Towards Efficient and Effective Testing in Automotive Software Development

Remo Lachmann and Ina Schaefer

Institute of Software Engineering and Automotive Informatics
Technische Universität Braunschweig
38106 Braunschweig
Email: {r.lachmann, i.schaefer}@tu-bs.de

Abstract: Software systems become more and more complex and control safety-critical applications. Hence, efficient and effective testing procedures are required in order to ensure software and system quality. Automotive software engineering has particular challenges with respect to efficient and effective testing, such as black-box test scenarios, natural language specifications and highly manual testing activities. In this paper, we present six automotive testing challenges and suitable approaches to solve them. We cover uniform test case specification to remove redundancies in test cases, automatic test case selection and prioritization, test case combination, and machine learning techniques for regression test selection. The presented approaches are illustrated by a case study from the automotive domain.

1 Introduction

Software systems become more and more complex and operate in safety-critical environments, e.g., in form of driver assistance systems for automobiles. Therefore, software quality has become an important property of automotive software systems, which has to be ensured for the resulting products. Software testing tackles this issue and is one of the most important and expensive parts of modern automotive software development. Failures, which have not been found by software testing, can cause immense financial downsides and risk for human lives.

The general testing process according to Spillner and Linz [SL12] is split into five phases as shown in Figure 1. It begins with the initial planning phase. Test concept and test plan are defined in this phase. Resources have to be allocated and test goals are defined. After the initial planning, the analysis and design phase follows. Here, test cases are described in an abstract way and defined according to the specification of the software. Subsequently, the test cases and test scripts are implemented. Then, the test cases to be executed are selected and run on the system-under-test, either automatically or manually. After test case execution, the results have to be documented and analyzed. The project lead has to decide whether further testing is necessary. This can be done by analyzing the rate of discovered and fixed failures over time or by other metrics. If testing is not finished, a new iteration of the testing process can be initiated.

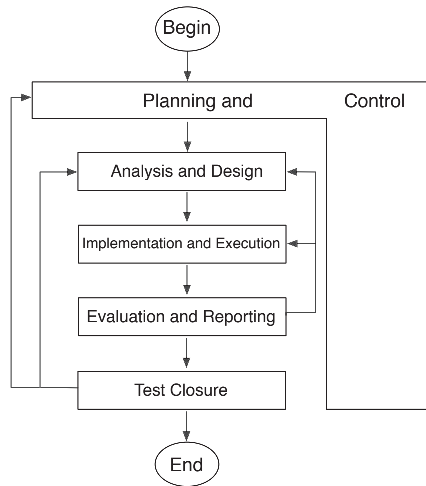

Figure 1: The general testing process [SL12]

Testing in general is heavily restricted by resources which leads to the goal to make testing more efficient and at the same time more effective. In automotive software testing, there are particular challenges. First, in the automotive industry there is a special relationship between OEM, supplier and testers. As software components are often developed by suppliers and tested locally, the integration step often happens by OEM or is performed by another supplier. Due to contractual restrictions, source code often is not available for integration and system testing. This leads to back-box testing scenarios where testing is based on the systems specification. Second, test artifacts are often specified in natural language [LS13]. This increases the difficulty of making assumptions about testing progress and coverage of the system-under-test and to automate test case selection and execution. Third, in most cases, not all specified test cases can be executed due to resource constraints. Therefore, a subset of test cases has to be selected. This is a very critical task, as it is not trivial to select the right test cases, which will probably detect failures. In black-box testing, which is the default for automotive software and systems, no source code is available. This leads to a lack of popular code-coverage metrics. Instead, other heuristics have to be introduced in the automotive domain for test case selection. Fourth, in automotive software development, many sub-processes of the overall testing process are performed manually. Automation of testing processes, including test case selection and prioritization, is a desirable goal, as it can reduce the testing effort compared to manual execution [KTS10].

In order to make automotive software testing more efficient, in this paper, we present six different testing challenges and according solutions within automotive software development taking the specific constraints and challenges in this domain into account. These approach cover test cases specified in natural language, automation of test case selection and prioritization and machine learning for regression test selection. We illustrate the proposed concepts using a running example, the *Body Comfort System* case study [LLLS13].

The BCS consists of different automotive comfort features, which are designed as software product line. Some of the included features are, for example, exterior mirror heating, automatic power window with finger protection, remote control etc. Certain features require other features, e.g., remote control requires a central locking system. Others exclude each other, e.g., manual and automatic power window.

The rest of the paper is structured as follows: Following to the general testing process, Section 2 describes two concepts to improve the design phase. Section 3 proposes approaches which support automated test case selection and prioritization in black-box testing. In Section 4, concepts to reuse knowledge from previous test executions for regression testing are introduced. Section 5 summarizes related work. Finally, in Section 6, we conclude the paper with possible future work.

2 Analysis and Design

Creating test cases in the analysis and design phase is a fundamental task, as it lays the foundation for the testing process. This phase is also important for any form of test automation, as it defines which information is available for testing and how it is accessible. However, occurring problems can get very expensive in later testing phases, e.g., missing test cases, insufficient test case descriptions, test case redundancy etc. The earlier such problems are detected, the cheaper they are to solve.

We describe two approaches to improve testing within this testing phase. First, we describe *systematical test case specification* for natural language test cases, which lays the foundation for test automation. Second, we propose an concept for *removal of test case redundancies*.

2.1 Systematic specification of black-box test cases

In the analysis phase, test cases are often formulated in natural language on system level. Usually, test cases consist of a precondition, action and expected result. Actions are divided into different steps of actions which the tester has to perform to execute the test case. Test cases are usually connected to certain requirements, which are often also specified in natural language. Both requirements and test cases are usually stored in test case management tools, e.g., HP Quality Center, IBM DOORS. Such tools allow for a traceability between test cases and requirements. One goal in black-box testing is to achieve at least requirements coverage. For example, a test case regarding the finger protection feature of the BCS case study could be written in natural language. The specification of the feature defines, that, if enabled, the finger protection detects objects (fingers) in the open window and blocks the window closure to prevent any harm to the finger. A test case for this feature could be defined by two independent testers as shown in Figure 2. A problem, which occurs in the definition of test cases is their actual description. As natural language is still predominant in industrial context, it depends on the tester how a test case is described.

Name:	01_powerWindow
Precondition:	Car has a power window, which is open. Finger protection is activated.
Action:	An object is held into upper third of the window. Then, button "window close" is pressed.
Expected Result:	Window closes until the object is detected. Then, it stops. The Finger Protection LED blinks.

Name:	tc_1_fingerProtection
Precondition:	PW is build in. PW is open. FP is activated
Action:	Hold hand in PW. Press BU.
Expected Result:	1) Window moves up 2) Window stops before hand 3) LED_FP blinks

Figure 2: Test cases for feature Finger Protection

For example, in the given test cases in Figure 2, one tester decided to write full sentences, while the other short bullets and abbreviations. Abbreviations are problematic, as they need to be defined. Otherwise, the test case description is opaque for other persons involved. There are many possibilities, where test case definitions can be different, but still have the same intention.

If test cases are analyzed the following aspects can be considered: if certain components or features are covered more than others, if there are redundancies between test cases, if test cases are too complex, if certain test cases can be combined etc. If a tool would have to analyze test cases as shown in Figure 2, it would be a very difficult and resource consuming task, as natural language processing is still a tough problem, especially if the descriptions are inconsistent as in the example. A solution to this problem is the definition of a systematic test case specification. This requires the test case designer to write test cases using a certain vocabulary, containing allowed abbreviations and terms, in a predefined structure. For example, in the given BCS scenario, the test designer always has to write *PowerWindow* and define precondition, action and expected result in this order. It can also be defined, how many test steps per test case are allowed. A thesaurus can be used to predefine all necessary buttons, fields, parameters etc. and how they are referenced within the test case description. This leads to a uniform test case specification. For example, one could argue that in the BCS case study, the test case variant shown right-hand side in Figure 2 is written according to the defined presets. Finally, such a test case specification has to be supported by the used tools. The tools can suggest to the test case designer, which parameters to reference, where to write them etc.

2.2 Removal of redundancies

Managing large sets of test cases as usual for automotive software development is a challenge. The number of test cases is growing over time, as new projects within a company are often based on previous software versions and reuse artifacts created for other projects [LLL+14]. Such artifacts can be requirements, software components, test cases

etc. But reuse can also introduce redundancy between artifacts. For example, assume a new version of the body comfort system if a new car model is developed. The previous project had several thousands of test cases, of which a lot can be reused. On the tooling side, the test cases are copied into the new project structure. But there is also new functionality to be tested. This leads to newly designed test cases, which are added into the projects structure. As a tester can not check thousands of old test cases for their exact description and test goal, it is possible that a newly described test case is covered by an already existing one. This can also easily happen if more than one test case designer works on the same subproject or functionality. If one now assumes the test cases are written in natural language as in Figure 2, redundancy is hard to detect. Within the BCS case study, there have been 97 requirements defined, for which we have defined 128 test cases in natural language [Beh13]. In interviews with industry experts, we found that automotive projects often comprise several hundred thousand requirements and test cases.

Redundancy can lead to highly inefficient testing as the *same* test case is performed several times without change leading to no novel results. One solution to tackle redundancy would be abandonment of reuse of test artifacts. This would force testers to write every test case from scratch. Thus, this is not an efficient approach due to the high number of test cases. We need a more intelligent approach, which can detect and reduce redundancy while still allowing for high reuse of artifacts. Therefore, test cases are processed for any reoccurring parts. This requires the test cases to be comparable. The approach shown in Section 2.1 builds a necessary foundation for the removal of redundancies in natural language test cases. If the test case specification is conform to a unified test specification, one can find similar parts within the test case structure over all test cases and identify redundant test cases. After the processing of the test cases, redundant test cases can either be marked and, if desired, removed. When writing a new test case, the test designer could be directly warned when his test cases is very similar to already existing ones. Another usage scenario is the merging of different projects. Here, redundancy could also be automatically detected and the respective test cases would be represented by one test case.

3 Implementation and Execution

After all test cases have been described, they have to be executed. It is also possible that the high-level test cases have to be implemented or additional test scripts have to be written. Even if the test cases are systematically described and free of redundancy, one has to find a trade-off between the potentially high number of designed test cases versus the very limited amount of available testing resources. The main challenges are the selection of test cases to be executed, and the ordering of these test cases. Test case prioritization [RUCH01, ZHZ+13] is necessary, as usually not all selected test cases can be executed because test cases have to be performed manually. For example, in the BCS case study, the finger protection has to be tested with human input, as an object has to be placed within the window. In test case prioritization, more important test cases are tested before less important ones. Therefore, some testing goals can be met earlier, which leads to benefits, especially, if not all selected test cases can be executed. Test case selection

and prioritization becomes even more challenging in the automotive domain since the source code of the system under test is unknown. Such black-box test scenarios offer a limited amount of information, e.g., about coverage metrics, than available in white-box techniques.

3.1 Effective test case selection and prioritization in black-box testing

A repeating effort has to be made to select the *right* test cases for test case execution. The first question, which needs to be answered, is what a good test case actually is. Kaner [Kan03] investigated this question and describes the quality of test case using several factors. In general testing, a good test case can be defined according to several distinct criteria, e.g., a test has a high likelihood to find a bug, it tests functions that are more likely to be used in reality, etc. Therefore, for creating test suites it is important to select those test cases, which fit the current test goal. The selected set of test cases defines a *test suite*, which then will be executed by the testers.

Assuming high-level test cases in a black-box testing scenario, the creation of a test suite usually is a manual task, which involves a lot of expert knowledge and experience regarding the system under test. This leads to two main problems. First, the knowledge of the test suite designer is often lost when the person leaves the project or company. Thus, the test selection is not exactly repeatable by others. It can be a very time consuming task to develop the necessary skills to create ”good” test suites. Second, manual selection is inefficient regarding the distribution of test cases, the time to select them and the failures they will find. As the selection procedure is not formally defined, results are not predictable, but only by the personal estimation of the test suite designer. Tool support could help to address test suite creation in a more structured and systematic way. Inexperienced testers could be guided to select and order test cases in test suites with the goal to test those first, which are more likely to expose failures.

To enable this, we introduce a greedy test case prioritization and selection technique, which is suitable for black-box testing. It requires a defined set of test cases and a set of selection/prioritization criteria as input. The result is a ordered list of test cases. A schematic description for this approach is shown in Fig. 3. The approach uses a given test suite, for which the test cases are ordered in some arbitrary way, e.g., alphabetical or after the date of creation. A set of predefined criteria is selected, such as the execution time, priority, tested functionality or history of the test cases. For example, the execution time of test cases could be used, to select those, which are executed fastest. This would be useful in a very time restricted environment. Another possible criterion is the test case history, which encompasses the number of failures found by a test case and their priority. Such a criteria is useful in a regression testing scenario, where test cases have to be executed again after some software update. Previously failed functionality should be tested again. A greedy ordering algorithm computes distinct ordered test suites for every given selection criterion. The best test case with respect to the selected criterion is always put first. To get a resulting test suite, the orderings are combined. To make this step more flexible for different project domains and project phases, the test suites can be combined based

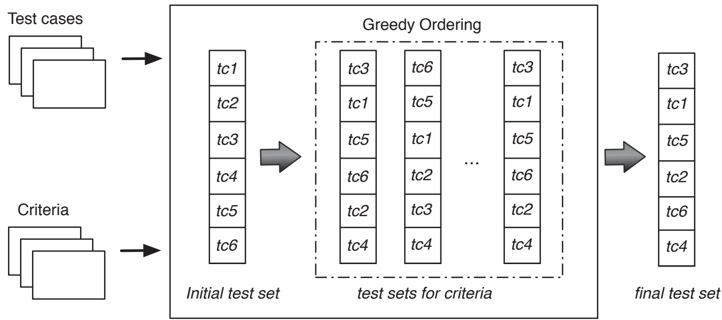

Figure 3: A greedy regression selection strategy

on some weighting function between the criteria. The weights for every criterion is either defined by the tester or can be automatically adjusted. For example, if a new software release within the BCS case study is to be released, where test case history is important, but the test resources are limited, the test case execution time and test history are used as criteria. The test history could be weighted with 75% and the execution time with 25% to impact the result. The resulting test suite is ordered in a form, that the first test case is the most important one, followed by the second most important one etc. Restricted in time and resources, the tester tries to test as many tests as possible in the given ordering.

This approach is very suitable for black-box testing in the automotive domain, as the defined criteria do not require any knowledge about the software and the source code itself, but only depend on the test cases. This solves the issue of missing code metrics, which would be primarily used in white-box testing. It is not guaranteed to find the optimum of selected test cases, but this approach could be used as a fast way to find a rather good result. A key factor for the success of this approach are the defined criteria and their weights.

3.2 Test case combination

One of the most common restrictions in testing is the available time. One possibility to increase the number of executable test cases under time restrictions is a combination of test cases. Similar to the approach described in Section 3.1, the ordering of test cases can lead to increased testing efficiency. Executing important test cases first is only a good guess, as failures are hidden and unknown both to tester and his tool. Thus, reducing the test execution time could lead to improved results compared to manual test selection. This can be achieved by ordering test cases according to their given pre- and post-conditions. For the BCS case study, there are two test cases for the feature *power window control*. This feature allows the user to move the power window using buttons on the remote control key. The test cases are specified as shown in Figure 4. One can see that these two test cases can be connected to a logical sequence, where first the window is moved up and then moved down. The post-condition of the first test case is the pre-condition of the second.

Name:	03_RemCon_Up
Precondition:	Central Locking System and automatic power window is build in. Power window is closed.
Action:	Press button "window up" on remote control.
Expected Result:	Power window moves up, until top position is reached (closed).

Name:	04_RemCon_down
Precondition:	Central Locking System and automatic power window is build in. Power window is closed.
Action:	Press button " window down" on remote control.
Expected Result:	Power window moves down, until bottom position is reached (open).

Figure 4: Two example test cases for control power windows

While the given example is rather trivial, finding such structures and coherences in several thousand test cases, which are written in natural language, is not. One possibility would be to investigate natural language terms and sentences to find possible combinations of test cases into test sequences. This requires a uniform test case specification as described in Section 2.1. Another possibility is to use model-based testing approaches [UL07] in order to generate test sequences from several test cases. State machines have proven to be a suitable method to represent executable system specifications in form of test models and to derive test cases for the modeled system. As benefit, transitions between states are clearly labeled and can be compared easily. Graph-based algorithms allow for fast traversal and comparisons within large sets of test models. Although model-based testing is an widely discussed and explored topic, one familiar problem is the lack of such models in industrial context. This has led to different approaches for *learning* test models based on given test cases [SM12, RSM08]. One can analyze these models and see what parts of the model are covered by which test cases. Therefore, it is possible to combine test cases,where the final state of one test case is the initial state of the other. Furthermore, thelearned models can be used to derive new test cases for the system. The better the model, the more detailed and distinct test cases one can generate.

4 Evaluation and Reporting of Testing results

After implementing and executing test cases, the test results have to be evaluated. Without evaluation, no knowledge can be extracted, and no new information is gained. Besides, the adaption of the testing process and the test cases is important. This incorporates the change of test case selection, test case prioritization, resource allocation etc. In regression testing, Kaner [Kan03] defines the goal to write *reusable test cases*. A good regression test case should be likely to fail, if changes induce errors in the tested area of the program under test. In regression testing, those test cases are selected for execution that are most likely to find failures introduced by change to an already tested system. This process is currently highly based on expert knowledge and the ability of testers to have insights into current test status. In the following, we show two possible approaches based on machine learning (ML) to introduce automation into this process.

4.1 Machine learning in regression testing

High-level test cases are often annotated with meta-data, e.g., the priority of test cases, tested functionality, test execution history. The meta-data can be used for test case selection and prioritization, as described in Section 3.1. For example, within the BCS case study one could argue that a test case regarding the basic functionality of an alarm system has a higher priority than a test case which tests the respective LED for this feature. The meta-data is defined by the test designer and is expensive in its creation, as it is not a trivial task to decide, e.g., what the priority of a test case is. This problem increases with every test iteration as associated meta-data should change over time, as the system under test does heavily change (e.g., bug fixes, new features, etc.). This leads to the question how meta-data can be changed after a test campaign without relying on manual intervention and expert knowledge.

One possible solution to avoid manual intervention when adapting the meta-data after test campaigns is to apply machine learning techniques (ML). ML has been already introduced to software testing [Bri08,LPV13]. ML approaches identify patterns in data, which can be used to derive new knowledge, and can be applied for decision-making. ML techniques can analyze the test case specification, requirements, test results and available testing history. Based on the findings, an ML approach can adapt the meta-data associated to the test cases for the next test iteration and selection process. For instance, one could envision a classifier, which decides if a test case is likely to reveal a failure or not and, hence, assign it a higher priority for a next test execution.

Figure 5 illustrates an example of the BCS study. The represented test case has been described in Figure 2. Let's assume, that the test case has been selected for a test suite and was executed. The test case revealed an important failure. The automated ML approach decides, that the priority of the test case has to be changed from the previous value *2* to *4* as the failure bug fix has to be verified. The execution time has been updated accordingly to the new result.

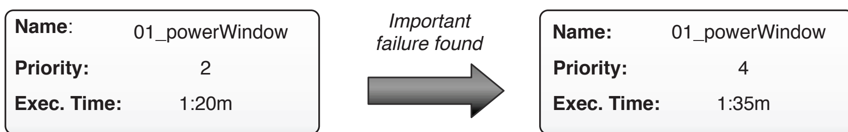

Figure 5: Adaption of black-box meta-data after change

4.2 Automatic regression test criteria selection

Test case selection for regression testing is a recurring task, which is often done manually. As described in Section 3.1, one can introduce a concept, where specific selection criteria are used to select an ordered list of test cases. The described method still needs human input, as the meta-data assigned to the test-cases for selection, the selection criteria and

the corresponding weights have to be provided by the tester. As shown in the previous section, machine learning techniques can be used to adapt meta-data in a black-box testing context. After each test run, however, the tester still would have to decide, how he has to alter the previously used criteria and their weights.

Therefore, we propose an extension to the approach of Section 3.1 in order to make it more automated. Instead of manually defining selection criteria, the extracted data from previous test runs can be used to adapt the selection criteria and weights for the next test campaign. For example, assume there is a defined testing criterion that a certain feature, e.g., the finger protection, is always tested. Within the latest test campaign, no failures have been found for this feature. Subsequently, the test system will automatically deselect this selection criterion or assign it a lower weight in the next run. This leads to a very flexible testing approach, where the tester is guided in his or her decision making, but can still make manual adjustments.

5 Related Work

In this section, we review related work in the areas covered in this paper, i.e., black-box testing in general and the specification of natural language test cases, test case selection and prioritization and machine learning-based testing.

Unsystematically specified test cases can lead to a high effort in introducing automation. Barros et al. present controlled natural language for use cases [BNHT11]. Strobbe et al. defined the *Test Case Description Language* in form of an XML-structure to describe test case meta-data [SHVV06]. This restricted languages help to create comparable and processable test case specifications. They also lay the foundation for possible tooling.

Test case selection and prioritization has been covered in the literature in the context of regression test selection and efficient testing of software product lines. Evolutionary algorithms have shown to be very convenient in prioritizing test cases [FKCA12]. Regression testing leads problems in selection and prioritization of test cases. Engström et al. [ERS10] have conducted a systematical review of empirical evaluations of regression test selection approaches. They notice that a comparison of the reviewed 28 techniques is rather hard, as they depend on many factors, which also influence the respective evaluations. Some examples for such factors are the granularity of the techniques, how frequent the regression cycles are or the products under test.

Software product lines are very relevant in the automotive context, as cars can be configured by the customer in many ways. Engström and Runeson [ER11] surveyed testing techniques for software product lines. They have identified three main research challenges in this field: (i) A (too) large number of tests, (ii) the trade-off between reusable components and the concrete products and (iii) treatment of variability. Cmyrev and Reißig [CR13] introduced a test case reduction technique for variant-rich systems, where a subset of product variants are selected while guaranteeing a complete requirements coverage. This approach is based on a greedy selection strategy and the connection between features of the product line and its requirements. The selection tries to find a minimal set of product configu-

rations while still covering 100% of the defined requirements. Johansen et al. present an algorithm to generate t-wise covering arrays for large feature models [JHF12]. This allows for combinatorial interaction testing of software product lines.

Briand [Bri08] has given an overview of machine learning approaches in software testing. He defines the role of ML in software testing and states that it can help to solve some of the long-standing software testing problems. One of the presented approaches is called MELBA (*MachinE Learning based refinement of BlAck-box test specification*) which is introduced as an ML based approach, which helps to refine test suites in black-box testing [BLB08]. It uses an incremental process, which is semi-automated. The approach helps to understand limits of test suites and to find redundancies. It does not, however, sort test cases in any order or make predictions about test case outcomes.

While the above works focus only on a single approach, this paper gives an overview of different techniques over three phases of the testing process to make testing in the automotive domain more effective and efficient.

6 Conclusion and Future Work

In this paper, we gave an overview of some challenges in automotive software testing. We focused on black-box testing scenarios, as source code often is not available in an OEM-supplier scenario in the automotive domain. Following the general testing process, we have presented six approaches to improve test efficiency considering the constraints of automotive software development. Some of the proposed concepts are already realizable, e.g., the systematic test case description. Others need some future work to be of practical use, e.g., machine learning approaches in black-box testing.

References

[Beh13] M. Behr. Evaluation eines anforderungsbasierten, delta-orientierten Testverfahrens für Softwareproduktlinien. Bachelor thesis, TU Braunschweig, 2013.

[BLB08] L.C. Briand, Y. Labiche, and Z. Bawar. Using Machine Learning to Refine Black-Box Test Specifications and Test Suites. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 135–144, Aug 2008.

[BNHT11] F. A. Barros, L. Neves, E. Hori, and D. Torres. The ucsCNL: A Controlled Natural Language for Use Case Specifications. In *SEKE*, pages 250–253. Knowledge Systems Institute Graduate School, 2011.

[Bri08] L.C. Briand. Novel Applications of Machine Learning in Software Testing. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 3–10, Aug 2008.

[CR13] A. Cmyrev and R. Reißig. Optimierte Varianten- und Anforderungsabdeckung im Test. In *Automotive Software Engineering Workshop, 43. GI Jahrestagung*, 2013.

[ER11] E. Engström and P. Runeson. Software product line testing – A systematic mapping study. *Information and Software Technology*, 53:2–13, 2011.

[ERS10] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52:14–30, 2010.

[FKCA12] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba. Evolutionary Algorithm for Prioritized Pairwise Test Data Generation. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 1213–1220. ACM, 2012.

[JHF12] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC*, pages 46–55, 2012.

[Kan03] C. Kaner. What is a good test case? In *Software Testing Analysis & Review Conference (STAR) East*, 2003.

[KTS10] J. Kasurinen, O. Taipale, and K. Smolander. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, 2010:571–579, 2010.

[LLL+14] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, and U. Goltz. Delta-oriented model-based integration testing of large-scale systems. *The Journal of Systems and Software*, 91:63–84, 2014.

[LLLS13] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study. Technical report, TU Braunschweig, 2013.

[LPV13] A. R. Lenz, A. Pozo, and S. R. Vergilio. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence*, 26(5–6):1631 – 1640, 2013.

[LS13] R. Lachmann and I. Schaefer. Herausforderungen beim Testen von Fahrerassistenzsystemen. In *11. Workshop Automotive Software Engineering (ASE)*, 2013.

[RSM08] H. Raffelt, B. Steffen, and T. Margaria. Dynamic Testing via Automata Learning. In *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'07, pages 136–152. Springer-Verlag, 2008.

[RUCH01] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on software engineering*, Vol.27 No.10:929–948, 2001.

[SHVV06] C. Strobbe, S. Herramhof, E. Vlachogiannis, and C. A. Velasco. Test Case Description Language (TCDL): Test Case Metadata for Conformance Evaluation. In *ICCHP*, pages 164–171, 2006.

[SL12] A. Spillner and T. Linz. *Basiswissen Softwaretest - Aus-und Weiterbildung zum Certified Tester*. dpunkt.verlag, 2012.

[SM12] M. A. Sindhu and K. Meinke. IDS: An Incremental Learning Algorithm for Finite Automata. *CoRR*, abs/1206.2691, 2012.

[UL07] M. Utting and B. Legeard. *Practical Model-based Testing*. Morgan Kaufmann, 2007.

[ZHZ+13] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the Gap between the Total and Additional Test-Case Prioritization Strategies. In *International Conference on Software Engineering, ICSE 2013*, 2013.