Generating Integration Test Cases Automatically

Konstantin Rubinov
Faculty of Informatics
University of Lugano, Switzerland
konstantin.rubinov@usi.ch

ABSTRACT

In this thesis, I investigate the problem of automatically generating test cases. In particular, I focus on the problem of automatic generation of integration test cases from unit test cases. I start from the observation that software is usually provided with many unit test cases, and that unit test cases carry a lot of information about the unit execution that can be used to generate integration test cases. This paper illustrates the approach that I am investigating and that consists in capturing information in unit test cases with static analysis techniques to automatically merge unit test cases to produce useful integration test cases. The preliminary results reported in this paper provide evidence of the effectiveness of the approach. My current research is developing the approach further and producing additional experimental evidence. I expect to complete the research by defining a set of design for testability rules to produce software that facilitates the automatic generation of integration test cases.

Categories and Subject Descriptors

D.2.5 [**Software**]: Software Engineering—*Test design*

General Terms

Reliability, Verification

Keywords

Software testing, unit and integration testing, automatic test generation, design for testability

1. MOTIVATION

Software testing is an important yet expensive part of the software development process. Automation of testing activities can decrease the cost and increase the quality of software [7]. Extensive research has addressed the problem of automating testing activities, with a focus on the different levels of testing, from unit to integration, system and acceptance testing [2]. The effort towards test automation has produced many tools that provide different degrees of automation of core activities [9, 4, 10][1], but there are many expensive testing activities that are still only partially automated. My PhD research focuses on the problem of automating one of these human-intensive activities: the generation of integration test cases.

Existing approaches for automating generation of integration test cases are few and often rely on system executions to generate test cases [12, 6]. However, relying on system executions has several disadvantages as pointed out by Xie [11].

2. RESEARCH GOAL AND CHALLENGES

The core idea behind my PhD research is to automatically generate integration test cases based on existing unit test cases. Unit test cases exercise the code of units independently from the other units. Integration test cases exercise the same code focusing on inter-unit actions and communication. I intend to exploit this overlap to automatically generate integration test cases by combining setup procedures, execution scenarios and oracles of different unit test cases. Such combination increases the granularity of generated test cases, and greater granularity positively affects their cost-effectiveness [8].

Many existing software projects produce unit test cases. Modern software development processes guided by test-first practices combined with extensive tool support have increased the automation level and facilitate the generation of a large number of unit test cases. Integration testing is also an important activity following unit testing, but there is no substantial tool support for generating integration test cases. Generating integration test cases from unit test cases can benefit from current test-first practice where a large number of unit test cases are generated before the units are implemented and integrated.

My goal is to develop a technique for the automatic generation of a considerable amount of integration test cases; a technique that operates with the information available in unit test cases and source code of the system under test. In a preliminary study I investigated unit and integration test cases from several open-source software projects. I studied their structure, complexity, and internal dependencies to determine the overlap between integration and unit test cases, and how it is reflected in the structure of test cases. The results are promising and indicate that due to substantial

[1]see http://www.opensourcetesting.org/ for a list of open-source testing tools.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

overlap I can generate integration test cases on the basis of unit test cases.

The preliminary study also highlighted several research questions that I will address in my doctoral work:

Q1: What information is required to generate integration test cases automatically?

Q2: How can this information be extracted automatically?

Q3: How can the extracted information be combined to generate integration test cases automatically?

Q4: How effective are the automatically generated integration test cases? In particular, to what extent can newly generated integration test cases substitute or augment integration test cases that can be derived with traditional techniques, and how many integration test cases can be automatically generated?

3. APPROACH

I apply an iterative experimental approach in my research. In each iteration I study the sources of information such as unit test cases to find patterns in the available data. Based on these data I develop test integration strategies that guide the test generation process. I evaluate the quality of generated test cases as described in Section 4 and, if necessary, refine my answers to research questions *Q1-Q3*. In each iteration I might face new challenges that I will address upon their appearance.

Q1: Required information. To address the first research question, I studied the characteristics of test cases using static analysis and inspection. In principle, test cases on all abstraction levels, from unit to integration and system testing, have a common structure. In the initial study I divided test cases into four parts: (1) test scaffolding, (2) mutual setup of interacting objects with a correct data, (3) execution of the functionality under test, and (4) oracles to verify the mutual state of interacting objects.

However, my investigation indicates that unit test cases use little or no scaffolding and usually have only three parts: test initialization, which corresponds to the mutual setup of objects, unit execution and verification of the results through assertions that serve as oracles. The structure of unit tests tends to be simple: most unit test cases include a small number of setup objects that form the test context (Figure 1, lines 7-14), and include many test methods applied to the few setup objects (Figure 1, lines 15-29). Moreover, unit test cases contain information about the order of object creation, mock objects, values passed in the method calls and assertions, order of execution, and how the objects are connected.

Integration test cases often require instantiating real objects rather than mock objects and more complex interactions between them. Generating such test cases requires specific information. Integration *test scaffolding* requires external dependencies to be resolved and the environment to be set up for interacting objects. The *mutual setup* of interacting objects requires information about object dependencies and the data for their initialization and wiring. The *execution order* for interacting objects must represent meaningful combinations of actions on integrated objects. The *oracle*

```
1  public class LanderTest {
2    Speedometer speed_mock;
3    Altimeter alt_mock;
4    Barometer baro_mock;
5    Lander lander;
6
7    @Before
8    public void setUp() throws Exception {
9      speed_mock = new SpeedometerMock(5);
10     alt_mock = new AltimeterMock(1);
11     baro_mock = new BarometerMock(300);
12     lander = new Lander(speed_mock,
13           alt_mock, baro_mock);
14   }
15   @Test
16   public void testStopLanding() throws
         Exception {
17     lander.startLanding();
18     lander.stopLanding();
19     assertTrue(lander.isLanded());
20   }
21   @Test (expected=Exception.class)
22   public void testStopLandingEx() throws
         Exception {
23     alt_mock = new AltimeterMock(100);
24     lander = new Lander(speed_mock,
25           alt_mock, baro_mock);
26     lander.startLanding();
27     lander.stopLanding();
28     assertTrue(lander.isLanded());
29   }
30 }
```

Figure 1: JUnit test case

for the mutual state of interacting objects requires considering the state transitions of these objects.

I examined more than 2000 test cases of many open-source projects. In particular, test cases developed with the JUnit framework[2]. Comparing unit and integration test cases from the same projects revealed a large overlap between the code of unit and integration test cases. Up to 80% of the code is identical in the sense that test cases instantiate the same objects and exercise the same code of the system under test. This overlap indicates that most of the information required for the generation of integration test cases is available in unit test cases.

Q2: Extracting information automatically. To address the second research question, I am investigating dynamic and static analysis techniques to automatically extract relevant information.

The external dependencies and the environment for interacting objects required for integration *test scaffolding* are available in the setup procedures of individual unit test cases (e. g., method `setUp()` in Figure 1). There is no need for specific analysis to obtain this information, because these procedures can be reused directly as they appear in the code of unit test cases. Minor modifications can be required to resolve naming conflicts and to substitute mock objects with real ones.

The *mutual setup* of interacting objects with correct data requires knowledge about object dependencies. My investigation suggests that the structure of unit test cases not

[2]http://www.junit.org

only allows to reveal dependencies among units, but also shows how dependent units interact with each other, for instance, when one unit produces primitive values, objects, or resources used to initialize another unit.

Object dependencies in unit test cases can be extracted by static analysis, and I capture them in a modified object relation diagram (ORD) [3]. Note that I construct the ORD from unit test cases, where relevant dependencies are present, and not from code or UML diagrams as in classic approaches. In such an ORD nodes represent test cases, and there is an edge connecting two nodes whenever the test cases use or aggregate objects of the same type. Consider the test case `LanderTest` and assume there are test cases for associated objects, like `Barometer`. `LanderTest` and `BarometerTest` have a dependency (`Barometer`), and thus are candidates to provide information to create an integration test case.

The other information required for the *mutual setup* is data to be passed to objects during their initialization. This information can be captured by either statically extracting constant parameters in unit test cases or by executing test cases and recording the actual parameter values.

The *execution order* for interacting objects requires capturing existing call sequences and combining them in a meaningful way. Control flow analysis can be used to capture existing call sequences in unit test cases as well as in the code of the units under test. As for the combination of call sequences, this is mostly related to the development of the test integration strategy discussed later in this paper.

The *oracle* for the mutual state of interacting objects requires modification of the individual oracles of unit test cases according to the new execution order and possibly according to the different states of objects. Currently I am investigating approaches for the generation of oracles and oracle transformation, such as recent work by Marinov et al. [5], and inference of invariants for integrated code.

Throughout experiments I observed that unit test cases are linear in their structure and their average cyclomatic complexity is close to 1, which means there is only one execution path in almost all test cases. Thus the intra-procedural analysis of such unit test cases can be computationally inexpensive. Therefore the overall technique could be very efficient in automating the generation of integration test cases.

Q3: Generating test cases automatically. I identified four main issues in generating integration test cases automatically. They are (1) determining candidate unit test cases for integration, (2) determining the order of execution within integrated test cases, (3) resolving the incompatibilities in test oracles for integrated test cases, and (4) determining the order of integration of multiple candidate unit test cases when generating complex integration test cases. These issues are inter-dependent and solutions to them should be considered in conjunction. In my work I define test integration strategies that represent combinations of such solutions.

(1) Useful integration test cases exercise relevant interactions between dependent units. I propose to combine pairs of unit test cases to obtain integration test cases. However, the number of possible test case combinations grows exponentially with the number of test cases, and many combinations do not yield useful integration test cases. This is because merging test cases that do not have anything in common does not produce test cases that focus on the integration of units, while merging test cases that share objects and actions generates useful integration test cases.

Currently I am using dependency analysis to determine which unit test cases can be integrated. I am using aggregation or use dependencies in ORDs to form pairs of candidate test cases to be integrated.

(2) To verify the correctness of interactions of several related units and to detect possible side-effects, integration test cases have to exercise non-trivial sequences of method calls. Determining meaningful sequences of method calls to form new integration test cases requires careful selection among large number of possible sequences.

I am extracting such sequences from unit test cases and forming various scenarios of unit integration. Currently I am experimenting with the order of integration of call sequences of different unit test cases. These sequences can interfere in many ways when merged. I plan to investigate in which cases the internal order in these sequences can be preserved or has to be modified, and when the data values passed to the method calls can also be preserved or have to be modified.

(3) Oracles in unit test cases are mostly implemented as assertions with value comparison. Such oracles are not general enough to be reused directly in the integrated test cases, because longer integrated execution sequences may involve state changes not considered in the unit test cases the oracles originate from. Oracles in integration test cases should also operate on bigger combinations of states than the oracles of individual unit test cases. This poses the more general problem of generating test oracles that has been addressed by extensive research [1].

Currently, I use original oracles in generated test cases and I suggest to the test developers to analyze and adapt the oracles when necessary. To address incompatibilities in test oracles in the future, I am investigating existing approaches for oracle transformation.

(4) Complex integration test cases can be generated by combining several unit test cases. I am experimenting with generating integration test cases by incrementally aggregating test cases generated in previous iterations to maximize the involved units and their interactions. To determine the order of integration I start integrating pairs of least dependent candidate test cases and proceed by integrating more dependent ones. Each step of integrating pairs of dependent candidate unit test cases is reflected in the ORD that is updated with the new nodes and relations accordingly.

The test integration strategies I will develop as answer to research question *Q3* are the main contribution of my thesis. These strategies rest on the answers to research questions *Q1* and *Q2* and are the subject of the evaluation described in the next section.

4. EVALUATION

To answer research question *Q4* I will evaluate my complete solution for automatically generating integration test cases both quantitatively and qualitatively. First, I will evaluate how many of the automatically generated integration test cases are equivalent to those generated with established techniques [7], and how many are not. The number of equivalent test cases shows to what extent my technique can substitute the techniques I compare against, and the number of non-equivalent test cases shows its added value. Second, I will evaluate the quality of the automatically generated test cases assessing their fault detection capabilities and explor-

ing their sensitivity to particular types of faults. To that end I will execute generated test cases on various versions of software with known integration faults and seeded faults.

In the evaluation I will rely on software obtained from open-source repositories with available unit and integration test cases. In case of missing test cases I will generate them manually applying well-known combinatorial testing approaches.

I will evaluate the completeness of the generated test suites according to different established combinatorial and model-based criteria. I will also investigate combinations of functional and structural criteria. In addition, I plan to develop a number of approach-specific measures to assess the completeness of the generated test suites.

5. RESEARCH RESULTS AND PLAN

At the present time I am in the second year of my PhD research. So far I worked on the *first research question* and I determined the structure of unit and integration test cases, the information available in unit test cases, and information required to generate integration test cases. I plan to complete answering the first research question experimentally to find out if additional information is required for automatic generation of integration test cases.

For the *second research question* I have selected various static and dynamic analysis techniques to automatically extract and capture information available in unit test cases. My plan is to complete answering the second research question by developing a framework for automating the analysis of unit test cases, ORD construction, and the information extraction processes.

With regard to the *third research question* I have identified the four main issues in automatic generation of integration test cases that I will resolve in my thesis work. So far I have successfully dealt with the first issue – determining candidate unit test cases for integration on the basis of ORDs. My experiments indicate that ORDs need to be extended to deal with test cases that have dependencies through multiple instances of the same type. I will investigate these cases in the next iterations of experiments.

Currently the focus of my research is on the second issue – determining the execution part of integration test cases. To automatically create integration execution scenarios I use sequences of method calls extracted from unit test cases, and if available, the integrated source code where actual integration scenarios are implemented, and information supplied with the development process.

Addressing the third issue of resolving the incompatibilities in test oracles for integrated test cases, I plan to apply the technique proposed by Marinov et al.[5] and to conduct controlled experiments with test developers to establish its sufficiency in the scope of my technique. I will use results of these experiments as a foundation for my future research.

With regard to the forth issue, I am investigating the process of generating complex integration test cases based on multiple candidate unit test cases. Following this process I plan to obtain system test cases from integration test cases in the future. This may require exploring some additional sources of information, which is a subject for the further investigation.

As my research progresses I will develop various test integration strategies corresponding to different solutions to the four indicated issues. Currently I am developing test integration strategies iteratively starting from simple test integration strategies and proceeding towards more sophisticated strategies as I am collecting more data.

In the scope of the *forth research question* I plan to evaluate my solution as described in the Section 4. I will evaluate the approach on several case studies of increasing complexity by comparing the set of test cases automatically generated with integration test cases produced with systematic approaches, and I will evaluate the completeness and scalability of my approach.

I plan to investigate the possibility to extend the approach to generate system test cases from integration ones, and identify the limits of the approach. I expect to be able to identify design patterns that simplify the automatic generation of complete sets of integration test cases, and other patterns that complicate the generation. Based on these data, I plan to define a set of design for testability guidelines that will complete the thesis work.

6. REFERENCES

[1] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, August 2001.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, 2007.

[3] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. SW Eng.*, 29(7):594–607, 2003.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer, 2005.

[5] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA '10: Int. Symp. on SW Testing and Analysis*, pages 207–218, 2010.

[6] S. Elbaum, H. N. Chin, M. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. SW Eng.*, 35(1):29–45, 2009.

[7] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, Inc, 2007.

[8] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. SW Eng. Methodol.*, 13(3):277–331, 2004.

[9] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. 23rd Int. Conf. on Automated Software Engineering ASE*, 2008.

[10] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: object-oriented unit-test generation via mining source code. In *ESEC/FSE*, pages 193–202, 2009.

[11] T. Xie. Improving automation in developer testing: State of the practice. Technical Report TR-2009-6, North Carolina State University Department of Computer Science, February 2009.

[12] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *WS on Automation of SW Test*, pages 64–70, 2006.