Tests as Documentation: a First Attempt at Quality Evaluation

Maura Cerioli and Giovanni Lagorio

DIBRIS - University of Genova
Genova, Italy
Email: {maura.cerioli,giovanni.lagorio}@unige.it

***Abstract*—We present a novel method, and its associated supporting tool, for automatically singling out *sloppy tests*; that is, tests that run successfully on (some) *incorrect* implementations, that violate the property they are expected to verify. Our freely available tool is written in C#, but the technique is language agnostic and can be easily applied to other languages.**

Keywords–Testing; Debugging.

I. Introduction

Test methods, for instance those written using a framework of the *x*Unit family [1], simply called *tests* from now on, have initially been introduced in software development process for *unit testing*, that is, testing of small units of code during their development, more than thirty years ago [2].

More recently, tests have been also used to capture information about the code to be developed, playing in some sense the role of *running specifications*. This is the case, for instance, of the test-driven approach [3], where tests are developed along with the code, and used to improve the developer understanding of the required code, making it explicit. Though tests in the test-driven approach are aimed more at knowledge capture than code improvement, they are still *white box*, written by the software developers taking advantage of private code structures for the set up.

A more innovative use of tests is in refactoring and/or migration of legacy systems [4], or in iterative development processes. Indeed, in those cases, tests are defined and verified against a version of the system, be it the system to be refactored or the current iteration prototype, but are intended to be run on the *next versions*, still to be developed at the moment such tests are written. In this way, tests capture observable behaviors of the system, approved by the stakeholders on the current version, and guarantee the next version to preserve such behaviors, complementing (or altogether replacing) the corresponding documentation.

To put the system in the required state, before the call of the method to be tested, this approach requires the tests not to rely on the internal structure of the system, that is going to change. Instead, each operation has to go through the *interface* of the system [5][6][7], in a *black box* style. Moreover, the design of the overall test suite cannot be driven by the current implementation, as it is going to change, and all the adequacy criteria based on code coverage are unreliable.

Analogous problems arise for tests distributed along the specification of components/services, as convincing evidence of their correctness [8]. Indeed, when the tests are used to capture knowledge about the functionalities of a system, they cannot rely on its internal structure when preparing the initial state for the *call under test*. Otherwise, they would risk undue disclosures of the system implementation to users, who have full access to the tests, and the approach would be brittle against changes to the implementation.

Moving from white box tests used to improve the system implementation, to black box tests used to document the system, requires to change the definition of test quality, as well as the techniques to evaluate it. Indeed, when tests are aimed at improving the technical quality of the system under test, it may suffice that the overall test suite is capturing enough bugs. Thus, in literature, we find plenty of techniques to assess the quality of a test suite, by measuring how extensively the test suite, as a whole, exercize the system. The exact meaning of *extensively* may vary, giving rise to different quality criteria. For instance, statement/branch/multiple condition coverage [9], or mutation testing [10].

However, in our target cases, tests are used as *living documentation* [11]. Thus, the description of each *individual* test must correspond to its implementation, because stakeholders and maintenance staff will rely on those test descriptions to understand the system behavior. Therefore, in this setting, we need to assess the quality of *each individual test*, as opposite to the quality of the overall test suite. Moreover, the meaning of quality is also different w.r.t. standard approaches, because a test has a *high quality* when it strictly conforms to its description, disregarding both its capability to spot bugs, and the portion of system it exercises.

Therefore, standard evaluation techniques are not appropriate, and we propose here a different approach.

The first step to get high-quality tests is to verify their *correctness*, that is, that they run successfully on a correct implementation of the system. In other words, tests, as any other software, need to be tested. Such a necessity is partially reduced by their intrinsically limited complexity. However, even when writing small tests, it is rather easy to introduce mistakes or to misinterpret their goals. There are basically two approaches to test verification: inspection by a human reader, as peer review can improve the quality of tests [12], and the execution on a reference implementation, known to be correct. The former method permits, in a single pass, to verify the correctness of the tests and evaluate their quality, in terms of correspondence to their definition. However, it is extremely time consuming [13], hence expensive. Moreover, as tests are many and often quite similar, the attention level of the human inspector and, accordingly, the number of detected imperfections/mistakes may decrease. Finally, such costs have to be sustained whenever tests are updated.

The automatic verification by a reference implementation, on the other hand, is widely used in practice whenever a reference system is available during the implementation of the tests, as in the cases we are addressing. In such settings, the reference system is assumed to be a correct *oracle* so, if a particular test fails, then it is known *a priori* to be incorrect. Once tests appear to be *correct*, i.e., they successfully run on the oracle, their *quality* has to be evaluated.

To carry on this task in automatic verification style, we need slight variations of the system. Each of these variations, that we call *anti-oracles*, intentionally violates the description of a specific test t, and hence it is a prospective victim of the t, that should be able to kill it. Then, from the outcome of t on that anti-oracle, we can get precise information about the adequacy of t w..r.t. its description, that is, on its *quality*.

To keep the design effort of such anti-oracles sustainable, we propose a method to instrument the oracle. The instrumented oracle, I, can *behave* both as a correct implementation *and* as the needed anti-oracles, in different runs.

Our method is supported by a lightweight tool, which takes care of differentiating the runs on I. Note that, in our target scenarios, tests, in their setup code, must (only) use the very same elements of the public interfaces under test. Thus, our anti-oracles, to be effective, should *behave correctly on all calls, except for the call under test*. This fact imposes further requirements on the design of the supporting tool.

Our approach is reminiscent of the mutation testing technique, in that our anti-oracles are variation of the oracle. But, while mutants are randomly generated and used to estimate the probability of the whole test suite to detect technical bugs, each anti-oracle is *designed* to target an individual test and verify the adequacy to its description. Thus, the mutation testing technique cannot address the problem we are interested into. We introduce our method in Section II, and we briefly sketch its implementation in Section III. A preliminary evaluation of our method is provided in Section IV, and in Section V we discuss its relations to mutation testing, while some conclusions are drawn in Section VI.

II. Proposed Method

We consider basic standard test methods consisting of three parts: the *setup*, usually few lines of code to initialize the status of the system, the *call under test*, that is the specific method invocation whose behaviour is verified by the test, and, finally, an *assertion* stating properties about the result of the call, in terms of both the yielded value, if any, and the resulting state of the system.

Our method assumes the existence of a working system, to be used as the oracle, and of the specifications of the tests to be implemented. Such specifications should be as accurate as possible, but cannot be expected to be formally expressed in a rigorous specification language, like, for instance, some kind of logic. Indeed, in most realistic cases it is not possible, or highly inconvenient, to formalize the properties to be checked.

This limitation rules out the possibility of automatically generating tests from their formal specification (such tests would be obviously consistent with their specification *by construction*, provided the generator to be correct).

Let us clarify the expected level of formality of test specifications on a toy example, written in C# and using NUnit [14] (however, the idea is independent from both). The class `IntStack`, shown in Figure 1, implements a very basic stack of integers as a tiny wrapper on the standard generic class `Stack<>`.

```
public class IntStack {
  private Stack<int> _stack = new Stack<int>();
  public int Size(){
    return this._stack.Count;
  }
  public void Push(int i){
    this._stack.Push(i);
  }
  public void Pop(){
    this._stack.Pop();
  }
  /* ... */ }
```

Figure 1. A stack of integers.

Using this prototype, we want to polish a set of tests, for instance, targeting the method `Push`:

- `PushDoesNotAffectPreviousElements`
- `AfterPushSizeIsPositive`
- `PushIncreasesSizeFrom3To4`
- `PushAddsElement`

`PushDoesNotAffectPreviousElements` could be specified by: "*after pushing a number on a stack, already containing some items, they will be still on the stack and in the same order*".

The goal is verifying the individual tests to be adequate w.r.t. their specification. The oracle (i.e., the reference implementation) is used first to verify that all the tests appear to be *correct*, that is, successfully running on the oracle. The next step of our method is to verify that they are also *sufficiently strict*. To this end, we first derive, from each test specification, a list of possible mistakes, which the test should be able to detect accordingly to its specification. For instance, for `PushDoesNotAffectPreviousElements`, the list of possible mistakes includes: one of the original items is dropped; one of the original items is replaced by another number; two of the original items are swapped.

Then, each mistake from such a list is implemented by an *anti-oracle*, which should replace the correct oracle implementation to answer the *call under test*, and only *that* particular call, making the test fail (if it is sufficiently strict).

For instance, `AfterPushSizeIsPositive` should be able to capture anti-oracles where the size is zero after an element has been pushed. So, the body of `Push`, in such a anti-oracle, could simply empty **`this`**`._stack`.

Finally, we instrument the oracle to derive an enriched system able to make the call under test fail for each test. At this aim we define a utility class `FindCaller` that, depending on a global switch (for instance, the value of an environment variable), may operate in two modes: *record* and *evaluate*.

The basic idea is that in *record mode* the enriched system behaves like the oracle, so all tests must be successful, while our utility class logs some information about the execution, which are needed for subsequent runs in *evaluate mode*. In such a mode, instead, for each test t, the anti-oracle(s) designed for t is used to answer the call under test by t, while all other calls are answered by the (correct) oracle methods. Thus, in

```
public void Push(int i) {
  string caller = FindCaller.GetTestName();
  if (caller == "AfterPushSizeIsPositive" ||
       caller == "PushIncreasesSizeFrom3To4") {
    this._stack.Clear();
    return;
  }
  if (caller ==
      "PushDoesNotAffectPreviousElements") {
    int previous = this._stack.Pop();
    this._stack.Push(previous+123);
    // continue as if nothing has happened
  }
  if (caller == "PushAddsElement") {
    this.Push(i+1);
      // the *direct* caller for this recursive
      // call is not PushAddsElement, so we get
      // normal behavior for this call
    return;
  }
// everything as before...
// (that is, the behaviour of the oracle)
```

Figure 2. Instrumented `Push` method.

```
[Test]
public void PushIncreasesSizeFrom3To4() {
  IntStack s = new IntStack();
  for (int i=0; i<=2; i++)
    s.Push(i);
  int sizeBeforePush = s.Size();
  s.Push(3);
  Assert.That(s.Size(),
               Is.Not.EqualTo(sizeBeforePush));
}
```

Figure 3. An example of a sloppy test.

this mode, all tests should fail.

The class `FindCaller` offers the method `GetTestName`, whose behavior changes dramatically depending on the operating mode:

- in *record* mode, `GetTestName` always returns **`null`**, but also logs some method call information, to be later used in *evaluate mode*;
- in *evaluate* mode, `GetTestName` returns the name of a test-method t if it is invoked by the call under test by t; otherwise it yields **`null`**.

Thus, to decide if the standard implementation of a method, or its anti-oracle, failing on a test-method t, has to be used, we can simply check if `GetTestName()` returns the string t. Hence, we can inject the failing anti-oracle into our oracle and have a single software product P to maintain (we will address how to keep the actual implementation and its anti-oracles separate in Section VI). This product P behaves, in *record* mode, as the original oracle, while in *evaluate* mode behaves as the needed sophisticated anti-oracles previously discussed.

Figure 2 shows how to instrument the method `Push` in P.

In *evaluate* mode, the instrumented code behaves as the provided failing anti-oracle for each test-method, on its call under test. Hence, all tests should fail. Yet, if we ran the test shown in Figure 3, we would discover that it still passes.

This points out an inadequacy of the test (i.e., the test is sloppy): indeed, instead of asserting that the `Size()` of the stack, after the `Push(3)`, is *different* than before, it should assert that the size *has increased by one*.

Notice that, in the code of anti-oracles, all the internal structure of the oracle can be used, including calls to the public methods, because such calls will be automatically answered by the oracle code. Thus, each anti-oracle is usually implemented by very few lines of code, in most cases just a single one.

Every time a verification fails, that is, tests pass in *evaluate mode*, a refinement step is needed. However, our method does not prescribe how to perform it. Thus, it can be plugged on different processes for improving test quality.

III. Implementation of the Utility Class

As described in the previous section, our utility class `FindCaller` offers the static method `GetTestName` that allows any implementation method m to know the name of the running test t, when the current call of m is the call under test of t (otherwise, `GetTestName` simply returns **`null`**).

The simplest way to detect if the current call of m is the call under test, for some test t, would be to require call under test to be annotated in some way (e.g., by some attribute, and use such information via reflection). However, we want to be able to *evaluate existing tests as they are*, without having to tamper with them, so the implementation of `GetTestName` is more challenging.

Here, we just sketch the idea, since our C# implementation takes into consideration some technical details that are not particularly relevant. We refer interested readers to the freely available source code [15].

In order to understand if a call to a method m, of the implementation, may be the call under test for some test t, `GetTestName` simply rules out the cases that cannot be a call under test, which are:

1) m has been called by another implementation method m' (possibly coinciding with m, in case of recursion), instead than directly by some test;
2) there is a (temporally) subsequent direct call to m, by the same test t, hence the current call is just part of the setup.

Thus, `GetTestName` returns t for the last call of *any* implementation method m directly called by t (or one of its auxiliary methods), even if m is not the one tested by t, say m_t. However, this is not a problem, as the condition `FindCaller.GetTestName()=="t"` is only checked inside the instrumented version of m_t, so that returning t, instead of **`null`**, to the call of some other method goes undetected, and is immaterial.

Therefore, the tasks of method `GetTestName` are to identify:

1) the name of the currently running test, say t;
2) if m has been called by another method of the implementation;
3) if this call to m by t is its (temporally) last call to m.

Since almost every programming language keeps the information about the (direct and indirect) callers of a method in the (machine) call-stack, we can address tasks 1 and 2, by performing a stack walk. The remaining task, 3, needs another technique, which is discussed below.

In languages offering reflection/introspection features, like C# and Java, the call-stack is readily available. For instance, in

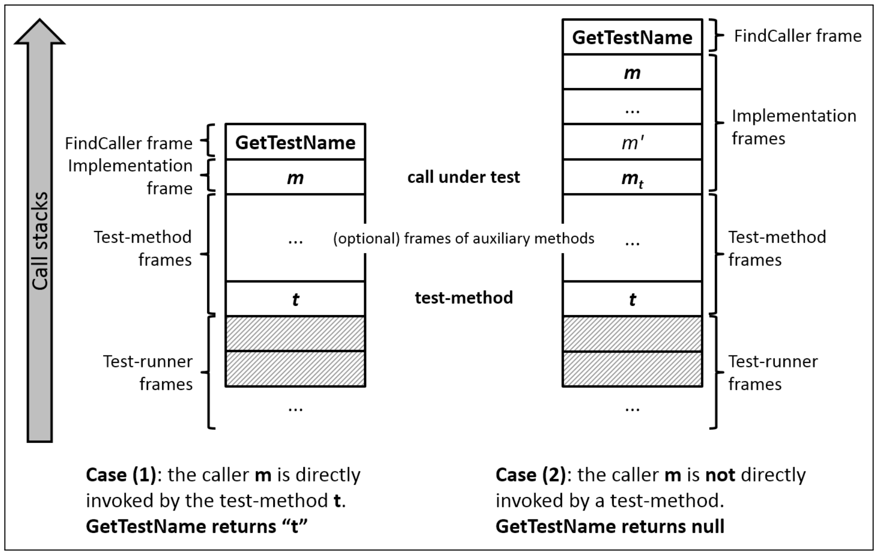


Figure 4. Two examples of call stacks.

C#, using the standard class `StackTrace` we can easily obtain an array of `StackFrame` objects.

Figure 4 shows two examples of the kind of call stacks that `GetTestName` has to deal with. The topmost frame is always for `GetTestName`. The rest of the stack consists of three groups of frames (from the bottom): those of the test runner, those of the test methods, and, finally, those of the implementation methods. The topmost of this latest group, that is, the second frame from the top, belongs to the method m, that has to decide whether to behave as the oracle or some anti-oracle.

The lowest frame of the test method group is the one for t (task 1) and is identified by our utility class by checking whether the method is annotated by one of the custom attributes of NUnit [14]. Of course, this can be easily generalized to other testing-framework.

Task 2 corresponds to checking whether, between the frame of m and the one of t, there are some other frames belonging to implementation methods. Figure 4 shows, side by side, two possibilities:

- left: m is directly called by t (or some of its auxiliary methods), so `GetTestName` must return t (unless Task 3 detects a subsequent call to m);
- right: t invokes m_t, which could for instance be the method under test, and then m_t calls some other implementation methods $(m', \ldots, m)$. Each of these methods, being instrumented, will call `GetTestName`, that must recognize that the current execution of its direct caller (m, in the figure) does not correspond to the call under test. Thus, even if its direct caller coincided with m_t, that is, $m = m_t$, `GetTestName` should return **`null`**.

These two cases are also exemplified in the sequence diagram shown in Figure 5, where the test runner invokes the test `PushAddsElement`, which, in turn, invokes `Push`, which invokes `GetTestName`. In this case, `GetTestName` returns the string `"PushAddsElement"`, so the instrumented `Push` method misbehaves by pushing `(i+1)`, instead of `i` (see Figure 2), by recursively invoking itself. In this second activation, `GetTestNames` returns **`null`**, so no anti-oracle is activated and the call is answered by running the oracle code. Notice that the call of `Top` invokes `GetTestName` too, and gets the string `"PushAddsElement"` as result. But, having no instrumentation for that test, `Top` behaves as the oracle.

Task 3, that is, detecting if this call to m by t is its (temporally) last call to m, can be tackled by exploiting the following idea: since we want any failing implementation (for t) to differ from the oracle only on the call under test c, the execution flow of t, until it reaches c, has to be exactly the same on both the oracle and the anti-oracle.

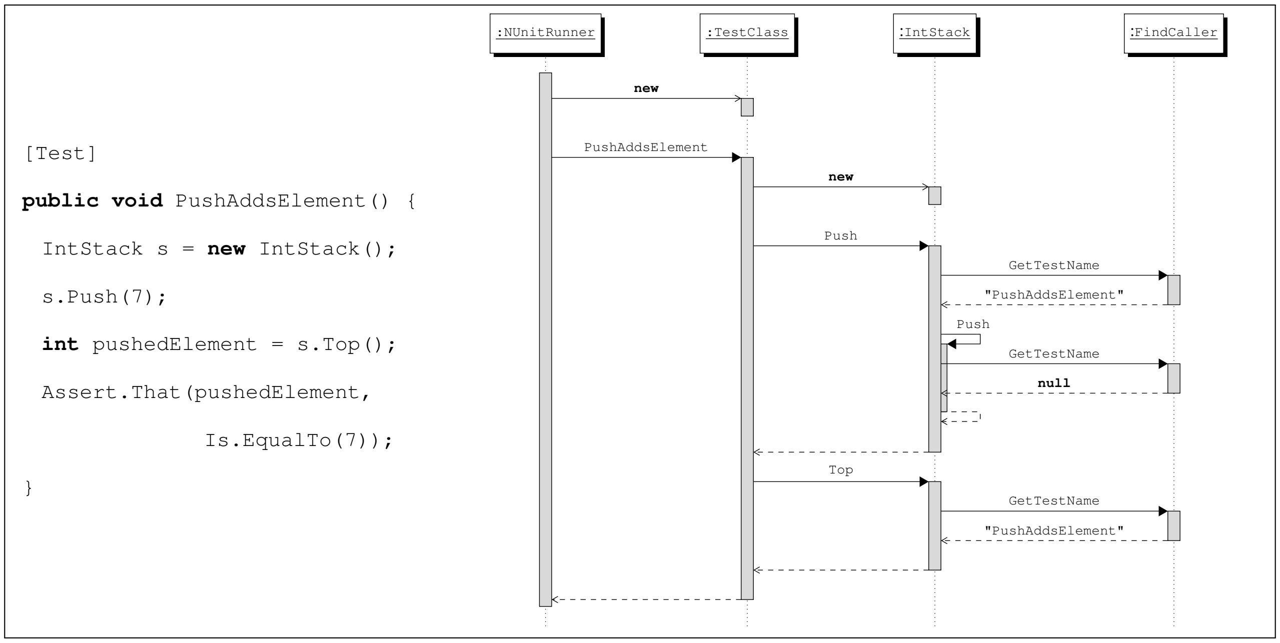
Because all tests pass on the oracle, a single run of all tests in *record mode* allows our class `FindCaller` to collect the information about the order of all the calls that any test makes to any implementation method. After these information have been collected and persisted, they can be used in *evaluate mode* to discern, for each test t, which call is, indeed, the temporally last one.

IV. Preliminary evaluation

The proposed method has been experimented in the context of the project evaluation of an undergraduate course on component based development. Such a project consisted of two independent phases: the development of tests against the specification of a toy component for the management of an auction site, and the implementation of the component itself. The component specification consisted of five small interfaces for about 20 methods and 12 properties, and a few exceptions. The semantics of each method/property was expressed by few lines of text in natural language, as in our running example, and the reference implementation was about 600 lines of code.

The first phase was a collaborative activity by 16 groups of 5 persons each, with the goal of redundantly implementing 150 test specifications. Each test specification, given in natural language, fixed the method to be tested, the call parameters (if any), the required setup of the system, and the expected result.

Each group member was required to individually develop 10 tests and inspect those written by the other group members. Students were equally penalized by errors made during the



```
[Test]
public void PushAddsElement() {
  IntStack s = new IntStack();
  s.Push(7);
  int pushedElement = s.Top();
  Assert.That(pushedElement,
                Is.EqualTo(7));
}
```

Figure 5. `PushAddsElement` and its sequence diagram.

development and peer review. Therefore, they were motivated to carefully read the tests by other members of their team. Indeed, from the discussions going on in a forum for intra-group communications, we know that most students took the assignment seriously and devoted energy and time to get it done at the best of their abilities.

At the end of the review phase, we evaluated the tests using our reference implementation, and 428 out of 590 passed (note that not all the enrolled students completed the project; so only 590 out of the expected 800 tests were submitted for evaluation). Then, we applied our method to those apparently correct, in order to detect sloppy tests: 13 tests out of 428 did not fail as they should have been. That is, about the 3% of the peer-reviewed tests were still slack. Notwithstanding the apparently low value, it is worth noting that:

- 50% of the groups delivered at least one sloppy test, and a group even produced 5 sloppy tests out of 30, as it can be seen in Table I (only groups with at least one sloppy tests have been inserted);
- the tests had been already manually inspected by other members of the group to improve their quality [12];
- for each test specification we implemented just the most obvious failure, hence, capturing only a part of the sloppy tests;
- the given test specifications have been kept very simple to simplify the students' work. With more complex test specifications a higher number of sloppy tests should be expected.

Group	# test methods	Failed	Correct	Sloppy
16	40	10	25	5 (16,67%)
1	30	7	21	2 (8,70%)
4	40	9	30	1 (3,23%)
5	30	6	23	1 (4,17%)
6	40	7	32	1 (3,03%)
7	40	2	37	1 (2,63%)
8	40	8	31	1 (3,13%)
13	30	7	22	1 (4,35%)

TABLE I. EVALUATION RESULTS.

The sloppy tests detected by our experiment can be roughly categorized into three classes:

- Verifying a property weaker than the one expressed by their informal specification. For instance, though required to verify that the result R of some operation is $S = \{a, b, c\}$, they just check that R has three elements, or that $R \subseteq S$, or viceversa. This is by far the most common error, and corresponds to the intuition of sloppy test.
- Verifying the thrown exception to be the one required, but without discriminating if it has been thrown by the method under test, or by some previous call during the test setup. This sloppiness may easily go undetected when system exceptions, like, for instance, `InvalidOperationException` or `ArgumentNullException`, are expected, since they may be thrown in many different situations.
- Making blatantly stupid mistakes, like, for instance, invoking a different method in place of the one to be tested. It may sound unlikely that such evident mistakes are overlooked by reviewers. But, it does happen since their attention is often focused on checking small details, or the logical flow of the test to make sense *per se*, forgetting to check it against its specification.

A threat to the validity of this experiment might be that the subjects were students instead of professionals. Thus, the evaluation could be biased by their limited skills. We plan to apply our technique to the tests of some open-source project in order to estimate its usefulness in a real world context.

V. Related work

To the best of our knowledge, our method is the first one proposed in literature for revealing sloppy test cases. While our work uses an idea similar to *mutation testing* [16] [17], there are substantial differences.

Mutation testing is a technique for evaluating the ability of a test suite in detecting faults, and can also be used as a tool to add new test cases to obtain higher coverage scores. The technique consists of two steps: the creation of mutants and their execution. First, mutants, i.e., clones of the original program with the exception of one random atomic change, are created. For example, a mutant could be produced by changing a binary operator (e.g., "+") into another (e.g., "*") to create a faulty version of the original program. The "rule" that changes an operator with another is called *mutation operator*. Then, the target test suite is executed against all the produced mutants. A mutant is said *killed* if at least a test case belonging to the suite is able to reveal the performed mutation. The test suite adequacy is computed by dividing the number of killed mutants by the total number of mutants.

Although mutation testing is largely recognized as a satisfactory technique for the improvement of a test suite, the aim is revealing parts of system code not exercised by any test, where randomly mutations go undetected. This allows to improve the test suite as a whole, by adding test cases targeting the unexplored parts. But, mutation testing is inadequate for our goal, that is, individual test adequacy against its specification, in a setting where testing must go through the public interfaces of the system. Indeed, mutation testing

- evaluates and improves test suites as a whole instead of individual tests;
- addresses a different concept of quality, without any connection to the users' expectations about the kind of bugs that should be detected by the tests, accordingly to their description;
- could yield a false positive, if mutants are killed by a failing test setup involving the very same methods to be tested; this cannot happen with white-box testing, where the setup explicitly accesses the internal structure of the system, but it is quite common when also tests must go through the public interfaces.

VI. Conclusion and Future Work

We have proposed a method to verify the adequacy of individual tests to their specification. Our method requires to elaborate minimal changes to a reference implementation, making a well written test fail on the resulting anti-oracle. Moreover, our method is supported by a tool, that takes care of having such changes executed only on that test.

Currently, the changes are manually injected into the oracle, except for those tests expecting an exception, where the tool can automatically throws an exception of unexpected type to verify that the test is correctly strict. We plan to use aspect-oriented programming [18] techniques in order to keep separate the code to be injected from the reference implementation. Indeed, we are currently evaluating PostSharp Express [19] for .NET, as a supporting tool.

A further enhancement is allowing several different anti-oracles for the same test t, implementing different bugs t should be able to detect. At this aim, the test-runner should be made aware that some tests need to be run several times, and method `GetTestName` should yield, on the call under test, not only the test name, but also the number of its run, in order to possibly change the behaviour.

The current version of the tool has been preliminarily evaluated by an experiment on the projects of a course. The results were quite encouraging, as we captured 3% of sloppy tests on a population already improved by a preliminary peer-review process. However, they were also obviously limited, being based on the performance of students instead of professionals. Further applications to some industrial sized project are needed in order to estimate its real usefulness.

Acknowledgment

We warmly thank Filippo Ricca for the helpful discussions on mutation testing.

References

[1] P. Hamill, Unit Test Frameworks: Tools for High-Quality Software Development. O'Reilly, 2004.

[2] G. Myers, The Art of Software Testing, ser. A Wiley-Interscience publication. Wiley, 1979.

[3] K. Beck, Test-Driven Development by Example, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[4] A. Marchetto and F. Ricca, "From objects to services: toward a stepwise migration approach for Java applications," International Journal Software Tools Technological Transfer, vol. 11, no. 6, 2009, pp. 427–440.

[5] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-Based automatic test case generation for web services testing," in SOSE '05: Proceedings of the IEEE International Workshop. Washington, DC, USA: IEEE Computer Society, 2005, pp. 215–220.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Systematic generation of XML instances to test complex software applications," in RISE, 2006, pp. 114–129.

[7] H. M. Sneed and S. Huang, "The design and use of WSDL-Test: a tool for testing web services: Special issue articles," J. Softw. Maint. Evol., vol. 19, no. 5, 2007, pp. 297–314.

[8] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in FASE, 2005, pp. 34–48.

[9] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," Communications of the ACM, vol. 6, no. 2, Feb. 1963, pp. 58–63.

[10] W. E. Wong, Mutation Testing for the New Century. Springer, 2001.

[11] G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications, 2011.

[12] F. Lanubile and T. Mallardo, "Inspecting automated test code: A preliminary study," in Proc. of 8th International Conference on Agile Software Development (XP 2007). Springer-Verlag, 2007.

[13] T. Thelin, H. Petersson, P. Runeson, and C. Wohlin, "Applying sampling to improve software inspections," Journal of Systems and Software, vol. 73, no. 2, October 2004, pp. 257–269.

[14] "NUnit," 2014, URL: http://www.nunit.org [accessed: 2014-03-09].

[15] "FindCaller," 2014, URL: http://www.disi.unige.it/person/LagorioG/FindCaller.cs [accessed: 2014-03-09].

[16] R. G. Hamlet, "Testing programs with the aid of a compiler," IEEE Transactions on Software Engineering, vol. 3, no. 4, Jul. 1977, pp. 279–290.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, vol. 11, no. 4, Apr. 1978, pp. 34–41.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-m. Loingtier, and J. Irwin, "Aspect-oriented programming," in ECOOP. Springer-Verlag, 1997.

[19] "PostSharp Express," 2014, URL: http://www.postsharp.net/ [accessed: 2014-03-09].