap` from the standard Java libraries [19]. A `repOK` and some helper methods were added to this class for evaluating Korat [4]. We also used this class in our previous work for evaluating Rostra [23]. The class has 597 non-comment, non-blank lines of code and its interface includes 19 public methods (13 observers), some of which are `[init](), void setLoadFactor(float f), void putAll(Map t), Object remove(MyInput key), Object put(MyInput key, MyInput value)`, and `void clear()`. Jtest generates 5186 tests for HashMap. These tests exercise 58 nonequivalent concrete object states in addition to the initial state and one exception state, 29 nonequivalent non-constructor method calls in addition to one constructor call, and 416 nonequivalent method executions. Obstra augments

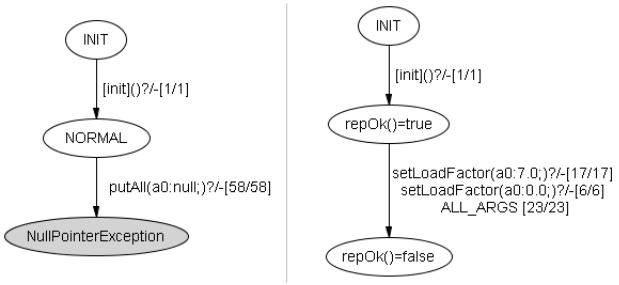

Fig. 3. `exception` observer abstraction and `repOk` observer abstraction of HashMap

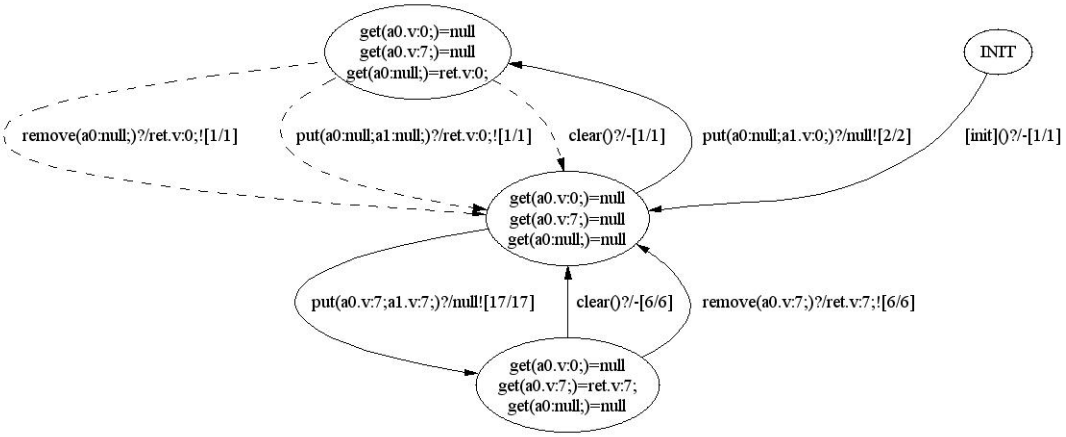

Fig. 4. `get` observer abstraction of HashMap

the test suite to exercise 1683 nonequivalent method executions. The elapsed real time for test augmentation and abstraction extraction is 10 and 15 seconds, respectively.

We found that the exception OSM of HashMap contains one deterministic transition `putAll(a0:null;)` from `NORMAL` to `NullPointerException`, as is shown in the left part of Figure 3. Therefore, we considered `null` as an illegal argument for `putAll`. We checked the Java API documentation for HashMap [19] and the documentation states that `putAll` throws `NullPointerException` if the specified map is null. This description confirmed our judgement. In other observer abstractions, to provide a more succinct view, by default Obstra does not display any deterministic transitions leading to an exception state in the exception OSM.

We found an error in `setLoadFactor(float f)`, a method that was later added to facilitate Korat's test generation [4]. The right part of Figure 3 shows the `repOk` OSM of HashMap. `repOk` is a predicate used to check class invariants [16]. If calling `repOk` on an object state returns `false`, the object state is invalid. By inspecting the `repOK` OSM, we found that calling `setLoadFactor` with all arguments deterministically leads to an invalid state. We checked the source code of `setLoadFactor` and found that its method body is simply `loadFactor = f;`, where `loadFactor` is an object field and `f` is the method argument. The comments for a private field `threshold` states that the value of

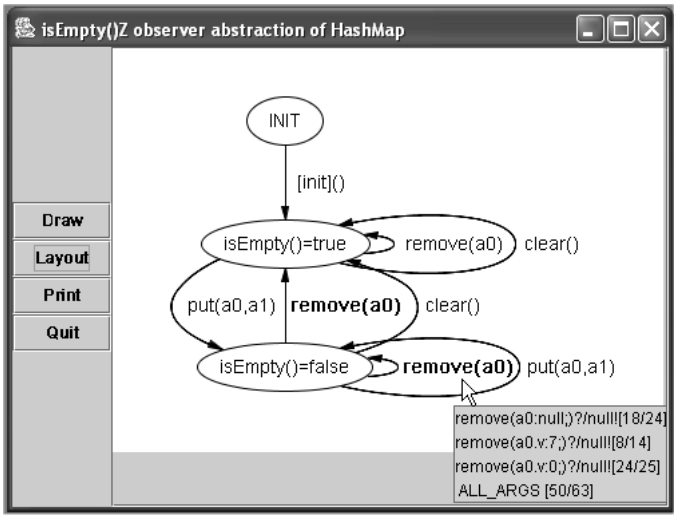

Fig. 5. `isEmpty` observer abstraction of HashMap (screen snapshot)

`threshold` shall be `(int)(capacity * loadFactor)`. Apparently this property is violated when setting `loadFactor` without updating `threshold` accordingly. We fixed this error by appending a call to an existing private method `void rehash()` in the end of `setLoadFactor`'s method body; `rehash` updates `threshold` using the new `loadFactor`.

Figure 4 shows the `get` OSM of HashMap. In the representation of method returns on a transition or in a state, `ret` represents the non-primitive return value and `ret.v` represents the `v` field of the non-primitive return value. Recall that a transition with a dotted edge is exercised only by new generated tests but not by the initial tests. We next walk through the scenario in which programmers could inspect Figure 4. During inspection, programmers might focus their exploration of an OSM on transitions. Three such transitions are `clear`, `remove`, and `put`. Programmers are not surprised to see that `clear` or `remove` transitions cause a nonempty HashMap to be empty, as is shown by the transitions from the top or bottom state to the central state. But programmers are surprised to see the transition of `put(a0:null;a1:null)` from the top state to the central state, indicating that `put` can cause a nonempty HashMap to be empty. By browsing the Java API documentation for HashMap [19], programmers can find that HashMap allows either a key or a value to be `null`; therefore, the `null` return of `get` does not necessarily indicate that the map contains no mapping for the key. However, in the documentation, the description for the returns of `get` states: "the value to which this map maps the specified key, or null if the map contains no mapping for this key." After reading the documentation more carefully, they can find that the description for `get` (but not the description for the returns of `get`) does specify the accurate behavior. This finding shows that the informal description for the returns of `get` is not accurate or consistent with the description of `get` even in such widely published Java API documentation [19].

Figure 5 shows a screen snapshot of the `isEmpty` OSM of HashMap. We configured Obstra to additionally show each state-preserving transition that has the same method name as another state-modifying transition. We also configured Obstra to display on each edge only the method name associated with the transition. When programmers 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. We have searched the Internet for manually created state machines for common data structures but few could be found. One manually created state machine for a container structure [17] is almost the same as the `isEmpty` OSM of HashMap shown in Figure 5. There are two major differences. The $INIT$ state and the `[init]()` transition are shown in Figure 5 but not in the manually created state machine. The manually created state machine annotates "not last element" for the state-preserving transition `remove(a0)` (pointed by the mouse cursor in Figure 5) on the `isEmpty()=false` state and "last element" for the state-modifying transition `remove(a0)` (shown in the middle of Figure 5) starting from the `isEmpty()=false` state; Figure 5 shows these two transition names in bold font, indicating them to be indeterministic.

3.3 Discussion

Our experiences have shown that extracted observer abstractions can help investigate causes of uncaught exceptions, identify weakness of an initial test suite, find bugs in a class implementation or its documentation, and understand class behavior. Although many observer abstractions extracted for the class under test are succinct, some observer abstractions are still complex, containing too much information for inspection. For example, three observers of HashMap, such as `Collection values()`, have 43 abstract states. The complexity of an extracted observer abstraction depends on both the characteristics of its observers and the initial tests. In future work, we plan to display a portion of a complex observer abstraction based on user-specified filtering criteria or extract observer abstractions from the executions of a user-specified subset of the initial tests.

Although the `isEmpty` OSM of HashMap is almost the same as a manually created state machine [17], our approach does not guarantee the completeness of the resulting observer abstractions — our approach does not guarantee that the observer abstractions contain all possible legal states or legal transitions. Our approach also does not guarantee that the observer abstractions contain no illegal transitions. Instead, the observer abstractions faithfully reflect behavior exercised by the executed tests; inspecting observer abstractions could help identify weakness of the executed tests. This characteristic of our approach is shared by other dynamic inference techniques [1, 7, 12, 21] discussed in the next section.

4 Related Work

Ernst et al. use Daikon to dynamically infer likely invariants from test executions [7]. 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 observer abstractions describe the observed state-transition relationships among multiple methods in a class interface and use the return values of observers to represent object states, without explicitly referring to object fields. Henkel and Diwan discover algebraic specifications from the execution of automatically generated unit tests [12]. Their discovered algebraic specifications present a local view of relationships between two methods, whereas observer abstractions 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 observer abstractions; our experiences show that these indeterministic transitions are useful for inspection. In summary, observer abstractions are a useful form of property inference, complementing invariants or algebraic specifications inferred from unit-test executions.

Whaley et al. extract Java component interfaces from system-test executions [21]. The extracted interfaces are in the form of multiple finite state machines, each of which contains the methods that modify or read the same object field. Observer abstractions are also in the form of multiple finite state machines, each of which is with respect to a set of observers (containing one observer by default). Whaley et al. map all concrete states that are at the same state-modifying method's exits to the same abstract state. Our approach maps all concrete states on which observers' return values are the same to the same abstract state. Although Whaley et al.'s approach is applicable on system-test executions, it is not applicable on the executions of automatically generated unit tests, because their resulting finite state machine would be a complete graph of methods that modify the same object field. Ammons et al. mine protocol specifications in the form of a finite state machine from system-test executions [1]. Their approach faces the same problem as Whaley et al.'s approach when being applied on the executions of automatically generated unit tests. In summary, neither Whaley et al. nor Ammons et al.'s approaches capture object states as accurate as our approach and neither of them can be applied on the executions of automatically generated unit tests.

Given a set of predicates, predicate abstraction [3, 9] maps a concrete state to an abstract state that is defined by the boolean values of these predicates on the concrete state. Given a set of observers, observer abstraction maps a concrete state to an abstract state that is defined by the return values (not limited to boolean values) of these observers on the concrete state. Concrete states considered by predicate abstractions are usually those program states between program statements, whereas concrete states considered by observer abstractions are those object states between method calls. Predicate abstraction is mainly used in software model checking, whereas observer abstraction in our approach is mainly used in helping inspection of test executions.

Turner and Robson use finite state machines to specify the behavior of a class [20]. 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. Their approach specifies specifications in the form of finite state machines and generates tests based on the specifications, whereas our approach extracts observer abstractions in the form of finite state machines without requiring a priori specifications. In future work, we plan to use extracted observer abstractions to guide test generation using existing finite-state-machine-based testing techniques [15] and use new generated tests to further improve observer abstractions.

This future work fits into the feedback loop between test generation and specification inference proposed in our previous work [24].

Kung et al. statically extract object state models from class source code and use them to guide test generation [14]. 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 finite state machines based on observers during test executions. Grieskamp et al. generate finite state machines from executable abstract state machines [10]. Manually specified predicates are used to group states in abstract state machines to hyperstates during the execution of abstract state machine. Finite state machines, abstract state machines, and manually specified predicates in their approach correspond to observer abstractions, concrete object state machines, and observers in our approach, respectively. However, our approach is totally automatic and does not require programmers to specify any specifications or predicates.

5 Conclusion

It is important to provide tool support for programmers as they inspect the executions of automatically generated unit tests. We have proposed the observer abstraction approach to aid inspection of test executions. We have developed a tool, called Obstra, to extract observer abstractions from unit-test executions automatically. We have applied the approach on a variety of programs, including complex data structures; our experiences show that extracted observer abstractions provide useful object-state-transition information for programmers to inspect.

Acknowledgments

We thank Arnaud Gotlieb, Amir Michail, Andrew Peterson, Vibha Sazawal and the anonymous reviewers for their valuable feedback on an earlier version of this paper. We thank Darko Marinov for providing Korat subjects. 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.

References

1. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. the 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. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. the International Symposium on Software Testing and Analysis*, pages 123–133, 2002.
5. C. Csallner and Y. Smaragdakis. JCrasher documents. Online manual, December 2003.
6. M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.
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. 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.
9. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. the 9th International Conference on Computer Aided Verification*, pages 72–83, 1997.
10. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. the International Symposium on Software Testing and Analysis*, pages 112–122, 2002.
11. N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proc. the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–244, 1998.
12. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. the 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.
13. JUnit, 2003. `http://www.junit.org`.
14. D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proc. the 18th Annual International Computer Software and Applications Conference*, pages 222–227, 1994.
15. 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.
16. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
17. D. Nguyen. Design patterns for data structures. In *Proc. the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 336–340, 1998.
18. Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.
19. 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/`.
20. C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proc. the Conference on Software Maintenance*, pages 302–310, 1993.
21. 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.
22. 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, January 2004.
23. 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*, 2004.
24. 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.
25. 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.