Automatically Identifying Focal Methods under Test in Unit Test Cases

Mohammad Ghafari, Carlo Ghezzi
DeepSE Group at DEIB
Politecnico di Milano, Italy
{mohammad.ghafari|carlo.ghezzi}@polimi.it

Konstantin Rubinov
School of Computing
National University of Singapore
rubinov@comp.nus.edu.sg

***Abstract*—Modern iterative and incremental software development relies on continuous testing. The knowledge of test-to-code traceability links facilitates test-driven development and improves software evolution. Previous research identified traceability links between test cases and classes under test. Though this information is helpful, a finer granularity technique can provide more useful information beyond the knowledge of the class under test.**

In this paper, we focus on Java classes that instantiate stateful objects and propose an automated technique for precise detection of the focal methods under test in unit test cases. Focal methods represent the core of a test scenario inside a unit test case. Their main purpose is to affect an object's state that is then checked by other inspector methods whose purpose is ancillary and needs to be identified as such. Distinguishing focal from other (non-focal) methods is hard to accomplish manually.

We propose an approach to detect focal methods under test automatically. An experimental assessment with real-world software shows that our approach identifies focal methods under test in more than 85% of cases, providing a ground for precise automatic recovery of test-to-code traceability links.

***Index Terms*—Traceability, unit testing, method under test.**

I. Introduction

Agile development is founded on a seamless integration between continuous code changes and unit tests. Once developers have written the test cases, they are executed every time the production code changes to support regression testing [1]. This process requires unit test cases to be always up-to-date and makes them an important source of system documentation, especially for guiding software maintenance tasks.

Understanding the relationship between test code and source code is essential for software evolution. Test-to-code traceability links help to keep unit test cases in sync with the changes to the source code. However, the practical automated realization of test-to-code traceability has received little attention in research. For instance, XUnit[1] testing frameworks lack predefined structures for explicitly linking program source code and test cases [2], [3]. In consequence, developers have to resort to costly manual detection and maintenance of the traceability links [4].

Previous research mostly derived traceability links between test cases and classes under test [5]. Although the knowledge of the class under test (CUT) is useful for program comprehension and maintenance, test cases are composed of method invocations that play different roles in the test, and more useful information can be derived by analyzing test cases with a method-level precision. In particular, identification of method usage patterns facilitates program maintenance, while a precise determination of an intent of a test case supports test comprehension and analysis.

```
1  public void testRegisterAndRemoveProxy() {
2   // register a proxy, remove it, then try to retrieve it
3   IModel model = Model.getInstance();
4   IProxy proxy = new Proxy("sizes", new
        String[]{"7","13","21"});
5   model.registerProxy(proxy);
6   // remove the proxy
7   IProxy removedProxy = model.removeProxy("sizes");
8   // assert that we removed the appropriate proxy
9   assertEquals(removedProxy.getProxyName() , "sizes");
10  // ensure that the proxy is no longer retrievable from
        the model
11  proxy = model.retrieveProxy("sizes");
12  assertNull("Expecting proxy is null", proxy);
13 }
```

Fig. 1. Unit test case for the `Model` class

Consider an example unit test case from *PureMVC*[2] in Figure 1. In the example, `IModel` is the CUT. The knowledge of the CUT can help to identify the method under test. However, CUT information is insufficient to identify the actual (or *focal*) method under test – in the example, three methods in the test case belong to the CUT `IModel` any of which can be the method under test.

The real intent in the test case in Figure 1 is to check the `removeProxy()` method. An expert engineer can identify this with the aid of comments, test method names, and assertions. Without this knowledge or additional analysis, one might mistakenly conclude that the goal of the test is to check `registerProxy()` or `retrieveProxy()` methods. Method `registerProxy()` seems to be a relevant method to the test case, but this method is ancillary, it brings the `model` object to an appropriate state in which it is possible to invoke the `removeProxy()` method. The `retrieveProxy()` helps to inspect the state of the class under test and it is the method `removeProxy()` that causes a side-effect on the current object and is the focal method under test.

[1]XUnit is a collective name for unit testing frameworks for different programming languages.

[2]http://puremvc.org

Existing techniques mostly focus on detecting CUTs. By applying some of these techniques to the example in Figure 1 we can highlight their limitations for establishing test-to-code traceability links with a method-level precision. The approach by Van Rompaey and Demeyer relies on the relative positions of methods and assertion statements [6]. It reports the last call before each assertion statement, i.e. `removedProxy.getProxyName()` and `model.retrieveProxy()` if all assertion statements were analyzed. These are not the methods under test as we can see from the test case description. An expert developer would understand that these are inspector methods that help to verify test results. While this approach only uses a static call graph, other work has exploited dynamic analysis to derive classes that affect the result of the last assert statement [4], [7], [8]. However, these approaches report all the invoked methods of the CUT as methods under test. The approach based on name matching would not be able to precisely identify the method under test using the test name "testRegisterAndRemoveProxy", as the test name does not entail a known type [9]. Moreover, this approach is brittle and applies to test cases that strictly follow naming conventions. Lexical analysis of similarity of wording would probably give better results for this test case since it is well commented and follows proper naming conventions. But, these cannot be assumed to be always available in test code.

In this work we propose an automatic approach for detecting the *focal methods under test* (F-MUT) in unit test cases. We focus on classes that define stateful objects, where the F-MUTs are responsible for system state changes that are verified through assertions by test cases. Each unit test case may have several F-MUTs, and we define a heuristic for their automated detection.

Our proposed approach builds upon classic analysis techniques. Data flow analysis and dependency analysis of source and test code provide insights into the test case structure. The approach works at the source code level, and automatically distinguishes F-MUTs from helper and inspector methods.

This information is useful for test case comprehension through identification of logical parts of test cases. It enables techniques for code recommendation systems [10], techniques for automated debugging and code repair that rely on failing test cases to identify program fragments to be repaired and guide the techniques [11], [12], [13], [14]. Finally, automated refactoring can benefit directly from test-to-code traceability links and F-MUT information in particular. Given the test-to-code traceability, one can extend widely used refactorings like: "rename class/method" and "move class/method" for having effect on corresponding unit test cases and in vice-versa – from test cases to source code.

In summary, this work develops a technique for automated recovery of test-to-code traceability links and makes the following contributions:

- a novel heuristic for identifying F-MUTs;
- a light-weight approach based on data flow analysis that implements the heuristic for object-oriented systems in Java and test cases in JUnit format;
- a study of four real-world open source projects to demonstrate the effectiveness of the technique.

The remainder of this paper is organized as follows. The next section motivates the need for identifying focal methods in test cases automatically. Section III describes our proposed approach, and its implementation. Assessment of the approach and obtained results are reported in Section IV, followed by a summary of the related work in Section V. The paper concludes in Section VI.

II. Motivation

Unit test cases are commonly structured in three logical parts: *setup*, *execution*, and *oracle*. The *setup* part instantiates the class under test, and includes any dependencies on other objects that the unit under test will use. This part contains initial method invocations that bring the object under test into a state required for testing. The *execution* part stimulates the object under test via a method invocation, i.e., the focal method in the test case. This action is then checked with a series of inspector and `assert` statements in the *oracle* part that controls the side-effects of the focal invocation to determine whether the expected outcome is obtained.

In the context of object-oriented systems, unit test cases often test a single method [15]. Nevertheless, occasionally test cases aggregate and test several methods in a test scenario. In this case a complete *test scenario* comprises several sub-scenarios, where a *sub-scenario* contains a set of non-assert statements (setup and execution) followed by inspector and `assert` statements. That is, each sub-scenario may have a different focal method, and therefore, a test case can have more than one focal method.

A focal method belongs to the execution part of a test case and method invocations used in the oracle part often only inspect the side effect of the F-MUT. Despite clear logical differentiation of test parts each having its own purpose, in practice the parts are hardly discernible both manually and automatically. This hinders identifying F-MUTs without expert knowledge of the system. It is difficult to establish whether a method invocation belongs to the setup or execution parts of a test. Even the oracle part associated with `assert` statements may contain method invocations that may be confused with the execution part of the test case.

In practice, data flow analysis is required to distinguish among different types of method invocations in test cases. We need to distinguish F-MUTs from inspector methods serving the oracle part. For example, the test case in Figure 2 belongs to the `ArrayTableTest` class in *Guava* library.[3] There are six method invocations within this test case. The first invocation `create()` belongs to the setup part of the test case. This invocation is a helper method which initializes the `table` object and puts this object in an appropriate state for testing. The invocation at line 4, `table.eraseAll()` is the F-MUT and belongs to the execution part of the test case. In fact, the method `eraseAll()` causes a state change of

[3]http://code.google.com/p/guava-libraries

```
1 public void testEraseAll() {
2   ArrayTable<String, Integer, Character> table =
3     create("foo", 1, 'a', "bar", 1, 'b', "foo", 3, 'c');
4   table.eraseAll();
5   assertEquals(9, table.size());
6   assertNull(table.get("bar", 1));
7   assertTrue(table.containsRow("foo"));
8   assertFalse(table.containsValue('a'));
9 }
```

Fig. 2. Unit test case for the `ArrayTable` class

```
1 public void testHasCommand() {
2   IController controller = Controller.getInstance();
3   controller.registerCommand("hasCommandTest",
4     new ControllerTestCommand());
5   assertTrue(controller.hasCommand("hasCommandTest"));
6   // second sub-scenario starts here
7   controller.removeCommand("hasCommandTest");
8   assertFalse(controller.hasCommand("hasCommandTest"));
9 }
```

Fig. 3. Different sub scenarios in one unit test case

the `table` object, whose effects are later inspected using four other invocations, namely `table.size()`, `table.get()`, `table.containsRow()`, and `table.containsValue()`. Invocations at line 5-8 are *inspector methods* and contribute to the oracle part of the test by inspecting the state of the object under test affected by the the focal method `eraseAll()`, while preserving values of the class fields.

Having observed that identification of F-MUTs is not trivial and requires custom analysis, it is natural to ask whether this information is useful. Without delving deep into discussion, we present several software engineering tasks with examples where this information is beneficial or indispensable.

Software evolution and maintenance: Unclear distinction among logical parts in test code may obscure the intent of a test, which in turn can increase the maintenance cost [3]. To the best of our knowledge, except for commenting test code manually to mark each test part, the is no approach to distinguish test parts automatically.

Since the focal method invocation is associated with the execution part of a test case, the knowledge of F-MUTs in a test case should enable us to delimit test parts automatically and thus exhibit the intent of a test case.

Synthesizing API usage examples: Test cases are a source of documentation indicating the expected behavior of the system under test [3]. Developers may learn how to use APIs by studying unit test cases of the APIs [16]. Recent research focuses on extracting API usage examples from test code automatically [10]. According to our observation in a large number of real-life open source projects, a unit test case represents more useful usage information of the focal method(s) rather than other (non-focal) methods forming a test case. For instance, the test case in Figure 3 includes five method invocations, each of which contributing to a different part of the test case. Nonetheless, the test case illustrates more useful usages for the F-MUTs, `removeCommand()` and `registerCommand()`, rather than other invocations on the `Controller` class. To synthesize useful examples,

The knowledge of F-MUTs and data dependency among mutator methods in a test case helps to partition a given test case into useful usage examples. Moreover, method invocations in the oracle part of a test case are typically inspector methods whose execution has no impact on the execution of the production code. If no data dependency is associated to these methods, we can discard inspector methods from usage examples and produce more concise examples.

Software debugging: There is usually an implicit dependency among tests manifested in an overlap among a large portion of executed methods in a test suite [11]. That is, a defect that causes one unit test to fail, usually makes the other affected test cases to fail as well. For example, 80% of unit test cases of the `Model` class in *PureMVC* execute the method `model.registerProxy()`. If this method does not work properly, the test cases involving this method are likely to fail. However, there is one test case `testRegisterAndRetrieveProxy` that particularly focuses on evaluating the `model.registerProxy()` method. In fact, analyzing this test case is sufficient to catch and fix the bug.

Existing unit testing frameworks provide limited help to guide developers in bug fixing. Developers manually browse failed test cases and choose the ones that give them the most relevant debugging context for bug localization. If a single defect causes failure of multiple unit tests, the F-MUT of a test case which has the least number of mutator methods in common with other failed tests is likely to be the best starting point for debugging.

Test coverage: The type of static analysis we present in the paper can also be used to assist designers in focussing on covering mutator methods in their test suites. This is motivated by the fact that in object-oriented programs the main functionality (object behavior) is typically exposed through mutator methods that manipulate the object state, while inspector methods are ancillary and provide required data for this purpose. Therefore, reporting test coverage of the program state changes carried out by mutator methods may provide a measure of the adequacy of a test suite with respect to covering the core system functionality.

III. Identifying Focal Methods

In this paper we present an automated approach to support the developer in the identification of the F-MUTs in a unit test case. We propose a novel heuristic that is based on our experience with a large number of real-life open source projects. We have observed that dependencies between F-MUTs, class under test and assertions manifest themselves through object state changes verified in the oracle part of a test case. This observation forms an underlying intuition for our approach.

We define a **heuristic** that characterizes the intuition derived from a logical structure of a unit test case. It gives weight to data dependencies between source code and test case and states that:

The last method invocation entailing an object state change whose effect is inspected in the oracle part of a test case is a focal method under test (F-MUT).

A. Approach

Our approach leverages data flow analysis to capture essential information in test cases and source code. The phases of the approach and the input/output for each phase are shown in Figure 4. Our approach is general and applies to object-oriented systems. In this work we instantiate the approach for projects in Java and test cases in JUNIT format, the de-facto standard for unit testing Java applications.[4] Our static analysis works on Abstract Syntax Tree (AST) representation of the source code.

The approach takes as input a Java project and separates test cases from the code being tested. It then analyzes test cases to establish the scope of the analysis, i.e., which system classes are involved in testing. It analyzes the identified classes of the system (source code) to extract system dependencies and construct partial system call graph. It detects mutator and inspector methods within the classes of interest. Afterwards, the approach identifies test methods within each test case, and partitions each of them into sub-scenarios. It analyzes each individual sub-scenario to retrieve list of invoked methods, and asserted expressions. Given the information about mutator and inspector methods, invocations within each sub-scenario, as well as asserted expressions, the approach reports the focal methods in each test method (sub-scenario).

B. Analyzing source code

Constructing call graph: The approach analyzes the system source code to build the class dependency graph for classes involved in testing (the scope of the analysis). It then traverses the dependency graph in pre-order and visits each class before visiting classes that depend on it. The analysis visits every declared method within the class and derives the calling relationships between methods of the system under test. For each method, it visits all method invocations rooted from that method to gradually build a call graph. The final call graph represents the calling relationship between the methods of all the classes involved in testing.

Our analysis to build the call graph is static. This bears limitations of the approach on type resolution for interface calls and polymorphic method calls. To resolve the runtime information for the implementations of interface types and target objects in polymorphic method calls, we follow two strategies. Given a variable whose type is an interface, we look at the variable definition (called a *def-site*) to identify the exact object to which the interface refers. If there is no def-site within the method scope or the enclosing scope (in case of a global variable), in a second strategy, we resolve runtime information for interface calls using the results of a search for class declarations within the system source and test code. If found, these instances serve as over-approximated substitutes for runtime data.

[4]http://junit.org

Tracing field access information to identify mutator and inspector methods: In object-oriented languages, an object stores its states in fields (instance variables) and exposes its behavior through methods which may access (read/modify) these fields. For each class, we categorize the methods into *inspector* and *mutators*. An inspector is a side-effect free method that returns information about the state of an object, whereas a method causing object state change is a mutator method.

We apply inter-procedural forward reachability analysis to infer the effects of each method call. For classes that store their state in standard container classes such as `HashMap`, we generate a list of standard mutator methods that our approach uses to register class state modifications. We track the flow of data involving the class fields starting from leaf methods in the call chain and moving towards caller methods used in test cases. The approach iteratively traverses the call graph while recording accesses to the fields of the classes of the system. The cumulative effects of accesses to class fields represent a transitive relation and we recursively map field access sets from the callees to the callers in the call graph.

To realize whether a method is mutator or inspector, our approach first analyzes which object fields a method accesses in its method body. We detect an explicit state change in two situations: (1) a class field or any object reachable from the field is on the left hand side of an assignment statement, or (2) a class field or any object reachable from the field is a target of method invocation whose effect changes the state of the invoked object.

Manipulation of local method variables can cause object state changes as well. We detect the local variables that reference class fields using our simplified intra-procedural alias analysis. We process the AST of each method and visit all `VariableDeclarationStatement`, `Assignment`, `MethodInvocation`, and `ReturnStatement` constructs within the method body. We detect a reference to a class field from a local variable if a local variable or an object reachable from the local is initialized or assigned any of the following – (1) a class field or any object reachable from the field, (2) an invocation on a field or any object reachable from the field that returns a reference to the field (e.g., a getter method).

In addition, a method may change states of an object external to the one which the method belongs to, if the object passed as a parameter to the method. Thus, to identify the mutator methods our analysis also keeps track of changes to method parameters. For example, in the unit test case shown in Figure 5, the `controller.executeCommand(note)` method has no side effect on the `controller` object, but modifies the state of another object, `note` passed to this method as a parameter.

C. Analyzing test code

Identifying test methods: We follow test naming convention in JUnit 3 and annotations in JUnit 4 to distinguish test methods from helper ones. A test method in JUnit 3 is named with a word which begins with `test` and a test method in JUnit

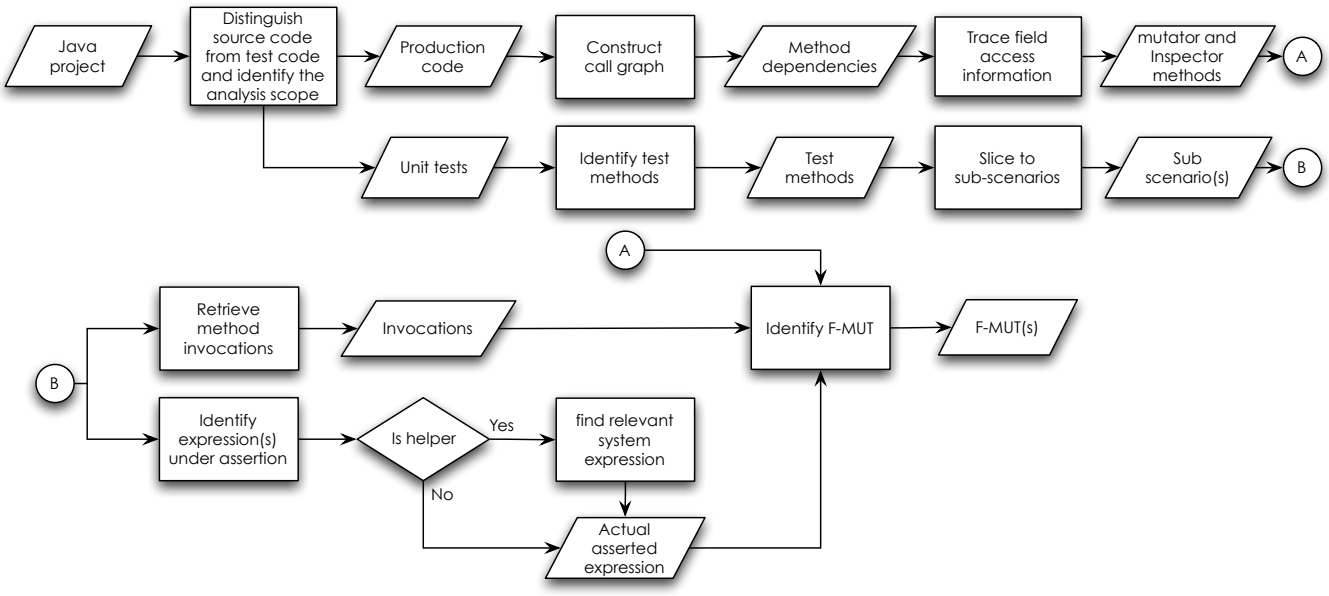

Fig. 4. The main phases of the proposed approach, and the input/output for each phase

```
1  public void testRegisterAndExecuteCommand() {
2    IController controller = Controller.getInstance();
3    controller.registerCommand("ControllerTest", new
       ControllerTestCommand());
4    ControllerTestVO vo = new ControllerTestVO(12);
5    Notification note = new
       Notification("ControllerTest", vo, null);
6    controller.executeCommand(note);
7    assertEquals(24, vo.result);
8    assertNotNull(vo);
9  }
```

Fig. 5. Unit test case highlighting an indirect object state change

```
1  public void testDefaultCharsetAppliesToTextContent() {
2    email.setHostName(strTestMailServer);
3    email.setSmtpPort(getMailServerPort());
4    email.setFrom("a@b.com");
5    email.addTo("c@d.com");
6    email.setSubject("test mail");
7    email.setCharset("ISO-8859-1");
8    email.setContent("test content", "text/plain");
9    email.buildMimeMessage();
10   final MimeMessage msg = email.getMimeMessage();
11   msg.saveChanges();
12   assertEquals("text/plain; charset=ISO-8859-1",
       msg.getContentType());
13 }
```

Fig. 6. Unit test case for the `Email` class

4 is annotated with `@Test`. In the analysis, we ignore the class dependencies belonging to testing environment (mock objects and stubs), while keeping track of all invocations belonging to the system source code and all classes external to the system.

Slicing to sub-scenarios: Next, our analysis partitions test methods in sub-scenarios w.r.t. sequences of method invocations followed by sequences of assertion statements. Each sub-scenario within a test method starts with a constructor or method invocation and ends with one or several assertion statements. To discover which mutator methods affect oracle part of a test case (assertion statements), we identify the expressions used in assertion statements in each scenario.

Identifying expression(s) under assertion: We consider all the common overloaded variants of the `assert` statements in JUnit format. If an asserted expression accesses a single variable, like the `proxy` in the unit test case in Figure 1, we find the method invocation from which the variable is assigned, i.e., `model.retrieveProxy("sizes")` in that unit test case. We use the JDT `IBinding` interface to find out whether the asserted expression contains instances of classes that belong to production code. If the declaring class of an instance used in an assertion does not originate in the project source code, we mark it as a helper. Commonly these instances belong to classes external to the system, e.g., libraries, mock objects and stubs. For instance in Figure 2, `assertEquals(9, table.size())` with asserted expression `table.size()` belongs to an object of `ArrayTable` type which is a system class. In contrast, `msg.getContentType()`, the asserted expression in another unit test case represented in Figure 6, invokes the `msg` object which belongs to `MimeMessage` class, a standard Java library.

Finding relevant (asserted) system expression: Mutator methods affect the state of the classes of the system under test and asserted expressions may check these state changes indirectly by accessing instances of external classes, rather than directly by accessing the classes of the system under test. To detect this situation, we search for declarations of external classes accessed in assertions. If the asserted expression accesses an instance of an external class (either through a field or an invocation on the external class), then we search for

an expression from which the external class is instantiated. The search continues recursively until we find an invocation or a field access on a class of the system under test that instantiates the external class. We register this invocation as the actual asserted expression. Consider again the unit test case in Figure 6 from the *Commons email* project.[5] The asserted expression `msg.getContentType()` is on the object of a helper/external class `MimeMessage`. To identify the actual asserted expression involving the class of the system under test, we trace all the statements that involve the object of this helper class and register from which class of the system under test it is assigned. The test initializes this helper object at line 10 by assigning the `email.getMimeMessage()` method to this object. This invocation is an inspector method that returns the field `Email.message`. We mark this inspector method invocation as the actual asserted expression.

Identifying F-MUT: Finally, having a set of actual asserted expressions for each sub-scenario, and the knowledge of mutator and inspector methods that our analysis discovered from the system source code, the approach reports a focal method that is a last mutator having side effect on the actual asserted expression in that sub-scenario. For example, method `email.buildMimeMessage()` in Figure 6 is the last mutator which modifies the `Email.message` field accessed in the assertion statement. We report this method as the F-MUT. If no focal method exists in a sub-scenario, then we assign it a focal method from a preceding sub-scenario.

D. Implementation

We implemented the approach in an Eclipse plugin using JAVA DEVELOPMENT TOOLS (JDT).[6] The plugin works with Java and test cases in JUNIT format. It automatically executes all the phases of the approach presented in Figure 4.

There are several structural variations of test cases that are currently not supported by the plugin. In particular, unit test cases that contain assertion statements in private methods, helper classes, and inherited methods are currently not detected by the plugin. The plugin also does not support analysis of inter-class dependencies in test cases formed through an inheritance class hierarchy. In these cases, it skips the test cases during the analysis.

IV. ASSESSMENT OF THE APPROACH

Our experimental evaluation investigates the effectiveness of our approach in identifying the F-MUTs automatically. To evaluate our approach we formulated the following research questions:

- **RQ**$_1$: Can relationships between unit tests and source code in the form of F-MUTs be identified automatically?
- **RQ**$_2$: How effective is the proposed approach to detect F-MUTs?
- **RQ**$_3$: Is the proposed approach helpful to developers for identifying F-MUTs?

[5]http://commons.apache.org/email

[6]http://www.eclipse.org/jdt

To answer the above research questions we developed a research prototype and selected test cases from real-world open source projects for analysis. The experiments have been conducted on an Intel 1.7 Core i5 CPU machine with 4GB of RAM, and Mac OS X 10.7.5 operating system. In the following we present the design of the experiments and discuss the experimentation results.

A. Design

We have selected four open-source Java projects as the context of our study. Table I presents the key characteristics of the subject programs. These are mature programs from different application domains with at least one major release. The subject programs are equipped with substantial test suites with test cases in JUnit format.

To select test cases for the analysis we first generated a raw dataset by randomly sampling 100 test cases from each of the subject systems. For the `PureMVC` project that has fewer test cases, we included all its test cases in the dataset. Then, we filtered out test cases that do not satisfy applicability criteria, for instance, test cases that do not use standard JUnit assertions. After pre-selection and filtering we obtained a dataset of 300 test cases. We inspected these test cases manually, as there is no such data before, to build a reference dataset (`ref-dataset`) which is used as an oracle in this study.[7]

For each test case, we have detected F-MUTs manually after thorough analysis of the system specification, javadocs, API usage manuals, and code comments that gave us the knowledge about the system. To improve the understanding of concepts between ourselves, we ran a pilot study comprising 5% of this set. In this phase we joined together to identify sub-scenarios and F-MUTs within each test case. As we assured ourselves that we agreed on how to extract expected information, we inspected the remaining test cases independently, while providing a short rationale for why each focal method is selected. This rationale is used for internal validation purposes. For all the studied test cases, we exchanged our results to detect potential conflicts. After we agreed on the expected results, we finalized them in the `ref-dataset`.

To evaluate the effectiveness of the approach, we applied our prototype to all the subject programs and selected test cases in the `ref-dataset`. We compared the results of the prototype with the manually identified F-MUTs.

B. Results

The first part of this study involves assessing the effectiveness of the proposed approach to identify F-MUTs, we calculate the precision and recall of our approach with respect to the results in our reference dataset. *Precision* is the number of correctly identified F-MUT versus all results returned by our prototype. *Recall* is the proportion of the results in the `ref-dataset` identified by the prototype. To take both into account, we assess the overall effectiveness of our approach using the harmonic-mean of precision and recall defined as follows:

[7]Two of the paper authors, Mohammad and Konstantin, were involved in this study.

TABLE I
QUANTITATIVE REPORT ON THE ACCURACY OF THE PROPOSED APPROACH ON THE SUBJECT PROGRAMS

Subject Programs	Source code characteristics			Experimental results		
	Version	KLoC	Test methods	Precision	Recall	H-mean
Commons Email	1.3.3	8.78	130	0.94	0.69	0.79
JGAP	3.4.4	73.96	1390	0.85	0.73	0.78
PureMVC	1.0.8	19.46	43	0.97	0.79	0.87
XStream	1.4.4	54.93	968	0.90	0.53	0.66

```
1  public void testRemovesAnItemThroughIteration() {
2    XmlMap map = new XmlMap(this.strategy);
3    map.put("guilherme", "aCuteString");
4    map.put("silveira", "anotherCuteString");
5    for (Iterator iter = map.entrySet().iterator();
       iter.hasNext();) {
6      Map.Entry entry = (Map.Entry) iter.next();
7      if (entry.getKey().equals("guilherme")) {
8        iter.remove();
9      }
10   }
11   assertFalse(map.containsKey("guilherme"));
12 }
```

Fig. 7. A test method with side effect from an external object

```
1  public void testDebugMessageIsNotNested() {
2    Exception ex = new
       CannotResolveClassException("JUnit");
3    ConversionException innerEx = new
       ConversionException("Inner", ex);
4    ConversionException outerEx = new
       ConversionException("Outer", innerEx);
5    StringTokenizer tokenizer = new
       StringTokenizer(outerEx.getMessage(), "\n\r");
6    int ends = 0;
7    while(tokenizer.hasMoreTokens()) {
8      if (tokenizer.nextToken().startsWith("----
       Debugging information ----")) {
9        ++ends;
10     }
11   }
12   assertEquals(1, ends);
13 }
```

Fig. 8. A test case with no relation between the asserted statement and system calls

$$F_{\beta=1} = \frac{2 \times precision \times recall}{precision + recall}$$

Table I summarizes the results. Our approach automatically identifies F-MUTs, in less than 2 minutes in total, and achieves a high precision and a good recall. The harmonic-mean over the four subject programs is also promising (66%–87%) which indicates that we can establish the traceability links between unit tests and source code in the form of F-MUTs in an automated manner with a high effectiveness (RQ_1 and RQ_2).

We further investigated the reasons for the lower-ranking recall score of the approach on some of the subjects. We manually inspected test cases corresponding to false negatives and false positives. The majority of these exceptions comes from the limitations of the prototype, as we could detect the F-MUTs by applying the approach presented in the paper to each test case manually.

Corner cases: In the following we briefly discuss some of the corner cases observed in our study. We focus on the results obtained for *XStream* that scored less among subject programs. *XStream* is a Java library to serialize objects from Java to XML and back. It consists of 968 test cases examining the behavior of over 54K lines of code.

An invocation of a standard Java class outside the system source code can affect the asserted statement. Consider a test case in Figure 7. An external call `iter.remove()` at line 8 affects the oracle part of this test case. An invocation of `map.put()` at line 4 is the last mutator, but the test case focuses on deletion of an item through iteration. In fact, the `iter` object at line 5 holds a reference to the entries in `XmlMap`. That is, the invocation of `remove()` method on the `iter` object at line 8 also removes an item from `XmlMap`. Our prototype does not consider such an implicit dependency between test code and external classes, and negatively reports `XmlMap.put()` as the F-MUT in this test case.

Also, none of the invoked methods in a test case may have computational relation with the oracle part. We report the last mutator as a F-MUT even if we see no relation with the assertion. Figure 8 shows `testDebugMessageIsNotNested` in `ConversionExceptionTest`. The asserted statement in this test case is the integer `ends` at line 9 whose value is only control dependent with `tokenizer`, a helper object in the test case. The approach ignores the last system call `outerEx.getMessage()` at line 5 which is inspector and successfully reports the `ConversionException()` constructor as the F-MUT. Our analysis does not consider control dependencies. We may improve the approach by considering the control dependencies with the oracle part.

The applicability of the approach depends on project guidelines and developer discipline. Our analysis treats customized assertions like the standard JUnit assertions. We skip analyzing body of these assertions, as according to our observation, it is more effective to look for a F-MUT in the test method body rather than in an assertion method body. For example, all test cases in `AliasTest` use the same custom assertion `assertBothWays` to evaluate their test cases. Figure 9 demonstrates two of these test cases. If we look at the implementation of this assertion to find out the F-MUT, we will mistakenly report the same focal method for both test cases, while each one has a different scenario and a different F-MUT.

Useful cases: In Section II we have shown several software engineering tasks that benefit from the knowledge of F-MUTs. In the following, we discuss some cases where the

```
public void testIdentityForFields() {
  Software software = new Software("walness",
     "xstream");
  xstream.alias("software", Software.class);
  xstream.aliasField("name", Software.class, "name");
  xstream.aliasField("vendor", Software.class,
     "vendor");
  String xml = ""
     + "<software>\n"
     + "  <vendor>walness</vendor>\n"
     + "  <name>xstream</name>\n"
     + "</software>";
  assertBothWays(software, xml);
}
public void testForFieldAsAttribute() {
  Software software = new Software("walness",
     "xstream");
  xstream.alias("software", Software.class);
  xstream.useAttributeFor(String.class);
  xstream.aliasAttribute("id", "name");
  String xml = "<software vendor=\"walness\"
     id=\"xstream\"/>";
  assertBothWays(software, xml);
}
```

Fig. 9. The same custom assertions used in test cases with different scenarios

```
public void testUnmarshalsObjectFromXml() {
  String xml =
    "<x>" +
    "  <aStr>joe</aStr>" +
    "  <anInt>8</anInt>" +
    "  <innerObj>" +
    "    <yField>walnes</yField>" +
    "  </innerObj>" +
    "</x>";
  X x = (X) xstream.fromXML(xml);
  assertEquals("joe", x.aStr);
  assertEquals(8, x.anInt);
  assertEquals("walnes", x.innerObj.yField);
}
```

Fig. 10. A complex test to detect F-MUT

approach was more helpful in reducing manual effort to identify the F-MUTs. We gather this information from the examination of our experience during manual analysis of test cases and a similar study with few external developers proficient in Java and JUnit testing.

The number of statements required to be inspected to identify F-MUTs varies due to the source code structure and the degree of coupling between objects. In our experience, the approach is particularly useful in cases where one needs to inspect more than three different locations (in different classes and methods) in the source code. Without systematic tracing one can lose the clue of the context and of what has been inspected before. Consider the test case in Figure 10. Though this is a trivial example, in practice to resolve the effect of `xstream.fromXML()`, we needed to inspect the behavior of six other methods in the source code, including an interface call, whose result is passed to another interface call.

Despite being useful in many cases, we observed cases when the approach applies, but does not yield useful results. This includes trivial test scenarios with a single inspector method in a test case, such as, for example, unit test case `testCanConvert` shown in Figure 11 that invokes a single `canConvert()` method. Our approach reports a constructor as F-MUT in such a test case.

Some test cases consist of a few non-assertion statements and many assertion statements. Figure 12 demonstrates one of the test methods of this kind within `BasicTypesTest`. These test cases usually examine some basic or simple functionality in the target API, and an assertion statement itself establishes a test scenario. Identification of F-MUTs for such test cases provides little useful results.

Manual inspection: To further investigate our observations, we hired a small group of developers (undergraduate students) not familiar with the subject programs to identify F-MUTs for 50 test cases each manually. These test cases were chosen at random from the set of test cases successfully identified by the prototype, amongst 300 test cases used for evaluation of the effectiveness of the approach (reported in Table I). The task was integrated as an assignment to their Software Evolution course. We taught the participants all the necessary information for the completion of the task. They were allowed to browse the source code, documentation, and use any of the features from the standard installation of Eclipse to perform this task.

The results of the study indicate that for the subjects with initial familiarity with the system under test, the manual analysis of test-to-code traceability to detect focal methods in 50 test cases is laborious and takes considerable effort, 35 minutes on average. Many participants reported tracing data dependencies between different fields and objects challenging, especially when they needed to consider aliasing. As we ourself also experienced, the task becomes more complex and arduous when the inspection involves polymorphic method calls. In case of inspecting an interface call one needs to search the entire source code for the implementation of the interface. Furthermore, identifying dependencies to helper classes to find actual asserted expressions is non-trivial. In fact, distinguishing helper classes from classes of the system under test without prior knowledge of the system is error-prone. For example, some participants misclassified helper method `msg.getContentType()` shown in Figure 6 due to naming similarity with methods of the system object of type `Email`.

Besides saving a significant amount of manual effort, the prototype implementation of the approach highlighted 12% of F-MUTs missed or mistakenly reported by the participants of the study. The results of the study with the external developers corroborate our experience during manual inspection of the test cases and provide initial evidence of the approach being helpful and advantageous to developers through automation (RQ_3).

C. Threats to Validity

We note several limitations and threats to validity of the results pertinent to the presented approach. We mitigate the risks to the external validity and the generalization of the results by selecting real world systems from different application domains with manually generated test cases. The study involved a randomly selected sample of test cases from the different systems.

```
public class URIConverter implements Converter {
  /* implementation omitted to save space */
  @Override
  public boolean canConvert(Class type) {
    return URI.class == type;
  }
}
public void testCanConvert() {
  final Class type = URI.class;
  final URIConverter instance = new URIConverter();
  final boolean expResult = true;
  final boolean result = instance.canConvert(type);
  assertEquals(expResult, result);
}
```

Fig. 11. A simple test with a single observer method

```
public void testNegativeIntegersInHex() {
  assertEquals(new Byte((byte)-1),
     xstream.fromXML("<byte>0xFF</byte>"));
  assertEquals(new Short((short)-1),
     xstream.fromXML("<short>0xFFFF</short>"));
  assertEquals(new Integer(-1),
     xstream.fromXML("<int>0xFFFFFFFF</int>"));
  assertEquals(new Long(-1),
     xstream.fromXML("<long>0xFFFFFFFFFFFFFFFF</long>"));
}
```

Fig. 12. A test with repeated assertions

Threats to internal validity might arise from the process used in our empirical study. We used statistical methods to evaluate the result of the experiments where results could have been affected by a randomness in the test case selection. The accuracy of the results used to evaluate our approach affect the results achieved. We did not have access to the original program developers to indicate the focal methods in each test case. For this reason, we familiarized ourselves with the project documentation and details of the source and test code, and cross-checked the results. The fact that the approach is validated against manual analysis that is performed by two of the authors is a threat to construct validity through potential bias in experimenter expectancies. Further, to estimate the manual effort to find F-MUTs we have hired a small group of undergraduate students who may not be a representative of industrial developers targeted by our technique.

Our approach shares inherent limitations with other static analysis techniques that are generally not sound. According to our findings in this study, many of the false negatives are due to implementation problem, but not the limitations of the approach. We plan to develop a solid implementation of the proposed approach to utilize better points-to analysis for type resolution and more precise call graph construction, for instance, using Spark [17].

V. Related work

This section reviews relevant work to test case comprehension and recovery of traceability links between the source code and test cases.

From the industrial point of view, today's integrated development environments provide minimal help to the developers to link unit test cases with code under test. The Eclipse IDE allows Java developers to associate the class under test to each unit test using the specialized test case wizard.[8] Eclipse also provides a "Referring-Tests"[9] search command that retrieves all JUnit tests that refer to a selected type.

Recent research work targets the test-to-code traceability link recovery. A tested class usually depends on other helper classes that makes the test case analysis difficult, and recovering test-case traceability link a complex problem. Bruntink et al. suggest using "cascaded test suites", where a test case of a complex class uses the test cases of its required classes to organize a complex test scenario [18]. Bouillon et al. leverages a failed test case to narrow an error location in the source code [19]. To link the tests with source code, they build the static call graph of each test method and annotate each test with a list of methods which may be invoked from the test. Sneed uses timestamps to restore the link between test cases and source code for selective regression testing [20]. He relates the time at which the methods were executed to the execution time of the test cases.

Van Rompaey and Demeyer compare several traceability resolution strategies to link test cases and the units under test [6]. We have encountered some of these strategies in the Introduction section of this paper. Naming convention (NC) matches production and test files in the source code by removing the string "Test" from the name of the test case. This strategy falls short if the test name does not contain the name of the unit under test or does not entail a known type. Another strategy analyzes call behavior before assertion statements (last call before assert - LCBA) and presumes that a test case calls a method on the unit under test right before the assertion statement. It exploits the static call graph to identify the last class called before an assert statement. This strategy fails when, right before the assert statement, there is a call to a class other than the tested class [8]. A strategy based on Lexical Analysis builds upon an assumption that developers use similar vocabulary to write the source code of a test case and the corresponding unit under test. Latent Semantic Indexing, an information retrieval technique, calculated this similarity. However, their study shows that a significant amount of vocabulary in a test case does not repeat in the unit under test. Finally, a version log mining strategy builds upon an assumption that test cases and their corresponding unit under test co-evolve together throughout time. This strategy bears a risk to wrongly identify production code that change frequently as the unit under test.

Van Rompaey and Demeyer report that among these strategies, NC is the most accurate, and LCBA has a high applicability. These strategies and related approaches have important limitations when applied to detect F-MUTs. They are unlikely to detect focal methods in test cases with several sub-scenarios using test naming conventions. The LCBA approach does not distinguish calls to helper classes from the actual

[8]http://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/gettingStarted/qs-junit.htm

[9]http://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/reference/ref-menu-search.htm

classes under test. Furthermore, instead of identifying F-MUTs, it may mistakenly identify inspector methods as MUT when such methods help to verify test results before the assertion.

Qusef et al. propose to use data flow analysis to circumvent the limitations of these strategies [8]. They apply reachability analysis and exploit data dependence to identify a set of classes that affect the result of the last assertion statement in each unit test. This analysis, however, does not consider inter-procedural flow, inheritance, and aliasing. SCOTCH is an improvement over this work – a technique based on dynamic slicing to restore test case traceability links [7]. The set of identified classes by dynamic slicing is an overestimate of the set of classes actually tested in a test case. In fact, a slice will contain all the helper classes used in a test case as well. In a recent work, the same authors employ another filtering strategy based on name similarity to enhance the accuracy of their earlier approach [4]. These approaches rely on a "stop-class-list" to hold the names of the classes to be considered as helper class in the analysis, however, these classes have to be manually identified prior to the analysis. Our study of the real world unit test cases has shown that considering the last assertion statement may suffice to identify the class under test, however it may not suffice to detect F-MUTs correctly.

Galli et al. provide initial evidence that a single method is most often the unit under test in object oriented programs [15]. Nevertheless, to date there is a scant work on automatically identifying methods under test. Ying et al. propose a call graph filtering approach to detect methods that are probably irrelevant during program investigation [21]. According to their findings, methods closer to the leaf of a call graph, as well as those with a small number of callees are unlikely to contribute to the understanding of the application logic. They use this approach to eliminate irrelevant methods from the set of methods that can be invoked, transitively, from a JUnit test case. This heuristic highlights the setup part of a test and misses to detect a focal method which is called right before an assertion in a test. In addition, it fails to retrieve relevant invocations in a test with multiple sub scenarios.

VI. Conclusions

We present an automated approach for recovering traceability information between source code and test cases for object-oriented software. The main contribution of this work is the idea that the relationship between the source and test code can be established on the method level by detecting the focal methods under test (F-MUTs) in unit test cases. F-MUTs are the methods that are responsible for system state changes that are verified through assertions by test cases. The knowledge of F-MUTs facilitates program maintenance and supports test comprehension and analysis.

We report experimental results obtained with the prototype implementation of the approach that identifies F-MUTs in unit test cases. The approach is fully automated and applies to object-oriented software. The results obtained with four real world Java projects are encouraging. The approach precisely identifies F-MUTs in more than 85% of cases.

This research is the first step towards establishing a precise method level traceability between source code and test cases. In this work we have focussed on links between source code and unit test cases. Similarly, other types of test cases, such as integration ones, use method calls as atoms to construct test cases from. Identifying F-MUTs in these types of test cases needs a dedicated study that we plan to investigate in our future work.

References

[1] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 51–56, 1998.

[2] P. Hamill, *Unit Test Frameworks*. O'Reilly, 2004.

[3] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ: Addison-Wesley, May 2007.

[4] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *J. Syst. Softw.*, vol. 88, pp. 147–168, Feb. 2014.

[5] R. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *Reliability, IEEE Transactions on*, vol. 63, no. 4, pp. 913–926, Dec 2014.

[6] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 209–218, 2009.

[7] A. Qusef, G. Bavota, R. Oliveto, A. DeLucia, and D. Binkley, "Scotch: Improving test-to-code traceability using slicing and conceptual coupling," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, Williamsburg, VA, USA, 2011, pp. 63–72.

[8] A. Qusef, R. Oliveto, and A. DeLucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, Timisoara, Romania, 2010, pp. 1–10.

[9] P. Marschall, "Detecting the methods under test in java," in *University of Bern*, 2005.

[10] M. Ghafari, C. Ghezzi, A. Mocci, and G. Tamburrelli, "Mining unit tests for code recommendation," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC '14, 2014.

[11] M. Galli, M. Lanza, O. Nierstrasz, and R. Wuyts, "Ordering broken unit tests for focused debugging," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04, 2004.

[12] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 121–130.

[13] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An approach to debugging evolving programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, pp. 19:1–19:29, Jul. 2012.

[14] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013.

[15] M. Gälli, M. Lanza, and O. Nierstrasz, "Towards a taxonomy of unit tests," in *Proceedings of the 13th International European Smalltalk Conference*, Brussels, Belgium, 2005.

[16] S. Nasehi and F. Maurer, "Unit tests as api usage examples," in *Software Maintenance (ICSM), IEEE International Conference on*, Sept 2010.

[17] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03, 2003, pp. 153–169.

[18] M. Bruntink and A. van Deursen, "Predicting class testability using object-oriented metrics," in *Proceedings of the 4th International Workshop Source Code Analysis and Manipulation*, Montreal, Canada, 2004.

[19] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: A framework for associating failed unit tests with potential programming errors," in *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming*, Como, Italy, 2007.

[20] H. M. Sneed, "Reverse engineering of test cases for selective regression testing," in *Proceedings of the 8th Working Conference on Software Maintenance and Reengineering*, 2004, p. 69.

[21] A. T. T. Ying and P. L. Tarr, "Filtering out methods you wish you hadn't navigated," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07, 2007, pp. 11–15.