Inner Oracles: Input-Specific Assertions on Internal States

Yingfei Xiong, Dan Hao, Lu Zhang, Tao Zhu, Muyao Zhu, Tian Lan
Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
School of EECS, Peking University, China
{xiongyf04,haod,zhanglu,zhutao10,zhumy12,lantian12}@sei.pku.edu.cn

ABSTRACT

Traditional test oracles are defined on the outputs of test executions, and cannot assert internal states of executions. Traditional assertions are common to all test execution, and are usually more difficult to construct than on oracle for one test input. In this paper we propose the concept of inner oracles, which are assertions on internal states that are specific to one test input. We first motivate the necessity of inner oracles, and then show that it can be implemented easily using the available programming mechanisms. Next, we report two initial empirical studies on inner oracles, showing that inner oracles have a significant impact on both the fault-detection capability of tests and the performance of test suite reduction. Finally, we highlight the implications of inner oracles on several research and practical problems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Testing, Test oracles, Assertions

1. INTRODUCTION

The capability of a test suite to detect faults is decided by two factors: 1) how well the test inputs trigger faults, and 2) how well the oracles catch the triggered faults. Traditional research efforts, such as test generation [10, 6] and test coverage criteria [19, 4], focus on test inputs. Recently, Staats et al. [14] show test oracles are as important as test inputs in deciding the capability of a test suite to detect faults.

Traditionally, test oracles [1] are defined on the outputs of test executions [15]. However, there are cases where the faults are triggered but not propagated to the outputs, and in these cases we cannot capture the faults using traditional test oracles. For example, a fault may result in an erroneous value in a temporary variable, but the later execution may happen to produce the correct result with this wrong value. Recently, Staats et al. [14] demonstrate that test oracles defined on internal states of the software under test can further improve the power of testing systems. However, the main existing mechanism for checking the internal states is to use the `assert` statement within production code, which is usually defined to check properties common to all inputs. Such an assertion is not always easy to construct because we need to consider all possible inputs.

In this paper, we propose a novel concept, inner oracle, which is an assertion declared on the internal state and is specific to one test input. We show that this concept can be conveniently implemented using existing programming mechanisms, is easier to construct than traditional assertions, and is useful in many aspects of testing.

We also report two initial empirical studies on inner oracles. First, we performed the first quantitative study on how much inner oracles can improve the fault-detection capability of existing tests, and analyzed the reasons why inner oracles outperformed traditional oracles on output. These results give a direct empirical evidence that supplements the theory of testing [14]. Second, we also evaluated how inner oracles help reduce the size of a test suite. The result indicates potential usefulness of inner oracles in practice.

2. MOTIVATION

We motivate inner oracles with an example. The following portion of code defines a method of determining whether the number of positive elements in a list is larger than that of negative elements. In particular, the method returns `1` if the number of positive elements is larger, `-1` if the number of negative elements is larger, and `0` if the list contains equal numbers of positive and negative elements. However, Line 6 is incorrect because it wrongly treats zeros as negatives.

```
1: public static int count(List<Integer> a) {
2:    int positive = 0, negative = 0;
3:    Iterator<Integer> i = a.iterator()
4:    while (i.hasNext()) {
5:        if (i.next() > 0) positive++;
6:        else negative++; } //incorrect
7:    if (positive > negative) return 1;
8:    else if (positive == negative) return 0;
9:    else return -1;
10:}
```

This error is obvious, but many tests may fail to reveal this error if only the actual output is checked. The reason is that an incorrect value of `negative` may not always result in an incorrect output. For example, if the input list is $\{-2, -1, 0, 1, 2, 3, 4\}$, the preceding code would calculate an incorrect value of `negative` before executing Line 7 but still can correctly output `1`. It is not convenient to check the value of `negative` using an input-unspecific assertion, because, at least before Java 8[1], another round of iteration over the input list may be unavoidable to obtain the correct number of negative elements in the list. On the other hand, checking with inner oracles is relatively easy: given the input $\{-2, -1, 0, 1, 2, 3, 4\}$, it is quite convenient to have the assertion `assert(negative==2)` before executing Line 7.

3. APPROACH

Given that inner oracles can sometimes be more convenient to define, in this paper we argue that inner oracles should be taken into consideration in both real-world testing and research in test generation. More concretely, given a test input and a point of execution with the input, an *inner oracle* asserts whether the actual internal state at that particular point of execution satisfies certain properties related to the given test input.

We first demonstrate that an inner oracle can be easily defined using the existing mechanisms. To declare an inner oracle in our running example, we can introduce a guard variable when we declare an assertion. In the running example, instead of directly declaring `assert(negative==2)`, we can insert the statement as shown in Line 6a below:

```
6:        else negative++; } //incorrect
6a:   assert(!CountTest.testGuard1 || negative == 2);
7:    if (positive > negative) return 1;
```

The variable `testGuard1` is a Boolean variable declared in the test class `CountTest`. Initially this variable is assigned `false`, so that this assertion is disabled in all tests. To enable this test for a specific test input, we assign `true` to `testGuard1`, as shown in the following code.

```
1: class CountTest {
2:   public static boolean testGuard1 = false;
3:
4:   @Test
5:   public void test1() {
6:     List<Integer> a = ...;
             // construct list {-2, -1, 0, 1, 2, 3, 4}
7:     testGuard1 = true;
8:     count(a);
9:     testGuard1 = false;
10:  }
11:}
```

Variable `testGuard1` is assigned `true` only within the context of the specific test, and thus the guarded assertion is only enabled for the specific test input, while asserting on an internal state of the test execution.

If there is a large number of tests, this direct implementation may introduce a lot of guarded assertions into the production code, breaking the understandability of the code and causing extra runtime overhead. To overcome this problem, we have built a specialized supporting mechanism for inner oracles, borrowing the idea of aspect-oriented programming [9]. In our supporting mechanism, inner oracles are declared within test code, and are woven into production code at compilation time. This supporting mechanism is implemented as an open source tool. More information about this mechanism and the tool can be found at the tool website[2].

[1]In Java 8, it is possible to get the value of `negative` by the high-level method `filter`, but this requires advanced knowledge of the language.

[2]`http://ayzk.github.io/InnerTest/`

4. ENHANCING FAULT-DETECTION CAPABILITY

Based on the theory of testing [14], adding inner oracles to existing tests should improve fault-detection capability. In this section we try to empirically understand the upper bound of this improvement in practice. In particular, we focus on how much extra improvement can be obtained by adding inner oracles on top of traditional oracles.

In our evaluation, we used four Java programs as subjects: *jodatime* v1.5.2[3], *timeandmoney* v0.2[4], *barbecue* v1.5.0[5], and *xmlsec* v1.2.1[6], which have been widely used in the literature of software testing and analysis [3, 11, 7]. We ran Javalanche [13] to generate mutants as injected faults. Since the generated mutants are more than 45,000 , we randomly selected 0.3% of mutants as our subjects. We also manually reviewed the mutants to remove the equivalent mutants. As a result, we got 139 mutants on the programs. Each test and each mutant form a test-fault pair, and thus we got 97,582 test-fault pairs. Table 1 lists the statistics about the subject programs and the generated faults.

Next, we classify the test-fault pairs by whether the test triggers the fault or not. The untriggered pairs can be identified by examining two causes: (1) the test does not cover the mutation; (2) the test covers the mutation, but the mutation does not change the runtime state. An example of the second case is mutating `a*2` to `a+2`, but `a` is 2 when evaluating the expression. The "Untriggered" part in Table 2 shows the number of untriggered pairs based on the two causes.

The more interesting category is the triggered pairs. We further classify these test-fault pairs into four categories: (1) the original test detects the fault; (2) if not, adding more traditional oracles on the output can detect the fault; (3) if not, adding inner oracles can detect the fault; (4) none of the above. The first category is easy to identity. To identify the other three categories, we try to manually add `assert` statements to the test code and to the production code as inner oracles. We constrain that within an `assert` statement we can only invoke *observer* methods, which do not change the state of the software, such as get methods. We avoid non-observer methods because, in theory, we actually changed the test input by calling them.

The classification result is shown as Detected$_o$, Detected$_a$, Detected$_i$, Undetected in Table 2. From the result we can see that, there are 30.72%-69.65% pairs where the fault is triggered but cannot be detected by traditional oracles on output, and these pairs can all be detected by inner oracles. That is, inner oracles may significantly boost the fault-detection capability of tests.

To further understand why traditional oracles cannot detect the faults, we manually investigated the code and found two main reasons. The first one is that the fault results an error in an intermediate state, but this error is not propagated to the final state. In total, 294 of the 1369 pairs are caused by this reason. Our running example is also caused by this reason. The second one is that the fault results an error in the final state, but this error is not visible to the test code due to the accessibility rules, e.g., a private member of an object may contain an erroneous value, but this value is

[3]`https://github.com/JodaOrg/joda-time`

[4]`http://timeandmoney.sourceforge.net`

[5]`https://github.com/cloudsoft/barbecue`

[6]`https://www.aleksey.com/xmlsec/`

Table 1: Statistics of subjects

Subject	KLOC	#Method	#Class	#Test	#Mutant	Pairs
jodatime	25.8	3276	198	3417	26	88842
timeandmoney	1.1	262	30	104	39	4056
barbecue	8.0	283	55	51	50	2550
xmlsec	16.2	1213	181	97	22	2134

Table 2: Fault-Triggering Capability and Fault-Detection Capability of Tests

Subject	#Total	#Untriggered			#Triggered				
		#Total	#Uncovered	#Equiv	#Total	#Detected$_o$	#Detected$_a$	#Detected$_i$	#Undetected
jodatime	88842	86947	86811	136	1895	445	760	690	0
timeandmoney	4056	3519	3488	31	537	120	43	374	0
barbecue	2550	2398	2360	38	152	29	27	96	0
xmlsec	2134	1828	1767	61	306	207	5	94	0

not accessible by the test code. On the other hand, inner oracles can access such members as they can be declared within the target class. In total, 1075 out of 1369 pairs are caused by the second reason.

5. REDUCING TEST SUITES

Since adding inner oracles enhances the fault-detection capability of tests, it would be interesting to see whether inner oracles help further reduce the size of test suites. By performing Study I (in Section 4), we had two test suites for each subject project, one augmented with traditional oracles and the other one augmented with inner oracles. Then we perform test suite reduction on the two suites to cover the mutants we used in the subjects. Since the test suites with traditional oracles are not able to kill all mutants, we only consider the subset of mutants that can be killed by the test suite with traditional oracles. In particular, for *jodatime* we had 15 mutants, for *timeandmoney* we had 27 mutants, for *barbecue* we had 7 mutants, and for *xmlsecurity* we had 20 mutants. The test suite reduction algorithm we used is the greedy algorithm by Zhang et al. [17].

Table 3 presents the reduced test suites with and without inner oracles. Row "#With traditional oracles" shows the number of tests in the reduced test suite with added traditional oracles, and Row "#With inner oracles" shows the number of tests in the reduced test suite with added inner oracles. Since we used a small set of mutants due to the limitation of manual investigation in previous study, the reduced test suites are very small. This result is consistent with existing studies [17, 18] where the size of the test suite shrinks dramatically after reduction.

From the table the test suites with inner oracles are further reduced by 14.3%-50.0%. That is, when inner oracles are used, test suites may be significantly further reduced.

6. DISCUSSION

Versus Test Inputs. In the running example, besides adding an inner oracle, we can also add a new test with a different test input to capture the fault. This observation leads to the question whether inner oracles are necessary at all. Here we list a few scenarios that inner oracles are more useful than new tests. First, some type of bugs do not affect the final state, such as the optimization bug discussed later in this section. Adding more test inputs and traditional oracles cannot help catch these bugs. Second, additional test inputs lead to longer execution time of the test suite, which is one of the main problems in testing large software systems. Third, test inputs may not be easily constructible. In a complicated case, a method may depend on a non-trivial global state, which requires creating a set of objects, and mock objects in case the state of the objects are not easily manipulatable.

Versus Code Refactoring. Another point of view is that, when it is difficult to test some part of software, it is a smell of bad code and we should refactor the code to enhance testability, rather than writing complex testing code or seeking new test mechanisms. This view is often advocated in text books [2] by practitioners. For example, we can extract two methods from the running example, one counting the number of positives and one counting the number of negatives, and then we can test the two methods individually without using inner oracles.

However, testability is just one quality attribute of the code, and there are many other quality attributes that are often contradict with testability. The above method extractions require to traverse the list twice rather than once, which may lead to performance penalty unacceptable in critical code. A further optimization is that, when the number of positives or the number of negatives is larger than half of the list, we can directly return the result without looking at the rest of the elements, and this optimization is impossible to apply when the methods are extracted. In those cases, inner oracles help maintain the desirable fault-detection capability without sacrificing performance.

Test Optimization. A common task in programming is optimization. Since optimization does not change the output of the program, it would be difficult for a test to check whether the optimization is effective or not. On the other hand, an inner oracle can easily assert on the internal state to check the existence of an optimization. For example, the following code shows a typical optimization, where shift is used instead of multiplication when the parameter `b` is two.

```
1: int times(int a, int b) {
2:   if (b == 2)
3:     return a << 1;
4:   else
5:     return a * b;
6: }
```

It is difficult to test the existence of the optimization using traditional oracles. To test the program using inner oracles, we can easily add an assertion `"false"` right before Line 5 for an input where b = 2.

Debugging. During debugging, inner oracles usually provide more information for localizing a fault than traditional

Table 3: Reduced Test Suites With and Without Inner Oracles

Subject	*jodatime*	*timeandmoney*	*barbecue*	*xmlsec*
#With traditional oracles	10	9	4	7
#With inner oracles	8	5	2	6

oracles on the output: only the statements executed before the inner oracle may contain the fault, rather than all statements executed. Note the information for fault localization is available for tools the same as for human developers, so inner oracles also provide opportunities for improving automated debugging techniques.

Mutation Testing. Our methodology may also have implications on mutation testing [5]. The long execution time is one of the biggest weakness of mutation testing. An existing study [8] has found out using weak mutation, in which whether a test detects a mutant is decided by whether the mutant results in an incorrect internal state, can significantly shorten the execution time by sacrificing precision. With inner oracles, the sacrificed precision may be much smaller, and research efforts on mutation can put more weight on weak mutation testing.

Regression Test Generation. A lot of research efforts have been devoted into the automatic generation of regression tests, and in particular, the generation of test oracles in regression testing [16]. In the generation of test oracles, a major challenge is to know what part of the final state is affected by the current test execution and how to access the part of the state using the public members. With inner oracles considered, we can directly generate oracles at the places where internal states are changed, without the need to analyze its effects on the final state. Furthermore, we do not need to construct the access path using the public members, because we can directly obtain the path from the statement changing the internal state. As a result, regression test generation can be greatly simplified, and the generated tests are probably stronger based on our empirical results.

Maintainability. Since inner oracles are declared within the implementation of components but not their interfaces, inner oracles are likely to be more fragile than traditional oracles on output during the evolution of source code. As a result, further investigation is needed to clarify the maintainability of inner oracles. However, this problem is probably not serious in practice. First, traditional assertions share the same fragility of inner oracles, and no maintenance problem is reported for them. Second, Pinto et al. [12] demonstrated changing test oracles accounts for only a small portion of test evolution. Although inner oracles is more fragile than traditional oracles on the output, the increased maintenance cost on oracles may still be small compared to the overall maintenance cost.

7. ACKNOWLEDGMENTS

This work is supported in part by the High-Tech Research and Development Program of China under Grant No.2015AA01A203, the National Natural Science Foundation of China under Grant Nos.61421091, 61332010, and 61272157.

8. REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE TSE*, 41(5):507–525, 2014.

[2] K. Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[3] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[4] C. Fang, Z. Chen, and B. Xu. Comparing logic coverage criteria on test case prioritization. *SCIENCE CHINA Information Sciences*, 55(12):2826–2840, 2012.

[5] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE TSE*, 3(4):279–290, 1977.

[6] D. Hao, L. Zhang, M. Liu, H. Li, and J. Sun. Test-data generation guided by static defect detection. *JCST*, 24(2):284–293, 2009.

[7] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test-case prioritization approach. *ACM TOSEM*, 24(2):1–31, 2014.

[8] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE TSE*, 8(4):371–379, 1982.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming.* Springer, 1997.

[10] P. McMinn. Search-based software test data generation: A survey. *STVR*, 14(2):105–156, 2004.

[11] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing JUnit test cases. *IEEE TSE*, 38(6):1258–1275, 2012.

[12] L. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proc. FSE*, pages 1–11, 2012.

[13] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. FSE*, pages 297–298, 2009.

[14] M. Staats, M. W. Whalen, and M. P. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proc. ICSE*, pages 391–400, 2011.

[15] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE TSE*, 18(8):717–727, 1992.

[16] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403. Springer, 2006.

[17] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *Proc. ISSRE*, pages 170–179, 2011.

[18] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *IST*, 50(6):534–546, 2008.

[19] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE TSE*, 22(4):248–255, 1996.