AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation

Nadia Alshahwan, Yue Jia, Kiran Lakhotia
University College London, UK

Gordon Fraser, David Shuler
Saarland University, Germany

Paolo Tonella
Fondazione Bruno Kessler, Italy

Abstract—**During testing, there are several reasons to exclude some of the components used by the unit under test, such as: (1) the component affects the state of the world in an irreversible way; (2) the component is not accessible for testing purposes (e.g., a web service); (3) the component introduces a major performance degradation to the testing phase (e.g., due to long computations); (4) it is hard (i.e., statistically unlikely) to obtain the output required by the test from the component. In such cases, we replace the component with a mock one. In this paper, we integrate the synthesis of mock components with the generation of test cases for the current testing goal (e.g., coverage). To avoid the generation of meaningless data, which may lead to assertion violation not related to bugs, we include a weak mock postcondition. We consider ways to automatically synthesize such postcondition. We empirically evaluate the quality of the mocks generated by our approach, as well as the benefits mocks introduce in terms of improved coverage and improved performance of the test case generator.**

Keywords: test case generation; code analysis; automated software testing.

I. INTRODUCTION

Code-based test generation has made tremendous progress in the recent past: Today, modern systems are able to generate inputs that drive execution to almost any point in the control flow. Automation, however, reaches its limits when it comes to controlling the test's environment: For example, if the tested code accesses an external component such as a service or a database, then this component needs to be controlled by the test as well in order to prevent unwanted side-effects like data loss.

A common solution in test generation is to create a stub version (mock) of the component that is difficult to control. Such a mock object provides the same interface as the component it represents, but returns predefined values on method calls. In this paper, we propose an approach that automatically generates such mock objects, helping to drive test generation towards its goal in cases where automatic generation is difficult or impossible otherwise. The solution applied is to map the mocked behavior to the input space of the test generation problem: For each method call on the mock object in the test case we need to find an appropriate value that the mock returns during execution.

A main difficulty in this approach is that within the scope of a test case the mocked component should behave similarly to the real component. If this is not the case then that is problematic for two reasons: First, it can lead to a false sense of confidence in the correctness, because a coverage goal may be satisfied in a way that cannot occur in practice. Second, it can lead to an incorrect test failure (false positive), i.e., a failure that is not caused by a real bug but by an invalid mocked behavior.

To overcome this problem, we investigate different methods to infer knowledge about the real behavior of the mocked instance. Such knowledge can be approximated from existing execution traces or dynamic invariants on the component's behavior derived from these traces, or it can be derived precisely by solving path constraints derived from the mocked component.

Automated mocking has been addressed in the past: Saff et al. [2] take an existing test case and try to replace some of the used objects with automatically generated mocks that simulate the same behavior, thus aiming to improve performance and making it easier to isolate and detect bugs. Tillmann and Schulte [3] implement mock objects that represent symbolic variables, which the test generator can interpret as inputs. A main objective of this approach is to test for robustness, i.e., unlikely (but admissible) test inputs are desired. Automated mock generation is also related to environment generation in software model checking: Tkachuk et al. [4] simulate a component's environment based on user specified assumptions and additional information derived with different types of static analysis.

In detail, the contributions of this paper are as follows:

- Approximation of method postconditions representing realistic behavior based on traces and dynamic invariants (Section II).

Dagstuhl Seminar Proceedings 10111
Practical Software Testing : Tool Automation and Human Factors
http://drops.dagstuhl.de/opus/volltexte/2010/2618

- Automatic derivation of mock objects based on method postconditions and test goals such as coverage (Section II).
- Derivation of method postconditions based on symbolic execution of path constraints (Section IIA).
- Evaluation of the derived mock objects in terms of their quality, i.e., false negatives / positives (Section IIIA)
- Evaluation of the improvements achieved with auto-mocking in terms of coverage increase and performance gains (Section IIIB).

II. Mock synthesis

Consider for example the code shown in *Figure 1* that does a hotel reservation for a specific date range. Internally it uses another service via a HotelConnection. For testing however, we cannot use this service since this would result in many unintentional reservations. Thus, the HotelConnection needs to be mocked.

```
testMe(Date startDate, Date endDate) {
  conn = new HotelConnection();
  try {
    resp = conn.makeReservation(startDate,
           endDate);
    reserved = false;
    do {
       if (resp.available()) {
          assert(endDate > startDate);
          // money processing
       } else {
          // ask user for different dates
       }
    } while (!reserved);
  } catch(WrongResDatesException e) {
    // error handling code
  }
}
```

Figure 1. Running example

For the test generation this means that, in addition to the method inputs, sensible return values for the mocked methods have to be generated with respect to the testing goal. In our example this would be a start and end date as input to the method and either a response object or a `WrongResDatesException` exception as return value of the `makeReservation` method.

Furthermore, the mock has to have the same interface as the original object, but it does not need to preserve all its semantics. However, the returned values must not cause any exceptional behavior. Thus, they have to satisfy *weak postconditions*, which are relaxed postconditions of the method that is mocked. These weak postconditions can either be specified manually or automatically. One way to obtain postconditions automatically is to trace the program (or the object to be mocked) when it is regularly used and infer dynamic invariants from the traced data (e.g. by using Daikon [1]). These invariants can then be used as additional constraints when generating return values for the mocked methods. Another way to obtain or refine these invariants is to use an algorithm that is based on symbolic execution and is described below.

A. *A symbolic execution algorithm to infer an approximate mock postcondition*

The pseudocode for the symbolic execution algorithm used to infer an approximate mock postcondition is provided at the end of the paper. According to the algorithm, when required by the test generator, a mock value is produced, initially with an empty mock postcondition. When an assertion is reached which fails, the path condition produced by symbolic execution along the failing path is considered. The constraints on mock variables in the path condition are passed to a solver. If the constraints are satisfiable, they are added to the mock postcondition. Otherwise we have a "likely" bug, i.e., an assertion violation caused by the program, not by a too permissive mock postcondition. The bug is not certain since the mock postcondition is an approximate one, hence it may still generate meaningless values. It should be noticed that even when the constraints are satisfiable we may have a problem, in that we may miss a bug and generate a wrong (too restrictive) mock postcondition . So, in general we may have both *false positives* (reported bugs which are not real bugs) and *false negatives* (missed bugs).

Let us consider the running example and assume we want to mock the function *conn.makeReservation*. Further let us assume that the assertion *assert(endDate > startDate)* is violated during the execution of *testMe*. The symbolic path condition corresponding to the execution (including the assertion) is:

!thrown(WrongResDatesException) && resp.available() && (endDate > startDate)

After negating it and keeping only mock variables, we synthesize the following mock postcondition:

thrown(WrongResDatesException || !resp.available()

This postcondition expresses quite precisely the mock values that must be generated in this case to avoid raising the exception. Either the reservation service replies by raising the exception *WrongResDatesException* or it replies that the reservation is unavailable. Since the path condition includes symbolic variables which are dependent on our mock function/object, it could be that the mock function we created is (partly) responsible for the assertion violation, and therefore indicates a false alarm. Thus the goal is to try and refine the post condition of our mock function in order to avoid such false alarms.

A more precise mock postcondition is obtained if mock input variables are included. In the running example, we get:

thrown(WrongResDatesException) || *!resp.available()* ||
(endDate <= startDate)

This mock postcondition specifies that we need to either throw an exception or reply that reservation is unavailable only if the condition *(endDate <= startDate)* is false. Otherwise, such a condition on the mock inputs is enough to make the postcondition true.

III. Empirical study design

A. *Assessment of the quality of the syntesized mocks*

We intend to investigate the following research questions:

RQ1*: How good are the mocked data generated using our approach?*

> **RQ1.1***: Can our automatically generated mocked data find all bugs that are found by the manually generated mock, with exact mock postcondition? Is the approximate mock postcondition masking any real bug?*
>
> **RQ1.2***: How many bugs found by the automatically generated mock are not real bugs, being side effects of the mock postcondition approximation?*

RQ1 is trying to assess the quality of our automatically generated mocked data. In order to answer RQ1, we compare the bugs (assertion violations) found by our mocked data with the ones found by the manually generated mock, which contains the exact mock postcondition. We split RQ1 into two. In RQ1.1, we investigate the *false negatives*, i.e., the bugs missed by the automatic mock because its postcondition does not permit the generation of data that reveal them. In RQ1.2, we analyze bugs that are originated because the data generated by the automatically synthesized mock do not comply with the true mock postcondition, hence giving raise to assertion violations. These are *false positives* of our technique.

The metrics we use to address RQ1.1 and RQ1.2 are the number of false negatives and false positives respectively. The former are obtained by manually defining the exact mock postcondition and then determining the number of assertion violations raised with such mock, which cannot be raised when the approximate postcondition is used. For the latter we count the number of false alarms reported by the synthesized mock, i.e., assertion violations which are not bugs. We repeat this assessment for all alternative mock synthesis techniques and technique variants described in this paper.

B. *Assessment of coverage and performance improvement*

After we evaluated the quality of the mock objects our technique generates, we need to assess how effectively these objects serve the goal for which they were created.

Mock objects can be created for a variety of reasons. For example when the return value of a certain object (e.g. service, database query) affects branching statements in the system we want to test. The objective in this case is to improve coverage. Mock objects are also created when an object has long execution times (e.g. large file upload). The goal here is to improve performance during the test data generation process.

To evaluate our mock object generation technique we have to prove that the objectives we are trying to achieve are being met when they are relevant. We designed an experiment to answer the following research questions:

RQ2: *Does the use of automatically synthesized mocks simplify test generation for coverage adequacy?*

To answer this research question, we need to compare the levels of coverage achieved when using real objects and mocked objects. The applications we need to select for this experiment have to possess certain characteristics. They need to use values returned from database queries or calls to external services in control flow structures. We will use the same test data generation technique on those applications using the real objects and then repeat the experiment using the mock objects generated by AUTOMOCK and compare coverage.

RQ3: *Does the use of automatically synthesized mocks improve the performance of testing?*

To answer this research question, we need to compare execution times needed to reach a certain level of coverage with both real and mocked objects. The level of coverage is specified to provide a baseline when comparing the two techniques. The applications selected for this experiment need to have objects that require long execution times to be able to better observe the benefit.

IV. Acknowledgements

This paper would not have come to light without the Dagstuhl seminar 10111, where the core ideas have been formulated and discussed.

References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” IEEE Trans. Soft. Eng., vol. 27, n. 2, pp. 99-123, 2001.

[2] Saff, D., Artzi, S., Perkins, J. H., and Ernst, M. D. Automatic test factoring for java. In Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (Long Beach, CA, USA, November 07 - 11, 2005). ASE '05. ACM, New York, NY, 114-123, 2005.

[3] Tillmann, N. and Schulte, W. Mock-object generation with behavior. In Proceedings of the 21st IEEE/ACM international Conference on Automated Software Engineering (September 18 - 22, 2006). Automated Software Engineering. IEEE Computer Society, Washington, DC, 365-368., 2006.

[4] Automated Environment Generation for Software Model Checking, Oksana Tkachuk, Matthew Dwyer, Corina S. Pasareanu, in Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.

```
1. S = empty set
2. **foreach** uninterpreted function call *i* do
3.  analyse the interface of *i*, and let the return type of *i* be *t*
4.  add a mock function $m_f$ with the same interface as *i*
5.  add a variable $m_i$ of type *t* to the input domain of the SUT
6.  replace the call to *i* with a call to  $m_f$
7.  add $m_i$ to S
8. **end foreach**

9.  **while** not done do
10.    execute the SUT dynamically and symbolically in parallel with an input *I*
11.   form a path condition pc from the result of the symbolic execution
12.   **if** the dynamic execution raises an assertion violation in the SUT then
13.      add the assertion *a* to the path condition pc, describing the violation which raised the exception
14.      **if** the symbolic variables in *a* are (transitively) dependent on a mock variable $m_i$ then
15.         replace every symbolic variable in pc with its concrete value, except those in S
16.         **if** pc is satisfiable then
17.            add the simplified constraints over $m_i$ as a postcondition to the mock function
18.         **else**
19.            raise likely bug exception
20.         **endif**
21.      **else**
22.         raise bug
23.      **endif**
24.  **endif**
25.**end while**
```

Algorithm 1. Automated synthesis of mock postcondition using symbolic execution