Chapter 22
Testing Autonomous and Highly Configurable Systems: Challenges and Feasible Solutions

Franz Wotawa

22.1 Introduction

Quality assurance has always been an important topic in the automotive industry leading to standards like ISO 26262 [1] where automotive safety integrity levels (ASILs) are defined based on hazard analysis and risk assessments. Often failure mode and effect analyses (FMEAs) are used to identify potential hazards occurring when particular system components fail. Alternatively, such systems might be analyzed accordingly to Leveson [2] where the author focuses mainly on a system theoretic perspective. In order to assure that the implementation does never lead to such dangerous situations, tests have to be carried out. Although the automotive industry has spent a lot of effort in quality assurance, there has been a lot of vehicle recalls during the past years, many of them caused by software bugs detected after deployment. As discussed by Altinger et al. [3], such recalls also have a huge economical impact. When considering that even today’s quality assurance measures used in the automotive industry cannot prevent recalls, the question arises about consequences for safety of autonomous driving vehicles. For this purpose we discuss first the differences between ordinary cars and autonomous driving vehicles and second identify means for dealing with these differences.

Autonomous systems like autonomous mobile cars or robots have to provide a certain task, for example, driving from one place to another, while interacting with their surrounding environment. This interaction is based on observations such systems obtain from the environment using sensors and their internal current state from which actions are derived. These actions are executed using the attached actuators. In the case of autonomous driving vehicles, the information about the current position, the speed, other vehicles, and obstacles are obtained using attached...
sensors like GPS, laser range finders, or vision systems. Given this information and the planned journey, such an autonomous vehicle decides on the next step like changing the lanes on a highway or initializing an emergency braking action. All such decisions may harm others, and thus such systems have to be thoroughly tested to assure that all safety critical requirements are fulfilled. This situation becomes worst in case of self-adaptive systems like systems that learn during operation or systems having a huge configuration space. It is interesting to note that even in the case assuming that all such systems behave deterministically from the point of view of the implemented algorithms and behaviors, there might be cases where small deviations of the measured input signals have a huge impact to decisions. Thus, from outside such a system might look like as behaving non-deterministically.

In Fig. 22.1 we depict the main difference between classical cars, vehicles with assistive technology, autonomous driving vehicles (ADVs), and ADVs with adaptive technology. Whereas ordinary vehicles and vehicles with assistive technology always have an external supervisor in the loop, i.e., the driver, there is no external supervision in case of ADVs and adaptive ADVs. Hence, any failure occurring during driving has to be detected by the ADV itself. Furthermore, appropriate countermeasures have to be taken into account immediately in order to prevent hazardous situations. For this purpose the ADV has to have access to necessary information from which a failure can be deduced. This includes not only the internal state of the vehicle itself but also its surrounding environment.

When allowing systems more and more autonomy, the question arises of how to ensure a proper behavior under all circumstances? Due to variations of sensory inputs, changes in the environment, e.g., change of lighting conditions, faults that might occur, or different configurations, the whole space of possibilities grows very quickly and it is very likely to not consider certain cases during development. For example, there are a huge variety of different traffic signs around the globe. Some of them are outdated but might be still found. Hence, the vision system of an
autonomous vehicle responsible for detecting and classifying traffic signs has to be tested using many different versions of signs. Even worst the signs might be damaged, the color might not longer be the originally specified color, and so on. However, in any circumstances the autonomous vehicle has to react appropriately. From the stop sign example, we are also able to see that the variants depend on the region, which allows us to specify a certain context, e.g., the country where an autonomous vehicle is going to operate, in which certain rules, assumptions, or properties are valid.

In this chapter we assume that the subcomponents of the whole system already have been thoroughly tested and focus on the integration and system test. There we are in particular interested in assuring important requirements of autonomous vehicles under all circumstances. We will discuss the influencing parameters, which also entail consequences on how to test such systems. Depending on the degree of autonomy, we will see that testing during development might not be sufficient and thus require introducing monitoring at runtime for assuring meeting requirements like safety.

In the following we discuss related research of testing where we specially focus on autonomous adaptive systems. Afterward, we discuss the underlying problem and propose a testing method that takes the huge variety of configurations, parameters, and situations into account. Finally, we conclude the paper.

22.2 Related Research

Validation and verification (V&V) is an important part within the system development process. Under validation we subsume all activities necessary to check whether a developed system is the system that was indented to be developed, whereas verification is for checking that the system follows its specifications. Testing can in principle cover both parts of V&V. However, it is obvious that testing is only for detecting failures but not for ensuring that there are no faults in the system. Nevertheless, testing is currently the most important activity in V&V. In Fig. 22.2 on the left, the V process is depicted that is often used as a standard process in the automotive industry. In this process the different phases of the process like requirement analysis are paired with their corresponding testing activities.

For a general introduction into testing, we refer the interested reader to [4] and [5]. Testing as an activity generally speaking deals with search for interactions between the system under test (SUT) and a user in order to reveal an unexpected behavior. In general there might be an unfeasible number of potential interactions. Hence, testing as a discipline tries to identify the most relevant interactions with a SUT. Relevant in this context mean to cover sequences of interactions that if fulfilled assure that the SUT works as originally indented.

In order to reduce the testing effort, the automation of testing has been in the focus of research and development for several decades. Test automation comes with two favors: (1) automation of test execution and (2) automation of the generation
of interaction sequences. The former requires the implementation of frameworks that allows to program sequences of interactions with the SUT directly. The latter deals with the automated extraction of interaction sequences. This can be done from information about the input domains or models of the SUT. In case of model, we speak about model-based testing (MBT), which has gained a lot of attention.

For an introduction of MBT, have a look at [6] or [7]. A model in the context of MBT is usually a state-space representation of the relevant parts of a SUT from which interaction sequences can be obtained via traversing the state space. In addition MBT also allows for checking whether the output generated by the SUT when stimulated with an extracted interaction sequence follows the expectations. Hence, there is no need to have an additional oracle, which allows for full test automation. Test automation including the use of models is very much in use within the automotive industry. For a detailed discussion on testing methods used, we refer to [3] where the authors present the results obtained from a questionnaire survey carried out in the context of automotive system development.

In the context of autonomous and self-adaptive system, quality assurance has become more and more important. Cámara et al. [8] compiled several articles dealing with methods for quality assurance of self-adaptive systems that are also of particular interests for ADVs. The main methods further outlined in [8] make use more or less of monitoring and runtime verification to identify critical situations to be handled directly whenever they occur after deployment of such self-adaptive systems. Nafz et al. [9] introduced the concept of allowed deviations from the expected behavior and propose a method that allows for guaranteeing that self-adaptive systems always do not exceed these boundaries. Steinbauer and Wotawa [10] described methods that make use of model-based reasoning to adapt in case of
internal faults or changes in the surrounding environment. There the underlying idea is to reason from the models of the system and the environment to find a current configuration of the system that explains the current observations. In this way the given models describe more or less the behavioral boundaries of the system and the degree of adaption. In other chapters of [8], authors make use of runtime verification for identifying potential hazards. In runtime verification formal models of properties that are checked during operation are used to identify potential behavioral deviations. For more information about runtime verification and its use, consult [11] or [12].

In contrast to the mentioned previous related research, the methodology proposed in this chapter focuses on identifying potential hazards that might occur during operation at development time. We will see that such an endeavor requires dealing with multiple configurations and a huge parameter space in addition to dealing with the oracle problem in an appropriate manner. The oracle problem in this context can be solved when formalizing operational requirements and properties. For carrying out the tests, simulation environments have to be used for automation.

### 22.3 Problem Definition

Quality assurance of ADVs or adaptive ADVs has to consider two important issues. One is due to the fact that the ADV after delivery makes decision without human operators in the loop and thus requiring assurance that no unsafe decisions are taken at any circumstances. Therefore, testing has to consider all possible situations, which lead to a huge parameter space to be handled. Inputs for testing are formal descriptions of properties that have to be fulfilled always, e.g., safety requirements, the ADV configuration, external conditions, as well as internal faults that might occur during operation and that should be handled appropriately.

In Fig. 22.3, we depict some of the influencing parameters that fall into one of the three categories, i.e.:

- ADV configuration
- State of the ADV
- External conditions influencing the behavior of the ADV

All these categories have a lot of different parameters. An electrical ADV may have configuration parameters for the engine used, the availability of an air conditioner, the driving mode, and the used battery, all of them influencing the driving distance, the driving experience, and the functionality. The internal state of the ADV comprises information about internal faults, e.g., a broken wire, or even wrongly perceived external objects, all of them influencing the ADV’s behavior. External environmental conditions include the current weather or light condition and also the context information, e.g., the country where an ADV is used. Here we might have to consider, for example, different stop signs during operation.
When considering $n$ parameters each being able to take $k$ values, we finally would have to consider $k^n$ different combinations during testing. This is not feasible in practice and cannot be done within a reasonable amount of time. When assuming only two different values for all parameters and in total 100 different parameters falling in one of the three categories, we would require $2^{100} \approx 1.27 \times 10^{30}$ different tests. Assuming that each execution of one test needs one second, we need more than $10^{22}$ years to finalize testing. Hence, there is a strong need for a testing method requiring less tests but still assuring high quality.

Testing ADVs is somehow different from testing ordinary cars. Even in the case that there is also a high number of potential configurations and faults to be considered, we are able to test the different parts of the vehicle separately before performing an overall system test. In addition, because of knowing that there is a driver in the loop supervising the overall vehicle, testing can be more focused. This cannot be done in case of ADVs. Therefore, we have to consider both a high number of potential configurations and different situations that are influenced by the environment and its corresponding conditions. In the next part of this chapter, we introduce a solution to the testing challenge of ADVs that allows for keeping the testing overhead as low as possible. Besides complexity reduction the approach can also be fully automated.
22.4 Combinatorial Testing for Autonomous Adaptive Systems

In order to solve the testing problem for ADVs, we have to provide a solution that reduces the number of test cases and is also able to solve the oracle problem. The oracle problem in testing is the problem of classifying the output of a test as being correct or faulty. In case of MBT, the used models handle the oracle problem directly. In other test case generation approaches where the aim is to provide a set of input stimuli, solving the oracle problem is an important task for automating test execution. In the following we first describe how to solve the problem of generating test inputs. Afterward, we discuss how to solve the oracle problem, and finally, we bring all these parts together to provide a general and feasible solution.

22.4.1 Combinatorial Testing

Within the past decade researchers thought about the question of how to generate test inputs for highly configurable products or products having a large input space in a feasible way. For example, when providing software for mobile phones, someone has to take care of the different hardware devices as well as the different releases of the operating system among other important parameters like special settings of the phone network provider, the configuration of the software under test, or other software installed that might interact. Any combination of such parameters or factors might be relevant for revealing a faulty behavior. However, this cannot be done. Hence, someone thinks about the question whether combinations of two, three, or more parameters might be sufficient for fault detection.

From this deliberation researchers come up with combinatorial testing where only combinations of a fixed number of parameters are considered for test case generation. In Kuhn et al. [13], the authors give an introduction into combinatorial testing. For being self-contained, we briefly discuss the underlying ideas behind combinatorial testing. The basic idea is that a combination of values for \( t \) parameters is enough to cause the system to be executed in a faulty way that finally leads to an observable failure. Such a fault is called \( t \)-way interaction fault. Instead of considering all combination of parameters, we only consider all combinations for \( t \) parameters, which obviously reduce the number of generated test cases substantially. Another important finding is that the number of parameters \( t \) to be considered needs not to be too high to reveal all faults in software. Kuhn et al. [14] showed that for a larger variety of different programs, at most 6-way interactions have been enough to detect all faults. In many cases a lower number of interactions is sufficient to detect most of the faults. Hence, combinatorial testing is a good method for reducing the test suite size while keeping the failure detection rate high, which is a prerequisite for testing ADVs in practice.

In the following we formalize the concept of \( t \)-way combinatorial tests. We assume that we have a model of the input space comprising a set of parameters \( P = \{P_1, \ldots, P_n\} \), where each parameter \( P_i \) has a domain \( D_i \), i.e., a set of values.
A test suite \( TS \) is a set of tuples \((v_1, \ldots, v_n)\) assigning a value \( v_i \in D_i \) to each parameter \( P_i \). Such a test suite \( TS \) is a \( t \)-way combinatorial test suite if and only if the following hold: For any selection of \( t \) parameters, all combinations of values for these parameters are represented in \( TS \). Note that there might be larger parameter sets in a \( t \)-way combinatorial test suite where all combinations are represented. Hence, a 2-way combinatorial test suite might capture 3-way interaction faults for some (but not for all) parameters.

Given a model of the input space and the parameter \( t \), there are algorithms that compute all input tuples, i.e., the test cases, where the condition for \( t \)-way combinatorial tests hold. The ACTS combinatorial testing tool [15] developed jointly by the US National Institute Standards and Technology (NIST) and the University of Texas at Arlington can be used for this purpose. It is worth noting that ACTS also allows for specifying constraints among the input parameters. For example, when adding an air conditioner into a car, the car requires a larger battery. This information restricts the potential combinations. In a combinatorial test suite with constraints, all tuples not fulfilling constraints are removed, and some new might be added such that the condition for combinatorial test suites is always fulfilled.

The selection of the parameter \( t \) for combinatorial testing influences the number of interaction faults to be detected. In Kuhn et al. [14], the cumulative error detection rate reported ranged from almost 75 to more than 90\% for 2-way interaction faults and from 87 to 98\% when setting \( t = 3 \). Hence, in practice the value of \( t \) has to be chosen having the error detection rate into mind. It is worth noting that given practical requirements like a fixed budget for testing, a deadline for finalizing testing, or other restrictions, it is very unlikely to detect all faults during system development. Therefore, there is always a trade-off between those requirements and the number of tests to be carried out, which of course has consequences on the error detection rate. Whether a certain reduced error detection rate is acceptable or not relies on the application domain and within the domains on given standards and regulations.

Let us take the parameter space depicted in Fig. 22.3 and formalize the input parameter space such that we are able to extract a test suite. We take the car configuration as part of the example from [17]. There the electrical motor \( emot \) can take the two values \textit{standard} or \textit{powerful}. The air conditioner \( ac \) might be \textit{none}, \textit{manual}, or \textit{electronic}. The driving mode \( dm \) can be \textit{leisure} or \textit{race}, and the battery \( bat \) has the domain \{\textit{type1}, \textit{type2}, \textit{type3}\}. For this example, we ignore faults occurring. For the environmental conditions, we assume street conditions \( strcond \) to be \textit{highway}, \textit{city}, or \textit{country} road and three different \textit{stop} signs \textit{stop1}, \textit{stop2}, and \textit{stop3}. The following table summarizes the parameters and their values:
When using ACTS for computing a combinatorial test suite of strength 2, we obtain the following table:

| emot    | ac     | dm   | bat  | strcond | stop |
|---------|--------|------|------|---------|------|
| powerful| none   | race | type1| city    | stop2|
| standard| none   | leisure| type2| country | stop3|
| powerful| none   | leisure| type3| highway | stop1|
| standard| manual | race | type1| country | stop1|
| powerful| manual | leisure| type2| highway | stop2|
| standard| manual | race | type3| city    | stop3|
| powerful| electronic| leisure| type1| highway | stop3|
| standard| electronic| race | type2| city    | stop1|
| powerful| electronic| race | type3| country | stop2|
| standard| manual | race | type2| highway | stop2|
| powerful| electronic| leisure| type3| city    | stop3|

Instead of 324 test cases, which are all different possible combinations, the 2-way combinatorial test suite for the small example only comprises 11 test cases. A 3-way combinatorial test suite comprises 33 test cases and a 4-way one 83. The latter is still only one fourth of the total number of combinations, which saves a lot of testing effort while still being able to reveal potential faults.

For smaller input models used for obtaining a \( t \)-way test suite and a smaller number of \( t \), the whole test suite generation time using available tools can be neglected. In Yu Lei et al. [16], the authors introduce algorithms that allow for computing 5-way tests for 20 parameters each having 4 values in less than a minute with their fastest algorithm leading to a test suite comprising 8606 elements. Note that the number of all possible combinations in this case is \( 4^{20} \times 10^{12} \) and thus we obtain a much smaller test suite. However, depending on the test execution time, the number of test cases might be too high to meet given deadlines. In addition, for large values of \( t \) and a large domain of parameters, test case generation most probably takes too long when using current combinatorial testing algorithms. In such cases the combinatorial testing problem has to be partitioned into subproblems, to be handled independently. In each subproblem we take certain parameters and set them to a fixed value. The remaining parameters form the subproblem for which we are able to generate a combinatorial test suite. The selection of parameters can be done considering their likelihood for revealing an interaction fault. If two parameters are likely to not interact in an unwanted manner, their values can be fixed. Such information has to be obtained from domain experts.

When starting a test using the tuples specifying combinations of parameters, we are able to carry out specific parameterized tests. However, what is still missing is the question whether such test execution fails or passes. We discuss this issue in the following section.
22.4.2 Test Oracles

Carrying out a particular test case requires the execution of the SUT using the specified parameter values. The SUT execution in our context may require carrying out a simulation environment taking care of the specified parameters. The question now is how to distinguish a faulty behavior from a correct one in the simulation? One answer would be to have a user observing the simulation results and classifying a particular simulation run as passing or failing. However, this is not a practical approach because of the larger number of tests to be carried out. Hence, we need to automatize the test oracle.

One way of automatizing the test oracle is metamorphic testing (see [18] and [19]). The idea behind metamorphic testing is to utilize symmetries in functions or systems to be tested. For example, it is well known that \( \sin(x) \) is equivalent to \( \sin(x + 2\pi) \). Hence, we use the constraint \( \sin(x) = \sin(x + 2\pi) \) as an oracle. Whenever this constraint is not fulfilled, we know that the test is a failing test. In general, we use all constraints that represent symmetries to form the test oracle. In the context of ADVs, we also might be able to define such symmetries and to use them as test oracles. However, metamorphic testing may require carrying out the test twice. For example, when checking whether \( \sin(x) \) is equivalent to \( \sin(x + 2\pi) \), we have to call the \( \sin \) function twice. This would increase testing effort.

Another possibility, which is more appropriate in the context of ADVs, originates from the basic ideas behind runtime verification [11] where formal properties are checked when executing the SUT. This idea can be easily translated into our domain when formalizing safety properties and other requirements. For example, in none of the cases, we want an ADV crashing into an obstacle. Such a requirement can be represented formalized and also later on represented programmatically in the simulation environment. There the functionality of the simulation environment can be used, for example, to detect overlaps between the ADV and an obstacle and to determine that the crashing property is violated.

Hence, we assume that we are able to implement an oracle function that checks for safety property and requirement violations. Such an oracle function has to be closely integrated into the simulation environment carrying out the generated combinatorial test suite. It is worth noting that such an oracle is not always able to classify a test as passing or failing. In some cases the outcome might be inconclusive. For example, stating that a property must always hold cannot be proven within a limited simulation time. Only in cases where the property is violated we exactly know that the test is a failing test. Otherwise, there might be an interaction between the ADV and its environment not experienced so far that would be able to violate the property.
22.4.3 The Automated Testing Methodology

When combining the combinatorial testing approach with the automated testing oracle comprising information about properties, requirements, and metamorphic relations, we are able to come up with testing methodology for ADVs that can be fully automated. For this purpose we have to have an execution environment, e.g., a simulator, where each test can be carried out and where the testing oracle can be integrated. In the following we describe the proposed automated testing methodology in more detail. In Fig. 22.4, we give an overview of the testing methodology comprising the following three parts:

1. **Test case generation**: For the purpose of generating test cases, we assume that we know all parameters and their domains for the ADV configuration, e.g., the kind of battery or engine, the environmental conditions, as well as faults that might occur during operation. In addition we might also know some constraints between parameters that might limit potential combinations of parameter values. From the parameters, their domains, and the optional constraints, we use a tool (e.g., ACTS [15]) to generate a \( t \)-way combinatorial test suite where \( t \geq 2 \). The resulting test suite comprises tuples of values for each parameter that work as input for the test execution.

![Fig. 22.4 Overview of the automated testing methodology for ADVs](image-url)
2. **Testing oracle**: For generating the testing oracle, it is essential to have knowledge about safety property and other functional and nonfunctional requirements of ADVs. This knowledge has to be formalized. The formalized knowledge is used similar to runtime verification [11] to check the current state of the ADV during test execution. Here the idea is to implement a function `check(.)` that takes the values observed from execution and checks whether a failure occurs. For implementing `check(.)`, we rely on previous work that has already been done in the context of runtime verification. After implementing `check(.)`, we obtain a testing oracle that automates the task of judging the outcome of a test applied to the SUT. Note that `check(.)` returns not only *pass* or *fail* but also *inconclusive* in cases where properties cannot be finally verified because of restrictions due to the finite execution time.

3. **Test execution framework**: The third part of the framework for automating the system test for ADVs is the execution framework. Here we assume that the input from the test case generation component, i.e., a tuple of parameter values, configures a simulation run of the SUT, which might be itself represented as a model, software, or hardware. Note that there is a difference between a model representing the SUT and the input model used to generate the combinatorial test suite. A model representing the SUT has to capture the system’s behavior, whereas the input model used for generating the test suite only comprises parameters, their domains, and an optional set of constraints. The execution framework executes this test and calls the oracle function `check(.)` each time new observations from the SUT are available. In case `check(.)` returns *fail*, the test is terminated and classified as failing test. In cases where the test is terminated due to given limitations on execution time, the `check(.)` function is called for finally classifying the test as *passing* or *inconclusive*.

The advantage of the proposed testing methodology is that the test generation and execution part can be completely automated. Moreover, combinatorial testing guarantees that important combinations of parameters with respect to their potential of detecting failures are considered during testing. This limits the number of combinations of test parameters substantially and makes the overall approach feasible even for a larger set of parameters and their corresponding domains. The disadvantage is that the input parameter space as well as the safety properties and other requirements has to be identified and formalized, which require effort. However, within the development processes usually used, the identification of such requirements is mandatory. Hence, the only additional effort is to formalize these requirements in order to allow utilizing runtime verification techniques directly. It is worth noting that the test execution environment has also to be extended in order to be able to cope with the testing oracle function. The required additional effort, however, should be insignificant.
22.5 Conclusion

In this chapter, we discussed the challenge of testing ADVs and adaptive ADVs. We identified the huge parameter space to be considered as one of the main reasons behind the complexity of testing ADVs. We also argued that ordinary vehicles even with attached assistive technology could be more easily tested due to the fact that we are able to define the scope of subsystems to be tested separately. Moreover, because of humans in the loop during execution, we have behavioral supervision available that further helps to restrict testing in this case. For testing ADVs we have to identify a method that (1) allows for reducing the testing parameter space while (2) keeping the potential failure detection rate high. In this chapter, we argue that combinatorial testing fulfills both criteria and make appropriate citations that support the used argumentation chain.

In addition to the generation of test cases, which are in our context the parameter values used as input to a simulation environment where tests can be carried out, we discussed the automation of the test oracle. A test oracle allows classifying tests as passing, failing, or inconclusive based on the given observations of variable values during test execution. In order to automate the test oracle, we suggested to use runtime verification that is based on safety properties and other requirements an ADV has to fulfill. Such information is always available in the context of vehicle development within the automotive industry. The properties and requirements have to be formalized in order to automate the test oracle.

The proposed testing methodology for ADVs combines the test case generation approach based on combinatorial testing with the automated test oracle based on runtime verification. For this purpose, we assume the availability of a test execution framework, i.e., a kind of simulation environment, which takes the tests as input and allows for integrating the test oracle. The proposed testing methodology is feasible because it requires only important combination of parameter values that capture faults that can only be revealed in cases of interactions between a fixed number of input parameter values occurring at the same time.

Considering only one testing methodologies for quality assurance is in general not a good idea, mainly due to differences in the objectives of the methodologies. Model-based testing aims at generating test suites in a rigorous and complete manner based on a model of the SUT. Combinatorial testing focuses on interaction faults that can only be revealed using the right combination of parameter values. Random testing tries to find interactions with the SUT that have not been foreseen and also interactions that are outside of the specifications. The purpose of manual testing is mainly on capturing possible interactions of humans with the SUT. The combined used of testing methodologies during development as well as other measures like coding guidelines is essential for quality assurance. An early use of the proposed testing approach within the ADV development cycles using models and software in simulation does not provide any guarantee to detect all faults. However, it would allow for detecting interaction faults earlier during development.
Acknowledgment. The research work has been carried out as part of the 3CCar project co-funded by the Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU) grant agreement number 662192-3car-ECSEL-2014-1 and the FFG grant agreement number 848715.

References

1. ISO 26262:2011: Road vehicles functional safety (International Organization for Standardization, Geneva, 2011)
2. N.G. Leveson, Engineering a Safer World (MIT Press, Cambridge, MA, 2011)
3. H. Altinger, F. Wotawa, M. Schürus, Testing Methods Used in the Automotive Industry: Results from a Survey. Proceedings JAMAICA’14, San Jose, CA, 21 July 2014
4. P. Ammann, J. Offutt, Introduction to Software Testing (Cambridge University Press, Cambridge, MA, 2008)
5. J. Glenford, Myers, The Art of Software Testing (Wiley, New York, 1979)
6. I. Schieferdecker, Model-based testing. IEEE Software 29(1), 14–18 (2012)
7. M. Utting, B. Legeard, Practical Model-Based Testing – A Tools Approach (Morgan Kaufmann, Burlington, MA, 2007)
8. J. Cámara, R. de Lemos, C. Ghezzi, A. Lopes (eds.), Assurances for Self-Adaptive Systems – Principles, Models, and Techniques. LNCS 7740 (Springer, Berlin, 2013)
9. F. Nafz, J. Steghöfer, H. Seebach, W. Reif, Formal Modeling and Verification of Self-Systems Based on Observer/Controller-Architectures, in LNCS 7740 [8] (Springer, Berlin, 2013), pp. 80–111.
10. G. Steinbauer, F. Wotawa, Model-Based Reasoning for Self-Adaptive Systems – Theory and Practice, in LNCS 7740 [8] (Springer, Berlin, 2013), pp. 187–213
11. M. Leucker, C. Schallhart, A brief account of runtime verification. J. Logic Algebr. Program. 78(5), 293–303 (2009)
12. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, R. Washington, Combining test case generation and runtime verification. Theor. Comput. Sci. 336(2–3), 209–234 (2005)
13. D. Kuhn, R. Kacker, Y. Lei, Introduction to Combinatorial Testing. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series (Taylor & Francis, London, 2013)
14. D.R. Kuhn, R.N. Kacker, Y. Lei, J. Hunter, Combinatorial software testing. Computer August., 94–96 (2009)
15. L. Yu, Y. Lei, R. Kacker, D. Kuhn, Acts: A combinatorial test generation tool, in IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013, pp. 370–375
16. L. Yu, K. Raghu, D. Richard Kuhn, V. Okun, J. Lawrence, IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing. Software Test. Verif. Reliab. 18, 125–148 (2008)
17. F. Wotawa, I. Pill, Testing Configuration Knowledge-Bases, in 16th International Configuration Workshop, 2014. http://ceur-ws.org/V ol-1220/06_confws2014_submission_13.pdf, pp. 39–46
18. T.Y. Chen, S.C. Cheung, S.M. Yiu, Metamorphic Testing: A New Approach for Generating Next Test Cases, Technical Report HKUST-CS98-01 (Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998)
19. T.Y. Chen, J. Feng, T.H. Tse, Metamorphic Testing of Programs on Partial Differential Equations: A Case Study, in Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC ’02), Los Alamitos, CA (IEEE Computer Society, 2002), pp. 327–333