Improving Integration Testing of Web Service by Propagating Symbolic Constraint Test Artifacts Spanning Multiple Software Projects

Andreas Fuchs ✦ Vincent von Hof ✦

✦ University of Münster

Abstract

Software testing is a process involving multiple consecutive testing phases. Once software-units and -components have been tested, the integration of software-components itself must be tested. This discipline of Integration Testing involves multiple systems and resources (human and hardware) and the effort for successfully implementing an integration testing scheme is easily underestimated. Yet errors detected in this phase of the testing process, may require fixes in the individual software components. Trivially, if many such problems appear, the testing process is delayed and in the worst case, the roll-out of the system has to be rescheduled. Focusing on the widely adopted Java Enterprise Edition platform for web-services, this paper presents an approach to move the detection of possible integration errors from the integration testing phase to the earlier and simpler unit testing phase, by generating and collecting constraints relating to web-service calls on the client side, in order to improve the overall time required for the testing process.

1 Introduction

Recently, it was demonstrated how to automatically generate unit tests for the business logic of Java applications that include web services [11]. More specifically, a minimal set of test cases is generated which cover the control-flow graph of that application, e.g. for the `Checkout Counter` employing a `Library Service` as depicted in Figure 1. If a path depends on the response of a web-service request, an executable web service with the desired behavior is generated as well. To achieve this, the application is executed `symbolically` in order to systematically explore all paths through the application. During this execution, a system of constraints with symbolic variables for both the application's input is created, as well as the web service responses. Once the symbolic execution of one path is finished, a constraint solver is used to generate concrete values for every symbolic variable constellation. In the end, those values guide the generation of executable test cases and a set of web services which correspond to the expected behavior. Thus, the generated service acts as a `stub`, which always returns the same result regardless of change in input, or as a `mock`, where the return value is based on method-internal logic.

It is important to note, that the web services generated by this approach only mimics the model based on the *client's* knowledge of the server, since the server source code is a black box to the client during generation time. While this has its own merit, integration testing is still required to ascertain flawless component integration. In software engineering systems are often designed in a waterfall model approach. After writing the code, unit tests are written (or generated and validated). Afterwards the individual software components are tested regarding their *integration* with one another. Since more than two components may exist in a system, the integration between each and every one of them needs to be tested. This may occur for the individual component pairs, larger sets or in a "big bang" approach. For large systems with several integrated components, there may exists a large amount of these tests. Larger software companies, e.g. Google LLC, restrict themselves to only write integration tests for "large, high priority" features [21]. Integration tests are often run during off-hours of development. Problems that occur, can only be fixed the next day. If problems relate to two interconnected components that both require correction, a problem with the fixes is only detected upon the next integration testrun—another round of fixes may be required.

We propose a system, that utilized the constraints generated during the web service automated test case generation phase from one *(client)* component, during the test case generation phase of a *(server)* component. The resulting *unit* test case suite is then to be inter-

DOI reference number: 10.18293/SEKE2018-128

preted as follow: If a unit test case of this specific test suite fails upon execution of the unit testing phase, it hints at a problem that might occur **later** during integration testing. By moving the detection time of potential problems from the integration- to the unit-testing-phase, problems that may occur due to changes that were made to the server component are uncovered earlier. Then, testers may run a small subset of the integration test only involving these involved components, thus reducing the total amount of time required to remedy the problem.

To summarize, this paper provides the following contributions:

- The detection of integration errors is *shifted* to *earlier* testing phases.
- A system for reusing and sharing of constraint information is presented.
- We utilize the constraint sharing system to propagate constraints to the automatic test case generation tool and generate a distinct integration test-suite for the web-service software component.

The rest of the paper is organized as follows. Section 2 gives an overview of the phases of software testing and specifically about the relation . In Section 3 we describe how the generated constraints can be gathered and propose an infrastructure schema for exchanging constraint data. In Section 4, we present related work that is comparable to our approach, and we conclude this paper in Section 5.

2 Integration Testing in the Development Process

The V-Model is an abstract representation of the process used for software development [9]. It can be considered an extension of the waterfall model [9]. In Figure 2, the left hand side of the model represents steps necessary to define the goal software system. Based upon the requirement specification, the function and non-functional specification are developed. This model omits, that the two lowest phases occur a number of times for a given requirement specification, depending on how many systems are involved.

Afterwards, a technical specification is defined, and the individual programs are specified. After the actual development of the software artifacts, they are

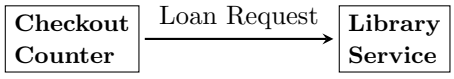

Figure 1: An overview of two systems that exchange data.

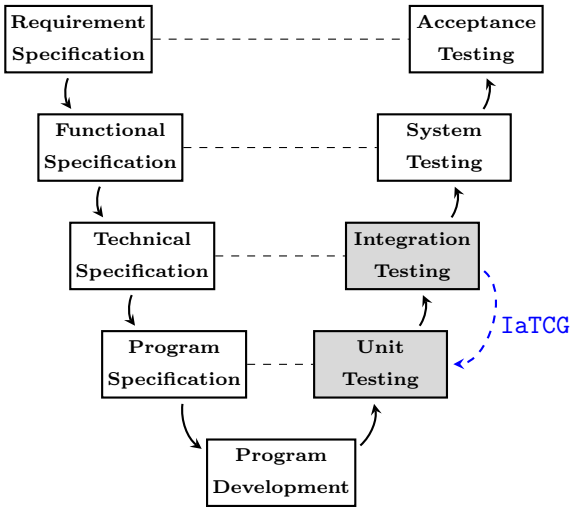

Figure 2: A Software Development V-Model with Integration aware Test Case Generation (IaTCG).

```
1
2  public class CheckoutCounter {
3   private final long maxBalance = 50L;
4
5   private Loan createLoan(Integer userId, List<
        String> books) {
6    User user = library.getUser(userId);
7    long balance = library.getBalance(user);
8    if (balance <= maxBalance) {
9     Loan approvedLoan = library.requestLoan(
          user, books);
10    return approvedLoan;
11   }
12   throw new InsufficientFundsException();
13   return null;
14  }
15  ...
16 }
```

Figure 3: Checkout Counter Client.

unit tested to ascertain whether or not they meet the program testing goals. At this step, automated test case generation may be utilized to develop the unit test cases. If the individual programs adhere to the goals, the *integration* of the programs according to the technical specification is checked. The goal of the paper, relates to this and the aforementioned step *unit-testing*.

Integration testing of the individual software components may occur in a *top-down*, *bottom-up* or *big-bang* fashion, among others [5]. Assuming, that there is some hierarchy in the way modules are integrated, we may call modules, that require none of the other modules to offer their functionality as basic. For bottom-up and for top-down testing modules will be assigned to groups. In the case of bottom-up integration testing, basic modules make up level 0. Level 1 makes up groups of modules, that require some of the level

```
1
2  public class LibraryService {
3   private final long maxBalance = 50L;
4
5   @WebMethod
6   public Loan requestLoan(User user, List<
        String> books) {
7    long balance = getBalance(user);
8    if (balance >= maxBalance) {
9     return null;
10   }
11   Loan loan = createLoan(user, books);
12   return loan.getId();
13  }
14  ...
15 }
```

Figure 4: Initial Library Service Server.

```
1
2  public class LibraryServiceV2 {
3
4   @WebMethod
5   public Loan requestLoan(User user, List<
        String> books){
6    long maxBalance = getMaxBalance(user);
7    long balance = getBalance(user);
8    if (balance >= maxBalance) {
9     return null;
10   }
11   Loan loan = createLoan(user, books);
12   return loan.getId();
13  }
14  ...
15 }
```

Figure 5: Second Version of Library Service Server.

0 modules to be present upon execution. This holds for all other levels $i + 1$, $i \in N$. Abiding by this process, the modules and their dependencies are executed concretely during this stage.

Using the top-down approach, the lower level dependencies are not present upon integration testing. The functionality of the lower-level dependencies is provided by a mock service instead, that mimic the behavior of the lower-level dependencies. This is also a viable approach,as the mocks can be designed in such a way, that they are very close in behavior to the functionality checked by test cases used in the lower-level dependencies interfaces. A mismatch of mock behavior and lower-level interface test behavior is problematic. Of course, bottom-up and top-down approach can be combined into a mix to decrease integration testing time, depending on the tree structure of the dependency hierarchy tree.

Finally, integration testing can occur by starting all modules at the same time and testing the modules in arbitrary order or even at once. However, first, depending on the amount of modules, it may be problematic to create an runtime environment as that matches the hardware of the production system. Second, if the tests are not run at the same time, the integration test will take a large amount of time, compared to the other approaches. Third, if the tests are run at the same time, it may be hard to determine which module in the module call tree failed. This can be remedied, if error handling and propagation are well designed and allow for tracing by bubbled errors messages. Side effects of running all modules of the system at once may occur that influence the test. Coincidentally, the detection of occurrences of these kinds of problems, is one reason why a big-bang test may be employed in complement with bottom-up, top-down or mixed strategies.

Next, system testing involves testing concrete user workflows, e.g. from user creation, login, and book lending in our library example illustrated in Figure 1.

Finally, the process concludes with the acceptance testing phase. Initially, based upon the initial requirements specification, project acceptance criteria were devised, that specify, what parts of the requirements need to be fulfilled, so that the software component is accepted as complete. Furthermore, to allow for some room for error in software development, it specifies how many high level, medium level and low level test failures are acceptable for the module to still past the acceptance test.

Considering the example in Figure 34, Line 2 in CheckoutCounter and Line 2 in LibraryService specify a maximum balance in unpaid fees that a library may accumulate, before he is denied a book loan. If the implementation of LibraryService changes as illustrated in Figure 5, where the maximum balance is now an attribute defined on a per-user basis, the CheckoutCounter would potentially not throw the desired InsufficientFundsException to display a message to the counter clerk.

3 Integration Aware Test Case Generation

This section, will describe the approach, constraint propagation system and test case generation of the proposed new JUnit test suites for automatically creating integration checker tests that take the real behavior of multiple modules into account. The scheme proposed here, is depicted in Figure 6. (1) The modified symbolic execution system takes as input the client under test (CUT). For each symbolically executed path π through the CUT, the system generates a set $\mathcal{C}_\pi$ of constraints. (2) Next, the constraints are filtered into the set $\mathcal{C}_I \subseteq \mathcal{C}_\pi$, This represents only the constraints relating to variables involved in web services calls. Currently, we use a file that stores these constraints (*constraint store*).

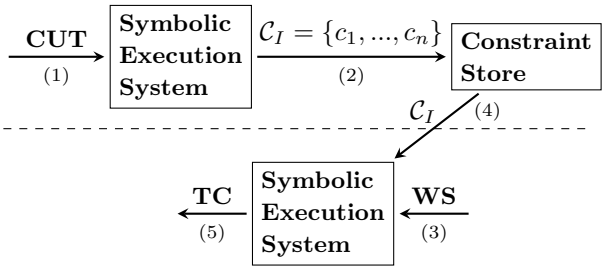

Figure 6: An overview of the constraint propagation and test case generation approach.

Additionally, a unique service identifier name $\mathcal{N}$ that represents the called component is calculated. It contains the following information.

I) The domain or the network address of the called web service,
II) the number and type of parameters of the called method and
III) the return type of the called method.

Finally, $\mathcal{C}_I$ and $\mathcal{N}$ are send to the constraint store.

In a second phase, we use the *constraint store* and the service under test (including the web service) to generate a test case (TC) for π through the CUT.

Using stored constraints for integration checker test suite generation. (3) The second part of the schema occurs, once the modified symbolic execution system is started in server side generation mode. The result of the server mode execution depend on the constraint store. The modified symbolic execution system generates the service ID object for the method under test, by examining the annotations present prior to the method declaration. The constraint store is queried for the constraint set for this service ID. (4) If the constraint store offers an empty set as result, the computation is finished. Otherwise, the test case generator receives the result set $\mathcal{C}_I$ and starts in **normal** execution mode, i.e. without taking the received constraints into account. Once this step is completed, the test case generator starts in **additional** execution mode. The gathered constraints $\mathcal{C}_I$ relate to the parameters and return variable of the method under test. Whenever the symbolic execution passes a choice-point, we conjunct the constraints $\mathcal{C}_I$ to the constraint stack and check the constraint solver for satisfiability as usual. This continues, until the choice-point stack is empty and all node and edges of the control-flow graph have been covered. If a branch, that was previously unreachable in normal mode, becomes reachable in additional mode, this hints at a possible integration error. (5) If a branch becomes unreachable, we keep the normal branch, but generate a failing test case. The test cases generated for these

```
1  public LibraryServiceTest {
2   @Test
3   public void test() {
4    User peter = new User();
5    Account peterBilling = new Account();
6    peterBilling.setUser(peter);
7    peterBilling.setBalance(51L);
8    peterBilling.setLimit(52L);
9
10   LibraryServiceV2 ws = new LibraryServiceV2()
         ;
11   assertNotEquals(null),ws.requestLoan(peter,
         null));
12  }
13
14  @Test
15  @Category(PotentialIntegrationFault.class)
16  public void testIA() {
17   User peter = new User();
18   Account peterBilling = new Account();
19   peterBilling.setUser(peter);
20   peterBilling.setBalance(51L);
21   peterBilling.setLimit(52L);
22
23   LibraryServiceV2 ws = new LibraryServiceV2()
         ;
24   assertNotEquals(null, ws.requestLoan(peter,
         null));
25   fail("The additional execution revealed,
         that this branch became unreachable");
26  }
27 }
```

Figure 7: Normal WS' Test Case and additional (integration) Web Service WS' Test Case.

branches, are saved into a dedicated test suite, marked with PotentialIntegrationFault.

Making use of the dedicated test suite. The new additional automated generate test suite is used as follows. Instead of creating a component stub or mock for top-down and mixed-mode integration testing or including the whole lower-level component in the component currently under integration testing, neither is necessary to reap benefits of the new approach. Instead, the dedicated test suite is executed against the unmodified program, if information regarding the compatibility of component external use of the method is required. Should one of the dedicated test suites fail, this does not necessarily point to an error in the server code. However, it does indicate, that the client that was responsible for the creation of the relevant constraints, will fail if it runs into the parameter scenario described in the test case. If, e.g., the failure of the dedicated test suite occurs after the server code was changed, then if this change is deemed to be correct, then the corresponding client should be informed, that an impending change to the server will certainly impede the client operation. A secondary benefit of this approach is, that given the case of service usage across organizational borders, clients may voluntarily offer their server usage constraints to the constraint store, to

possibly guide development of the server component or at least be notified of impending compatibility breaking changes on the server side. Consider the simplified example: Figure 4 is the actual implementation on the server side. The developers introduce a change to the service which results in Figure 5. The client is unaware of these changes and does not adapt the implementation accordingly. However, if the constraints regarding the service usage are stored in the constraint store, upon generating test cases for the server project, the problematic usage of the client can be exposed. Figure 7 depicts the results of both the **normal** execution in Lines 1-12 and the additional generation step in Lines 14-26. The **additional** generation procedure test case will fail upon execution, giving the hint to the server, that there is a problem with the way the service is utilized.

4 Related Work

Automated test data and test case generation has been the subject of an extensive research effort. As a result, several techniques and tools have been proposed. *Search-based* approaches [13] use optimization algorithms to identify (near) optimal solutions. This approach can be applied to software testing [3, 6, 16, 18] by optimizing testing criteria. EvoSuite [10] is an automated search-based unit test generation approach.

Our approach is based upon *symbolic execution*. EFFIGY [14] is one of the earliest systems that uses symbolic execution to generate test cases for programs. Recent approaches [7, 12, 19, 20] use symbolic (or *concolic* as a combination of symbolic and concrete) execution to generate test cases for more complex programs. Muggl [17] is an symbolic-execution tool for automated unit test generation.

Based on this, several techniques exist, that aim to test web service, e.g. by generating data for requests that invoke specific operations of the service [2, 4, 22, 8, 15] and generate unit tests based upon the source code [11, 1]. However, these techniques take only either the client-, or only the server-side into consideration and are unaware of integration implications that can be observed if both sides are taken into account. To the best of our knowledge, no other approaches have used the constraints generated during symbolic execution of web service clients and reused them to generate a dedicated test suite for the purpose of detecting potential integration test failure ahead of the integration test phase.

5 Conclusion

Integration testing requires significant amount of time and resources to complete, which is why some larger companies restrict the use of integration testing to high profile features [21]. To answer this, we have presented an approach to move the detection of potential errors regarding the integration of service from the integration phase to earlier phases, by enabling the detection of *potential* problems during test case generation for single component (mock-less) unit testing.

The approach extends an existing symbolic execution automated test case generation tool, capable of generating web service mocks and stubs, with the ability of reusing constraints as test artifacts for future test case generation runs. This allows us to detect if service consuming clients—even from other organizations—may be affected by changes in the source code of the service providing classes, without doing integration testing.

Due to the prototypical nature of this work, the constraint store only stores its artifacts directly as objects, so a more robust architecture for storing constraints would be preferable. Currently, the work focused on web services, as they are a popular means of integrating components across organizational boundaries, but it can be easily applied to other modes of interaction, e.g. message oriented middleware.

References

[1] A. Arcuri. RESTful API Automated Test Case Generation. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 9–20. IEEE, 2017.

[2] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based Automatic Test Case Generation for Web Services Testing. In *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, pages 207–212. IEEE, 2005.

[3] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary Testing in the Presence of Loop-assigned Flags: A Testability Transformation Approach. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 108–118. ACM, 2004.

[4] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 326–335. IEEE, 2009.

[5] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[6] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms.

In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2005.

[7] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] T. Fertig and P. Braun. Model-driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1497–1502. ACM, 2015.

[9] K. Forsberg and H. Mooz. The Relationship of System Engineering to the Project Cycle. In *INCOSE International Symposium*, volume 1, pages 57–65. Wiley Online Library, 1991.

[10] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[11] A. Fuchs and H. Kuchen. Test-case generation for web-service clients. In *Proceedings of the The 33rd ACM/SIGAPP Symposium On Applied Computing*, Pau, France, 2018. Publication status: Accepted.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[13] M. Harman and B. F. Jones. Search-based Software Engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[14] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[15] P. Lamela Seijas, H. Li, and S. Thompson. Towards Property-Based Testing of RESTful Web Services. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 77–78. ACM, 2013.

[16] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on software engineering*, 33(4), 2007.

[17] T. A. Majchrzak and H. Kuchen. Automated test case generation based on coverage analysis. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 259–266. IEEE, 2009.

[18] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The Species per Path Approach to Search-Based Test Data Generation. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 13–24. ACM, 2006.

[19] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[20] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[21] M. Wacker. Just say no to more end-to-end tests. `http://googletesting.blogspot.co.uk/2015/04/just-say-no-to-more-end-to-end-tests.html`.

[22] W. Xu, J. Offutt, and J. Luo. Testing Web Services by XML Perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.