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

Tao Xie and David Notkin

Department of Computer Science & Engineering
University of Washington
WA 98195, USA
{taoxie,notkin}@cs.washington.edu
**Technical Report UW-CSE-04-05-05**
**May 2004**

**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 inspect the execution of each automatically generated test practically, cannot effectively isolate faults, or cannot easily understand the characteristics of either the tests or the component under test (such as a Java class). In this paper, we develop the observer abstraction approach for automatically extracting observer abstractions of a class from unit-test executions, without requiring *a priori* specifications. Programmers can inspect succinct observer abstractions for correctness checking, fault isolation, test characterization, and component understanding.

Given a class and a set of its manually or automatically generated tests, we identify nonequivalent concrete object states exercised by the tests and generate new tests to augment these tests. We map each nonequivalent concrete object state to an abstract state based on the return values of a set of observers (public methods with non-void returns) invoked on the object state. 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 a method call. We have implemented a tool, called Obstra, for the approach and have applied the approach on complex data structures; our experiences suggest that this approach provides succinct and useful information for programmers to inspect unit-test executions.

## 1 Introduction

Automatic test-generation tools are powerful; given a class, these tools can generate a large number of tests, including some valuable corner or special inputs that programmers often forget to include in their manual tests. When programmers write specifications, some specification-based test generation tools [1–3] automatically generate tests and check execution correctness against the written specifications. Without *a prior* specifications, some automatic test-generation tools [4] 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 execution of generated tests for checking correctness. However, relying on only uncaught exceptions for catching bugs is limited and inspecting the execution of a large number of generated tests is impractical. This problem is also known as the *test oracle* problem.

When manually or automatically generated tests throw uncaught exceptions, programmers still need to understand the causes of the failures exhibited by the exceptions. This problem is also known as *fault isolation*. Programmers want to investigate the fault-exposing conditions — once a test satisfies these conditions, it will expose the fault in high probability.

Programmers need to understand the characteristics of a manually or automatically generated unit-test suite. This problem is called *test characterization*. After understanding the weakness of a test suite, programmers can enhance the test suite by giving guidance to test generation tools or creating more manual tests. Residual structural coverage is one way of showing the weakness of a test suite [6]. If programmers write down specifications, specification coverage can also help programmers to understand the weakness of a test suite.

When programmers intend to reuse a third-party component (such as a Java class) whose implementation is not written by them or even is not accessible to them, they usually read the accompanying formal specifications or informal documentation for the component to understand how to use the component; however, formal specifications are often unavailable and informal documentation is often incomplete or inconsistent with the implementation. This problem is called *component understanding*.

In this paper, we develop the *observer abstraction* approach, a novel black-box approach for summarizing and presenting the dynamic information from unit-test executions. The approach is totally automatic without requiring *a priori* specifications. We use the approach to tackle the preceding four problems in unit testing and component reuse: correctness checking, fault isolation, test characterization, and component understanding.

A *concrete object state* is characterized by the values of all the fields of an 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 a method call. We have implemented a tool, called Obstra, for the observer abstraction approach. Given a Java class and a set of unit tests for it, Obstra identifies nonequivalent concrete object states exercised by the tests and generates new tests to augment these tests. Based on the return values of a set of observers, Obstra maps each nonequivalent concrete object state to an abstract state and constructs an OSM. Programmers can inspect these OSM's for addressing the preceding four problems. We have applied the approach on complex data structures and

[1] We follow the definition by Henkel and Diwan [7]. The definition differs from the more common definition, which limits an observer to methods that do not change any state. We have found that state-modifying observers also provide value in our approach.

their automatically generated tests; our experiences suggest that this approach provides succinct and useful information for inspecting test executions.

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 nontrivial data structures to tackle correctness checking, fault isolation, test characterization, and component understanding; our experiences show that extracted observer abstractions provide succinct and useful information for programmers to inspect.

## 2 Observer Abstraction Approach

We first discuss two techniques (developed in our previous work [8, 9]) 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

The technique of concrete state representation identifies nonequivalent object states [8]. Based on the nonequivalent object states, the test augmentation technique improves an existing test suite by generating new tests to exercise the nonequivalent object states exhaustively [9]. We augment an existing 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. In addition, the observer abstractions extracted from the augmented test suite can better help programmers to inspect dynamic behavior.

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. A *method execution* is characterized by the method call and a state representation of the receiver object at the beginning of the execution (called *method-entry state*). The method execution produces a return value[2] and a state representation of the receiver object at the end of the execution (called *method-exit state*). When argument values or return values are not primitive values, we represent them using state representations.

In previous work, we have developed the Rostra framework and five automatic techniques to represent and detect equivalent object states [8]. This work focuses on using one of the techniques for state representation: the WholeState technique. The technique models a concrete object state as a graph: nodes represent objects and edges represent

[2] A return value can be void.

object fields. Let $P$ be the set consisting of all primitive values, including `null`, integers, booleans, etc. Let $O$ be a set of objects whose fields form a set $F$. (Array elements are considered as object fields labelled with indices.)

**Definition 1.** *A* heap *is an edge-labelled graph* $\langle O, E\rangle$*, where* $E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}$.

**Definition 2.** *The* concrete state *of an object* $r$ *is a rooted heap* $\langle r, h\rangle$*, where* $r$ *is a root object and* $h$ *is a heap such that all nodes in* $h$ *are reachable from* $r$.

We transform a concrete object state into a sequence using a linearization algorithm [8], which has been applied in model checking for encoding states [10–12]. We then determine whether two concrete object states are equivalent by checking sequence equality. 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. We determine whether two method executions are equivalent by checking the equivalence of their corresponding method calls and method-entry states, respectively.

After we execute an existing test suite, the concrete state representation technique identifies a set of nonequivalent object states and nonequivalent method calls that 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 [9]. The test suite augmented with these new tests guarantees that each nonequivalent object state is exercised by each nonequivalent non-constructor method call at least once. The complexity of the test augmentation algorithm is $O(|CS|*|MC|)$, where $CS$ is the set of the nonequivalent concrete states exercised by $T$ 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 3.** *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* $c$*.* $\delta : S \times I \rightarrow S^*$ *is the state transition function and* $\lambda : S \times I \rightarrow O^*$ *is the output function. When the machine is in a current state* $s$ *and receives a method call* $a$ *from* $I$*, it moves to one of the next states specified by* $\delta(s, a)$ *and produces one of the method returns given by* $\lambda(s, a)$.

An OSM can be deterministic or nondeterministic.

[3] The definition is adapted from the definition of finite state machine [13]; however, an object state machine is not necessarily finite.

### 2.3 Observer Abstractions

The object states in an OSM can be concrete or abstract. We have defined the concrete state of an object in Section 2.1 (Definition 2). An *abstract state* of an object is defined by an *abstraction function* [14]; 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 [7]:

**Definition 4.** *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 5.** *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 for each nonequivalent method call $mc$ from a state $s$ to a state $s'$, the emission count for each combination of a state $s$ and a nonequivalent method call $mc$ transited from $s$, and the *abstraction ratio* for $M$. The *transition count* for $mc$ transiting from $s$ to $s'$ 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. The *abstraction ratio* for $M$ characterizes the abstraction capability of $OB$ that is used to abstract concrete states.

**Definition 6.** *Given a set of observers $OB$ and a test suite $T$ of a class $c$, the* abstraction ratio *of an observer abstraction $M$ with respect to $OB$ and $T$ is* 1 - $|AS|/|CS|$, *where $AS$ is the set of the abstract states in $M$ and $CS$ is the set of the nonequivalent concrete states exercised by $T$.*

Given a 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 these observer calls 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 method-entry state, method call, return value, and concrete method-exit state, 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 method-entry state of a constructor 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 tests before test augmentation but exercised by the 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 $cs_e$. By default, an OSM does not display $N/A$ states and the transitions leading to $N/A$ states.

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 $O(|CS| * |OB|)$, where $CS$ is the set of the nonequivalent concrete states exercised by $T$ 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 [8]. We use the Byte Code Engineering Library (BCEL) [15] 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 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 [16] 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 [16] to generate and execute new tests online. We export a selected subset of tests in the augmented test suite to a JUnit [17] 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 dot program, which is part of graphviz [18].

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 [2] and the Rostra framework in our previous work [8]. Parasoft Jtest (a commercial tool for Java) [3] 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)`, 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(arg0.v:7;)`[5] and `add(arg0: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 rightmost 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 throwing 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:

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

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

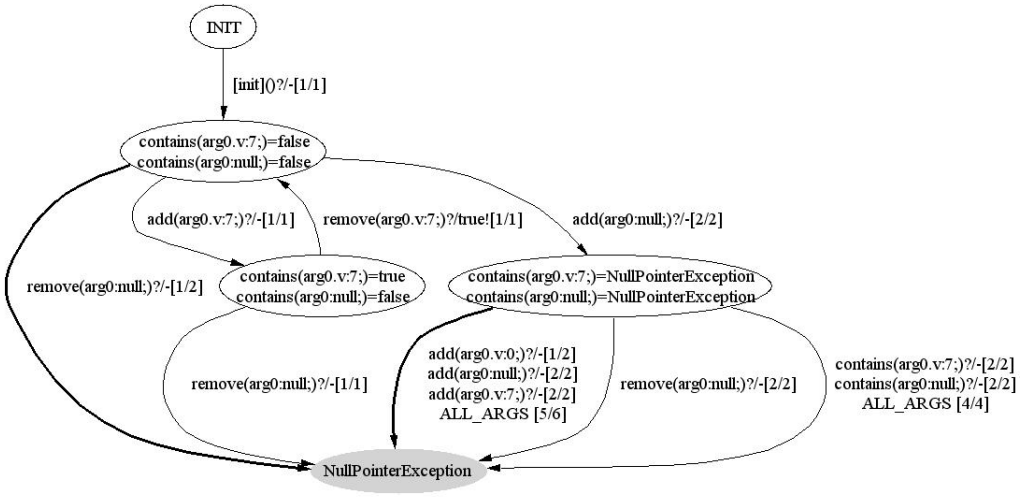


**Fig. 1.** `contains` observer abstraction of BSTree

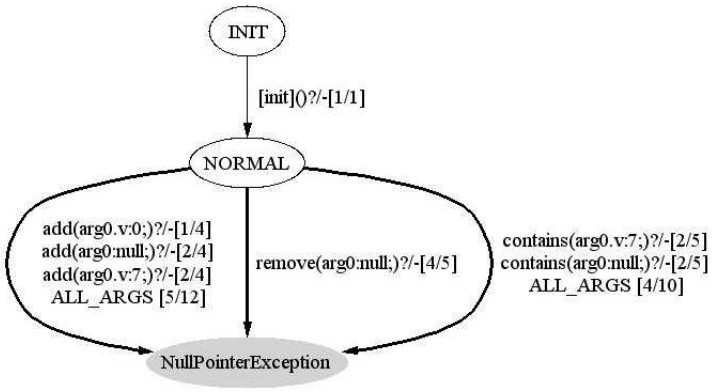


**Fig. 2.** `exception` observer abstraction of BSTree

```
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);
```

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(arg0: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(arg0:null;)` (one transition is state-preserving being extracted from T1), the transition `remove(arg0:null;)` from the second-to-top state to the bottom state is indeterministic, being attached with `[1/2]`. We draw thicker edges for nondeterministic transitions so that programmers can easily identify them based on visual effect.

To present a succinct view, by default 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. One example of such a combined transition is the rightmost transition from the rightmost state to the bottom state.

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 [19], test minimization [8], and test generation [9]. 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 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 abstraction ratios for the OSM's in Figure 1 and

Figure 2 are 0.29 and 0.58, respectively. 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. In fault isolation, 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(arg0:null;)` after calling the constructor leads to an undesirable state: calling `contains` on this state deterministically throws the exception. In addition, calling `remove(arg0: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(arg0:null;)`, two `remove(arg0: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`.

In test characterization, 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(arg0.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(arg0.v:0)=false` and the indeterministic transition of `remove(arg0:null;)?/-[1/2]` from the second-to-top state to the bottom state is turned into a deterministic transition `remove(arg0: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 [20]. A `repOK` and some helper methods were added to this class for evaluating Korat [2]. We also used this class in our previous work for evaluating Rostra [8]. 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 object states in addition to the initial state and

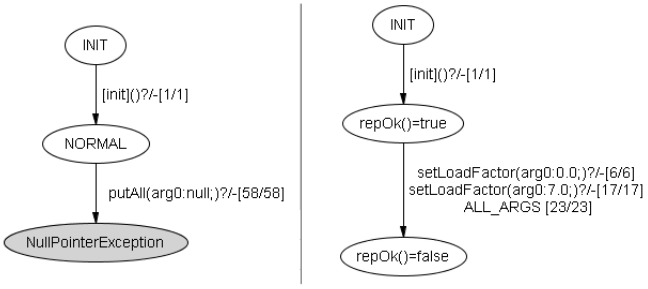


**Fig. 3.** `exception` observer abstraction and `repOk` observer abstraction of HashMap

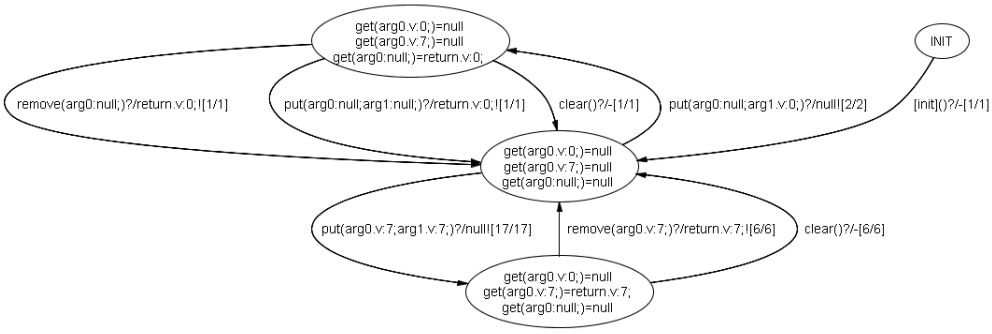


**Fig. 4.** `get` observer abstraction of HashMap

one exception state, 29 nonequivalent non-constructor method calls in addition to one constructor call, and 416 nonequivalent method executions. Obstra augments 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.

In fault isolation, we found that the exception OSM of HashMap (with the abstraction ratio of 0.95) contains one deterministic transition `putAll(arg0: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 [20] 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.

In correctness checking, we found an error in `setLoadFactor(float f)`, a method that was later added to facilitate Korat's test generation [2]. The right part of Figure 3 shows the `repOk` OSM of HashMap (with the abstraction ratio of 0.95). `repOk` is a predicate used to check class invariants [14]. 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. The comments for a private

field `threshold` states that the value of `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 (with the abstraction ratio of 0.94). In the representation of method returns on a transition or in a state, `return` represents the non-primitive return value and `return.v` represents the `v` field of the non-primitive return value. We next walk through the scenario in which programmers could inspect Figure 4 for correctness checking and component understanding. 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(arg0:null;arg1: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 [20], 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 documentation for the returns of `get` is not accurate or consistent with the description of `get`.

Three observers of HashMap, such as `Collection values()`, have a low abstraction ratio of 0.28; thus, their OSM's are large and difficult to inspect. In future work, we plan to display a portion of their OSM's based on user-specified filtering criteria.

## 4 Related Work

Ernst et al. use Daikon to dynamically infer likely invariants from test executions [21]. Invariants are in the form of axiomatic specifications. These invariants describe the observed relationships among the values of object fields, method arguments, and method 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 [7]. Their discovered algebraic specifications present a local view of relationships among 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 provide useful information during inspection. In summary, observer abstractions is a useful form of inferred properties, complementing invariants or algebraic specifications inferred from unit-test executions.

Whaley et al. extract Java component interfaces from system-test executions [22]. 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 method-exit states of the same state-modifying method 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 [23]. Their approach faces the same problem as Whaley et al.'s approach when being applied on the execution 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 does and neither of them can be applied on the executions of automatically generated unit tests.

Given a set of predicates, predicate abstraction [24–27] 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 [28]. 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. Gallagher and Offutt extend the finite state machines of a single class to multiple classes in integration testing [29]. These previous approaches specify specifications in form of finite state machines and generate tests based on the specifications, whereas our approach extracts observer abstractions in 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 [13] 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 [30].

Kung et al. statically extract object state models from class source code and use them to guide test generation [31]. An object state model is in 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 [1]. 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 are corresponding 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 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. The observer abstraction approach can help address correctness checking, fault isolation, test characterization, and component understanding. We have applied the approach on a variety of programs, including complex data structures; our experiences show that extracted observer abstractions provide succinct and useful information for programmers to inspect.

## Acknowledgments

We thank Arnaud Gotlieb, Amir Michail, Andrew Peterson, and Vibha Sazawal for their valuable feedback on the 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. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: Proceedings of the international symposium on Software testing and analysis, ACM Press (2002) 112–122
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: Proceedings of the international symposium on Software testing and analysis, ACM Press (2002) 123–133
3. Parasoft: Jtest manuals version 4.5. Online manual (2003) `http://www.parasoft.com/`.
4. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 231–244
5. Csallner, C., Smaragdakis, Y.: Jcrasher documents. Online manual (2003)
6. Pavlopoulou, C., Young, M.: Residual test coverage monitoring. In: Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press (1999) 277–284
7. Henkel, J., Diwan, A.: Discovering algebraic specifications from java classes. In Cardelli, L., ed.: 17th European Conference on Object-Oriented Programming, Darmstadt, Germany, Springer (2003) 431–456
8. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. Submitted to 2004 International Conference on Automated Software Engineering (ASE 04) (2004)

9. Xie, T., Marinov, D., Notkin, D.: 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 (2004)
10. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R.: Space-reduction strategies for model checking dynamic systems. In: Proceedings of the 2003 Workshop on Software Model Checking. (2003)
11. Iosif, R.: Symmetry reduction criteria for software model checking. In: Proceedings of the 9th SPIN Workshop on Software Model Checking. Volume 2318 of LNCS., Springer (2002) 22–41
12. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proc. 15th IEEE International Conference on Automated Software Engineering (ASE), Grenoble, France (2000)
13. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of The IEEE. Volume 84. (1996) 1090–1123
14. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley (2000)
15. Dahm, M., van Zyl, J.: Byte code engineering library (2003)
16. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley Longman Publishing Co., Inc. (2000)
17. : JUnit (2003) `http://www.junit.org`.
18. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software: Practice and Experience **30** (2000) 1203–1233
19. Xie, T., Notkin, D.: Tool-assisted unit test selection based on operational violations. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2003) 40–48
20. Sun: Javatm 2 platform, standard edition, v 1.4.2, api specification. Online documentation (2003) `http://java.sun.com/j2se/1.4.2/docs/api/`.
21. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27** (2001) 99–123
22. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: Proceedings of the international symposium on Software testing and analysis, ACM Press (2002) 218–228
23. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 4–16
24. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: Proceedings of the 9th International Conference on Computer Aided Verification, Springer-Verlag (1997) 72–83
25. Visser, W., Park, S., Penix, J.: Using predicate abstraction to reduce object-oriented programs for model checking. In: Proceedings of the third workshop on Formal methods in software practice, ACM Press (2000) 3–182
26. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, ACM Press (2001) 203–213
27. Dwyer, M.B., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C.S., Zheng, H., Visser, W.: Tool-supported program abstraction for finite-state verification. In: Proceedings of the 23rd international conference on Software engineering, IEEE Computer Society (2001) 177–187
28. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: Proceedings of the Conference on Software Maintenance, IEEE Computer Society (1993) 302–310
29. Gallagher, L., Offutt, J.: Integration testing of object-oriented components using finite state machines. Draft under review (2002)

30. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Proceedings of 3rd International Workshop on Formal Approaches to Testing of Software. Volume 2931 of LNCS., Springer (2003) 60–69
31. Kung, D., Suchak, N., Gao, J., Hsia, P.: On object state testing. In: Proceedings of Computer Software and Applications Conference (COMPSAC94), IEEE Computer Society Press (1994) 222–227