Towards Automatic Scenario Generation from Coverage Information

Melanie Diepenbeck[1] Mathias Soeken[1,2] Daniel Große[3] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3]solvertec GmbH, 28359 Bremen, Germany
{diepenbeck,msoeken,drechsle}@informatik.uni-bremen.de, grosse@solvertec.de

***Abstract*—Nowadays, the design of software systems is pushed towards agile development practices. One of its most fundamental approaches is Test Driven Development (TDD). This procedure is based on test cases which are incrementally written prior to the implementation. Recently, Behavior Driven Development (BDD) has been introduced as an extension of TDD, in which natural language scenarios are the starting point for the test cases. This description offers a ubiquitous communication mean for both the software developers and stakeholders.**

Following the BDD methodology thoroughly, one would expect 100% code coverage, since code is only written to make the test cases pass. However, as we show in an empirical study this expectation is not valid in practice. It becomes even worse in the process of development, i.e. the coverage decreases over time. To close the coverage gap, we sketch an algorithm that generates BDD-style scenarios based on uncovered code.

I. Introduction

As the name indicates, Test Driven Development (TDD, [1]) is a development technique driving the implementation where tests are the essential elements. More precisely, for each passing test, there must exist some implementation. However, the opposite direction is not necessarily required, i.e. there may be program code for which no test exists. Hence, it is likely to be the case that programs written in a TDD manner are not fully covered. In industry and academia some empirical studies investigated the quality of using TDD and found out that the testing coverage drops below 100% when the projects are developed for a longer period of time [2], [3].

Extending TDD, Behavior Driven Development (BDD, [4]) uses natural language to describe acceptance tests thereby easing the communication between designers and stakeholders. The used BDD language terminology focuses on the behavioral aspects rather than on testing. For this purpose, the test structure borrows its ideas from user story specifications of requirements engineering. These stories are called *scenarios*, which describe specific examples of how the software should work. Each scenario is written in a constrained natural language and constitutes one test case. As a result, in every step of the development cycle well-defined outputs are being specified before implementing the software. In particular, due to its more intuitive behavioral specification, it is a useful methodology for Completeness-Driven Development (CDD, [5]) and therefore the question is raised whether using BDD achieves a higher or the same code coverage.

The contribution of this paper is twofold. First, we have empirically evaluated BDD projects where we discovered that full coverage is not achieved. We compare these observations to reported coverage results of TDD. The studies on TDD underline our assumptions. While the test cases ensure a high coverage in early versions of the project, the coverage drops over time as the project becomes more mature and the project size grows. As a consequence, an increasing coverage gap results.

To close this coverage gap, we sketch an algorithm for generating test cases based on uncovered code. In doing so, test cases are generated in a BDD scenario style by making reuse of already existing step definitions.

The remainder of this paper is structured as follows: At first Section II discusses related work. Background on BDD is given in Section III. Our empirical study is presented in Section IV while the proposed algorithm idea is presented in Section V. Finally the paper is concluded in Section VI.

II. Related Work

Automatic test generation is a widespread research topic for software and hardware system design since defining good tests is a time-consuming task.

In [6], [7], and [8] test data is generated from UML specifications, in particular state diagrams. The authors of [9] present another automatic test case generation approach based on UML communication diagrams to target cluster level tests which can be used to describe a story similar to the ones in BDD. But generally these approaches generate tests from a predefined specification while our algorithm considers only the uncovered code parts in the development cycle and can suggest missing test cases in each step.

In order to improve the test coverage, [10] presents a tool that assists the programmer in determining for what code parts tests need to be written and gives some hints on how to exercise these. To achieve this it uses the program's control and flow graphs. But in contrast to our approach test cases need to be written completely by the programmer.

There exist many random testing techniques such as [11], which also incorporates feedback of executed inputs. Units are generated from the source code, however, unlike our approach the procedure is not trying to find missing test cases to achieve a full coverage.

978-1-4673-6161-3/13/$31.00 © 2013 IEEE

AST 2013, San Francisco, CA, USA

The generation of test data using model checking [12], [13] is being actively investigated in research. After the modeling step a model checker is applied, then the resulting counterexamples are used to generate new tests. One example for such a model checker is the Java PathFinder [14], which can generate test cases by means of symbolic execution.

To the best of our knowledge the automatic scenario generation for BDD is the first approach that supports the developer to complete her test set in an intuitive manner.

III. Behavior Driven Development

TDD is a design flow paradigm in which test cases are provided as a starting point and as central elements along the whole design process. First, all test cases are specified such that they all fail initially without an implementation — otherwise the test cases are not useful. Based on the error messages obtained from the failing test cases, the designer can extract feedback what needs to be implemented in order to fulfill the tests. This process leads to an incrementally growing implementation eventually leading to a system for which all test cases are passing.

Tests can be specified in different abstractions, ranging from low level unit tests that target very specific aspects of single classes and their methods to integration tests that consider the system as a whole. Based on the TDD design flow paradigm, in particular for the integration tests BDD has been recently proposed in which more coarse-grained tests are specified by means of natural language targeting the behavioral level of designing software.

In the context of this work, we will limit our observations on BDD for Ruby [15] development. Two frameworks are of a particular importance: Cucumber [16] and RSpec [17]. Note, that the general flow can be applied to other programming languages and BDD tools accordingly.

A. Cucumber

In Cucumber, the natural language specifications of the test cases are strictly separated from the actual test code. This allows different views on the tests, in which the plain natural language description of test cases offers a ubiquitous communication mean for both the software developers and stakeholders. The natural language ensures a common understanding of the system to be developed between all partners of the project. All test cases are called *acceptance tests* and structured by means of *features* or *feature files*, where each feature can contain several *scenarios*. Each scenario constitutes one test case and is based on the *Given-When-Then* sentence structure, in which each sentence is referred to as a *step*. This terminology is illustrated by means of an example feature on the left-hand side of Fig. 1.

Consider the first scenario in the same figure. This scenario originates from an advanced development stage of a submission system. It describes the submission of a paper to such a system. However, in order to execute the scenario, we have to bind the steps to actual test code. This can be achieved using so-called *step definitions* which are tuples of a keyword (such as *Given*, *When*, or *Then*), a regular expression, and *test code*. Whenever a step of a scenario matches the regular expression, the test code is executed. Some of the step definitions for the scenario described above are formalized on the right-hand side of Fig. 1.

Consider the first step definition that matches the first step “Given I am an authenticated “author” user” from the scenario in Fig. 1. Here, a new user of type “author” is created for which the authentication procedure is performed. For the remaining steps, the user selects the submission page by following navigation links, enters the details of her new submission to the system and submits the paper (*When*-step and *And*-step). The correct behavior is checked by inspecting the notification after the submission (*Then*-step).

As indicated in the second scenario of Fig. 1, the step definitions can be reused for similar steps since regular expressions are used to match the steps. For example, the user type “author” could be replaced by “chair” without modifying the step definitions.

The step definitions are written before the implementation phase has been started based on the scenarios given in natural language. Thus, design decisions affecting the structure of the implementation are taken while writing the test code for the step definitions.

For the scenario at hand, it has been decided that there needs to be a class for an authenticated *User* and a class *Submission* that has operations for adding a new submission to the database and presenting a notification to the user.

During the implementation phase, the test cases are usually executed whenever the implementation changes, preferably using a background task running autonomously. Each scenario is run as a separate test case with no interaction to other scenarios or features. Therefore for every test a clean *starting point* can be identified which is the first step in a scenario. When executing the test cases, steps can only fail due to two reasons, i.e. *syntactic* and *semantic* errors. The first class of errors occur whenever a name cannot be resolved, e.g. when there is no class called *User*, or there is no method *msg* available yet. The second class of errors consists of errors that occur whenever values do not match their expectations. This can only happen in assertions, i.e. step definitions such as the second one given on the left side of Fig. 1. This is done with the automatically loaded *assertion* library from RSpec, which offers an intuitively syntax such as in the last step definition using ‘`should contain()`’.

B. RSpec

RSpec is one of the first TDD frameworks that puts the focus on behavior. The programmer writes executable examples of the desired behavior of a small parts of code in a controlled context. In contrast to Cucumber, there is no strict separation of the natural language part of the test and the test code itself. Here is an example of the structure of an RSpec test case:

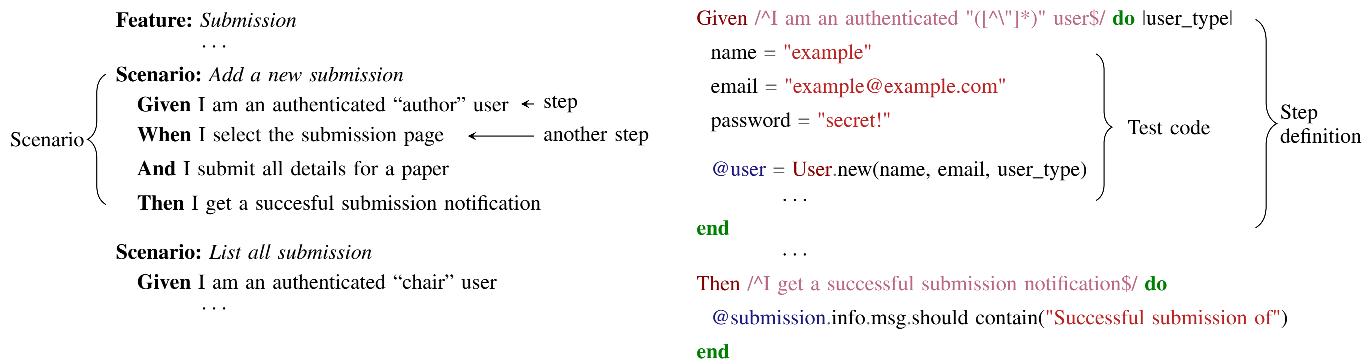

Fig. 1. BDD definitions

```
describe SubmissionDatabase do
  context "when first created" do
    it "is empty" do
      submission_database = SubmissionDatabase.new
      submission_database.should be_empty
    end
  end
end
```

As for any BDD tool, RSpec also builds on simple examples that express the basic expected behavior. RSpec embeds a domain specific language that covers the description of the behavior of an object. It consists of the three main methods '`describe`', '`context`', and '`it`'.

With the '`describe`'-method an *example group*, called `SubmissionDatabase`, is declared which is used to bundle the examples of the expected behavior of an object. In this case a class *SubmissionDatabase* should be developed. The '`it`'-method identifies an *example*. In the given example the behavior of the *SubmissionDatabase* is being described for the case it is created for the first time.

The '`describe`'-methods can be nested in order to trace different hierarchies of modules and classes. Another method '`context`' can be used for structuring the desired behavior better, although it is technically just an alias for the '`describe`'-method.

Applying the TDD approach, we would expect full code coverage since there should never exists any implementation code without writing a test for it first. Since BDD builds upon the same principle we would expect the same results. In the next section we are going to analyze code coverage of two BDD projects.

IV. Empirical Study on Coverage in BDD Projects

In this section we present an empirical study on coverage achieved in BDD projects. Since in BDD projects the implementation of code is directed by acceptance tests, only code that is necessary to make the tests pass should be written. Hence, the expected coverage should be 100%. The main goal of this evaluation is to investigate whether following the BDD model leads to full coverage.

The section begins with the set up of the empirical study, followed by a brief overview of the studied projects. Then the empirical results for each of these project are presented and discussed. Finally, we briefly address empirical results of studies on TDD quality.

A. Approach

The empirical study for these frameworks observes code coverage in the progress of development for each considered project. As observation points, the individual versions of each project have been taken. At each point, coverage measures are applied and the original test cases are run again. In order to find code parts that are never executed by any test case, line coverage was used as coverage measure. The employed coverage tool was SimpleCov 0.7[1].

B. Studied Projects

For the empirical study we evaluated two implementations from GitHub[2]. GitHub is a collaboration platform that offers Git as its revision control management system. Thus we only considered projects that were driven by one (or both) of the BDD frameworks Cucumber and RSpec. Both studied projects are applied by a large community in their own software projects.

1) Aruba: Aruba[3], which is continuously being developed for almost three years now, offers an extension to Cucumber that is used for testing command line applications. It can be easily integrated into any Cucumber project and offers many new completely implemented step definitions for tests that target command line interfaces. For example consider the step '`When I run 'echo $1'`'. When this step is executed, the shell command '`echo`' is called automatically. The step does not need to be defined first.

2) Celluloid: The Celluloid[4] project provides a developer tool that simplifies building multi-threaded Ruby programs. The main idea is a new data structure that is an active

[1] https://github.com/colszowka/simplecov
[2] https://github.com/
[3] https://github.com/cucumber/aruba
[4] https://github.com/celluloid/celluloid/

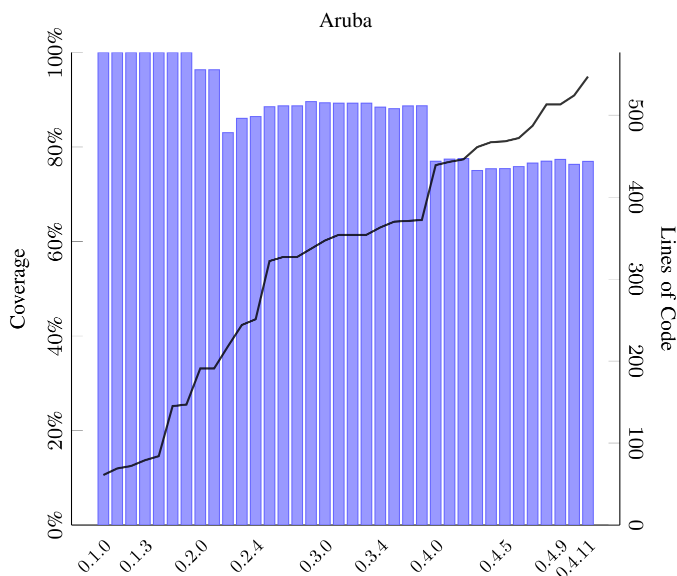

Fig. 2. Coverage results for Aruba with Cucumber (add. RSpec in 0.4.9 ff.)

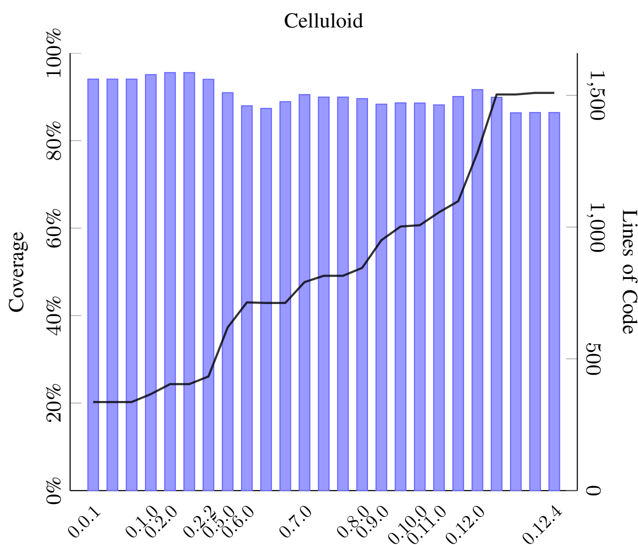

Fig. 3. Coverage results for Celluloid with RSpec

object which runs within a thread, called a “cell”. Celluloid is continuously being developed for almost two years now.

C. Results

In this section we provide the results of our empirical analysis. Figs. 2 and 3 present the line coverage results of the implementations of Aruba and Celluloid. The blue bars show the achieved code coverage after the execution of the original test cases. The lines of code (LOC) are shown for every version as an continuous line with the scale depicted on the right side of the chart. As primary tendency it can be seen that the line coverage for every project is almost always less than 100%. When considering both projects independently, some other observation can be made.

For Aruba, a trend towards a lower coverage can be noticed. In the first versions the development team started with 100% code coverage using Cucumber, which stayed the same for all revisions of version 0.1. But when changing to version 0.2.0 the coverage dropped while the implementation code increased by approximately 29%. The tendency of a decreasing coverage remains during the development of Aruba for the subsequent versions, although the coverage increases slightly in between. The last three versions were not run with Cucumber as the only testing framework but in combination with RSpec. They started with two additional RSpec examples in version 0.4.9 and rose to eight RSpec examples in version 0.4.11. A closer look at the number of Cucumber scenarios (70 in the last version) shows that the RSpec examples take only a small part in the testing task.

In comparison to Aruba, Celluloid shows a more constant code coverage over time. The coverage is in the range from 86.36% to 95.54%. The highest coverage was achieved shortly after the beginning of the development. When Celluloid had a big version jump from 0.2.2 to 0.5.0 the coverage decreased notably and the lines of code increased by approximately 43%. In later versions the LOC stabilized and the coverage grows a bit. Here some new test cases were added, which may be the reason for the increasing code coverage. The number of test cases in Celluloid increased from 26 in the beginning to 130 cases. In the last four versions, there were always 5 failing test cases, that may also be responsible for the decreasing code coverage in these versions.

In conclusion it can be said, that both projects show a similar constant reduction of code coverage during the project development time. In the first instances of the implementation they have a high code coverage, which reduces over the versions, while the implementation code for Aruba increased almost nine times in the period from the first version to version 0.4.11 and for Celluloid more than quadruples from version 0.0.1 to version 0.12.4.

In the next section we revise other empirical studies based on TDD which also made some coverage observations.

D. Studies on TDD

Since TDD can be seen as the predecessor of BDD and both are using the test-first principle, coverage results on TDD projects should also be considered.

In [18] three software development projects were realized using TDD with semi-industrial settings in an empirical study. These projects were compared with two iterative test-last projects. Among other quality measures test coverage was analyzed. In general TDD induced a higher code coverage in method, statement, and branch coverage and achieved up to 100% coverage. But this study lasted only 9 weeks, i.e. the coverage results are comparable with our results for Aruba and Celluloid when considering only the first few versions. If projects run for a short period of time with TDD, they can have a high test coverage.

Two industrial case studies were conducted for not less than four months by Bhat and Naghappan [2]. The first project in the case study reached a block coverage of 79%, while the second one accomplished 88% block coverage using a unit test approach with TDD.

The test cases in the experiment in [3] received a mean of 98% method, 92% statement, and 97% branch coverage. Method coverage ranged from approximately 72% to 100%, similar to the results obtained with Aruba in our experiments. While these coverage results are very high, the resulting application was very small with about 200 LOC in Java.

None of these studies analyzed how the code coverage changes over time, but they show a similar result to our study on BDD projects. When a product is developed with TDD or BDD for a short period, a high code coverage is achieved. But when the development time progresses, coverage often falls below 80%. One of the reasons for this coverage drop is, that developer start writing additional code while adding code to fulfill a certain test case. However, for the additional code no acceptance test is written afterwards. In the next section an approach is presented that will help the developer to close this coverage gap.

V. Coverage-driven Scenario Generation

Based on the investigations presented in the previous sections, this section illustrates an approach that automatically generates Cucumber-style scenarios based on uncovered code. We are only considering line coverage thus far. In order to obtain a testset that fully covers the implementation, we envision a two-stage approach. In the first stage, referred to as *global coverage* in the remainder, all uncovered methods are targeted such that at least all methods are called by the test cases. Afterwards, the remaining uncovered lines are separately considered in a *local coverage* stage. In this local coverage stage, we can always assume that we can start the execution from the method's entry point which is ensured after a successful application of the global coverage stage.

A. Global Coverage Stage

We are interested in high level test cases ensuring a global coverage. As a consequence, scenarios rather than explicit execution traces should be obtained which seamlessly integrate into the BDD-based design flow using tools such as Cucumber. The general idea how to obtain such a scenario is illustrated in Fig. 4. The goal is to generate a test case that covers the uncovered code, depicted by means of the solid circle in the cloud. For this purpose, we try to reuse already existing steps from other scenarios. Hence, the algorithm is aiming to approach the uncovered code as close as possible using consecutive calls of existing steps starting from an initial point. Obviously, no step exists that reaches the uncovered code, therefore, a new step needs to be created for the last step. According to the illustration in Fig. 4, the user will be offered a generated scenario as exemplary depicted in Fig. 5 (following the example of Section III). Aside from existing steps that

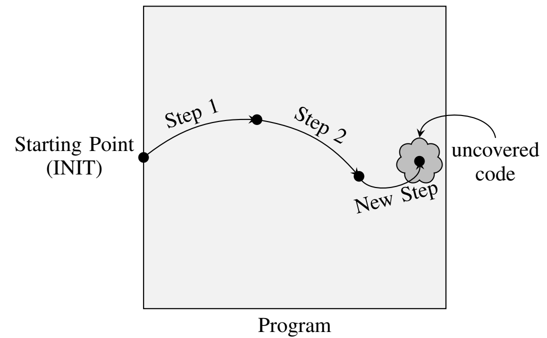

Fig. 4. Approaching uncovered code using step definitions

can be reused, the *new step* calling the method and a *Then*-step (*assertion step*) containing a useful assertion must be added by the developer. Since the new step often only consists of the method call, in practical applications the new step and the assertion step might be combined into a single step.

In order to automatically generate scenarios we introduce the *feature graph*, a new data structure that captures all traces based on the current set of features and their scenarios. The feature graph is a graph whose vertices represent system states and whose edges represent steps which occur in the features' scenarios. An example of a feature graph is shown in Fig. 6. A system state can be seen as an abstraction from the heap, i.e. which objects are initialized together with their type and the assignment to their attributes. For instance, after running the first step of our first scenario from Section III an object of type *User* with attributes `type`, `name`, etc. and an object of type *Page* with its attributes are created. The size of the feature graph is bounded by the number of steps in the features and thus does usually not get too large. The start vertex of the feature graph, called INIT, represents the initial state from which each scenario is executed.

Given the feature graph, scenarios can be extracted from it as follows. We start from an uncovered method m. Let T be the type of the class in which m is defined. Now we determine a vertex v from the feature graph which contains an initialized object o of type T. This is a possible candidate on which m can be called. T can be viewed as a precondition to call the uncovered method m. If a method has parameters, these can also be viewed as additional preconditions to call the method m. This can be further generalized by allowing other method preconditions to constrain possible candidates.

Now any path from INIT to v is a possible trace of steps which represent the first part of the newly generated scenario and the new step consists of the test code $o.m$, i.e. m is invoked on o, followed by a user defined assertion step. The quality of the generated scenario depends on the path that has been taken. For this purpose, we have different strategies in mind, e.g. taking the shortest path. To define better strategies we can assign weights to the edges, e.g. how often a step has been taken in all considered scenarios or the LOC of the test code in a step.

Scenario: *New Scenario*
Given I am an authenticated “author” user
When I select the submission page
And New Step
Then Assertion Step

with

```
When /^ [New Step] $/ do
  ### Insert your code here ###
  ### This needs to be tested ###
  @submission.edit(id)
end
```

Fig. 5. New automatically generated scenario

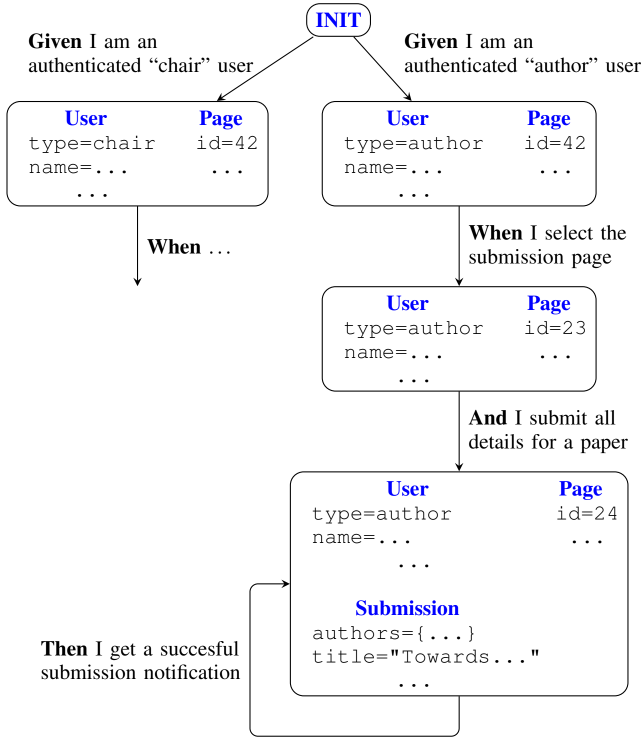

Fig. 6. Feature graph example

If no vertex can be found in the feature graph on which the method can be invoked a new scenario is generated that does not reuse any existing step but consists of only a new step that both creates an instance of the necessary class and then invokes the method on it followed by the assertion step.

B. Local Coverage Stage

Although all methods are covered, the test set may still not be complete since some blocks inside a method have not been covered. The local coverage stage considers these uncovered parts. As an example, consider an *if-then-else* statement in which all test cases visit only the *then* branch. However, in order to tackle these types of uncovered code, more sophisticated approaches are required than the approach being used for global coverage. The global coverage approach is efficient since it is based on a small data structure, which helps to partition the hard cases into their respective methods.

Test cases can then be generated as follows. Given an uncovered line l in a method m, we first determine a scenario from the set of features that covers m. Having performed the global coverage stage in advance, such a scenario must exist. Starting from the method’s entry point, existing techniques e.g. based on symbolic execution [14] are used in order to find a trace that leads to l. For this purpose, it might be required that parameters of m or global variables need to be adjusted. Nevertheless, the coverage problems that occur in the local coverage stage are more complex than the global coverage problem and for some lines no execution trace may be found efficiently.

VI. Conclusions

Although developing software with BDD (or TDD) is a good method to cover a high percentage of code with tests, putting the BDD model into practice is hard. In an empirical study we found out that the coverage drops for projects with long development periods. This observation can be explained since in practice either additional code is added by the developer while implementing code for fulfilling a certain test case, or because code is implemented without adding a test case at all. This observation is also confirmed by the fact that the evaluated projects have a high coverage in the starting phase of the project and drops after a certain amount of time.

Based on the results of the evaluations, we have proposed an algorithm that is generating test cases based on uncovered code. Moreover, these generated test cases are provided as scenarios and are therefore allowing for a holistic user experience when working with a BDD-based design flow. The algorithm separates the problem of finding test cases for uncovered code into two stages in which the first stage ensures a 100% method coverage. Using a new data structure, this stage can be performed efficiently. The second stage then focuses on uncovered code within methods, which is solved by making use of existing techniques. However, the effort is reduced since the first stage allows to consider this problem locally.

In future work, we want to formulate and implement the proposed algorithm and evaluate it by applying it to various projects, e.g. those presented in the study. In particular, we are interested in evaluating different heuristics and strategies on how to determine meaningful paths in the feature graph in order to obtain useful scenarios for the developer. Furthermore it is of interest to incorporate results from formal verification [19].

Acknowledgments.

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

References

[1] K. Beck, *Test Driven Development. By Example*. Amsterdam: Addison-Wesley Longman, Nov. 2003.

[2] T. Bhat and N. Naghappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *Empirical Software Engineering*, 2006, pp. 356–363.

[3] B. George and L. A. Williams, "A structured experiment of test-driven development," *Information & Software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[4] D. North, "Behavior Modification: The evolution of behavior-driven development," *Better Software*, vol. 8, no. 3, 2006.

[5] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *International Conference on Graph Transformations*, 2012, pp. 38–50.

[6] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *International Conference on the Unified Modeling Language*, 1999, pp. 416–429.

[7] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using uml statechart diagrams," in *Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology*, 2003, pp. 296–300.

[8] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "Test Data Generation from UML state machine diagrams using GAs," in *International Conference on Software Engineering Advances*, 2007, pp. 47–52.

[9] P. Samuel, R. Mall, and P. Kanth, "Automatic test case generation from UML communication diagrams," *Information and Software Technology*, vol. 49, no. 2, pp. 158–171, 2007.

[10] S. Horwitz, "Tool support for improving test coverage," in *European Symposium on Programming*, 2002.

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[12] P. Ammann, P. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *International Conference on Formal Engineering Methods*, 1998, pp. 46–54.

[13] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1999, pp. 146–162.

[14] K. Havelund, "Java PathFinder, a translator from Java to Promela," in *5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, 1999, p. 152.

[15] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly Media, 2008.

[16] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookshelf, 2012.

[17] D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesøy, *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends.* The Pragmatic Bookshelf, 2010.

[18] M. Siniaalto and P. Abrahamsson, "A comparative case study on the impact of test-driven development on program design and test coverage," in *International Symposium on Empirical Software Engineering and Measurement*, 2007.

[19] M. Diepenbeck, M. Soeken, D. Grose, and R. Drechsler, "Behavior driven development for circuit design and verification," in *High Level Design Validation and Test Workshop*, 2012, pp. 9–16.