2012 45th Hawaii International Conference on System Sciences

Effectiveness of Random Testing of Embedded Systems

Padmanabhan Krishnan
Centre for Software Assurance
Bond University
Gold Coast, QLD 4229, Australia
Email: pkrishna@staff.bond.edu.au

R. Venkatesh, Prasad Bokil, Tukaram Muske, Vijay Suman
Tata Research Development and Design Centre
54 B Hadapsar Industrial Estate
Pune-411013, Maharashtra, India
Email: {r.venky,prasad.bokil,t.muske}@tcs.com

***Abstract*—Embedded systems like those used in automobiles have two peculiar attributes - they are reactive systems where each reaction is influenced by the current state of the system, and their inputs come from small domains. We hypothesise that, because inputs come from small domains, random testing is likely to cover all values in the domain and hence have an effectiveness comparable to other techniques. We also hypothesise that because of the reactive nature long sequences of interactions will be important for testing effectiveness. To test these hypotheses we conducted three experiments on three pieces of code selected from an automotive application. The first two experiments were designed to compare the effectiveness of randomly generated test cases against test cases that achieve the modified condition decision coverage (MCDC) and also evaluate the impact of length of the test cases on effectiveness. The third experiment compares the effectiveness of handwritten test cases against randomly generated test cases of similar length.**

Our objective is to help practitioners choose an effective technique to test their systems. Our findings from the limited experiments indicate that random test case generation is as effective as manual test generation at the system level. However, for unit testing test case generation to achieve MCDC coverage is more effective than random generation. Combining unit test cases with system level testing increases effectiveness. Our final observation is that increasing the test case length improves the effectiveness of a test suite both at the unit and system level.

***Keywords*-automatic test generation, test suite evaluation, embedded systems**

I. Motivation

Testing is the most widely used method to detect bugs in automotive software. Broy et al. [1] describe challenges that are specific to the automotive domain. A major challenge in testing these types of embedded systems is the generation of effective test cases for the units, and more importantly, for the system at a reasonable cost.

Practitioners often try to achieve some coverage criteria [2] as a means to effective testing. The modified condition decision coverage (MCDC) scheme is considered a good criteria to test embedded systems [3] and is also mandated by standards such as DO-178B [4]. Our experiments too show this to be an effective criterion. It is, however, very difficult to generate tests that achieve MCDC at the system level. For instance, our automatic test data generator for the MCDC criterion, AutoGen [5], that uses the bounded model checker for C programs CBMC [6], only scales up to 5KLOC, and sometimes even lower in presence of complex computations. Though there have been attempts to improve the scalability of white-box techniques [7], [8], to the best of our knowledge, the scalability that can be achieved is still limited to much less than that required for industrial level software.

System test cases are typically generated manually, which is very expensive and induces a strong correlation between quality of the test and experience of the tester. In order to evaluate alternate techniques to generate test cases we conducted experiments that compared manual testing, specification based testing and testing using randomly generated values at the system level. This paper reports the findings of these experiments with a view to recommending *a specific* technique to practitioners. We also evaluate the impact on effectiveness of combining each of these with unit level testing. We assume that the unit tests satisfy the MCDC criterion.

We evaluate these techniques using mutation adequacy ratio [9]. Test effectiveness can also be measured using various coverage criteria [2] but since we are using the MCDC criterion in the test generation phase, we chose to use mutation adequacy ratio to measure the effectiveness of the generated tests. Mutation analysis has been shown to be an effective measure of a test suite's effectiveness [10], [9], [11].

Our key findings are that at the unit level a test suite that achieves MCDC coverage is more effective than a randomly generated test suite. Interestingly, arbitrarily increasing the length of the MCDC test cases with test vectors from the test case itself improves the effectiveness of the test suite.

Based on the above finding, we conducted our system test experiment with only long test cases. We found that randomly generated test cases are as effective as manually produced and also test cases generated from alternate specifications.

In the next section we review related work and present some concepts that are relevant to our work. In Section III we describe the experiments conducted along with the artifacts used. In Section IV we present the results of our experiments; we also analyse the results and draw

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.233

appropriate conclusions.

II. Related Work and Preliminaries

There is a vast body of knowledge that relate to testing. We review a few of the key results related to test automation.

Test automation (say via model based testing) of automotive systems has been researched [12], [13]. However they mainly focus on tests at the model level. Bringmann and Kramer [13] argue for the need for tests that can exercise complex interactive behaviour. The difficulty of the problem has been recognised in [14] which reports that none of the three particular strategies (random, adaptive random and search based) examined was better than the others. They suggest how all three testing techniques can be combined.

In our context random testing, unit testing and system level testing are relevant. While there is no guideline to select the type of testing, random testing is relatively inexpensive but may not be very effective. Ciupa et al. [15] have shown that each testing strategy has its own benefits and it is not always the case that manual testing will subsume random testing or vice versa, whereas Hamlet [16] has shown that in certain situations only random testing is effective. There are many variations derived from purely random testing [17], [18] none of which is mature enough to be used by practitioners in an industrial setting. Owen et al. [19] show how random testing using the tool Lurch can be effective in finding errors. The average time taken by random testing to detect faults was much less than model checking. However, they focus only three mutation operators limiting the type of errors that can be found.

Other types of testing (including integration testing and unit testing) can be used. Although integration testing differs from system testing Borner and Paech [8] use dependency information to select integration tests. However, they do not describe how such integration tests can be generated. Thus they only address the selection problem. The approach by Schätz and Pfaller [20] is to use system tests to exercise certain behaviours at the unit level. Thus the focus in [20] is the design of system tests that tests behaviour of key components and does not involve other components. So they are interested in deducing a system test from knowing what to test at the component level.

In the domain of automatic test generation, techniques based on formal methods such as symbolic execution or concolic execution [17], [21] and model-checking [22] have proved to be promising. However it is not clear if they scale up to large systems. In our work the AutoGen tool is based on model-checking but is used only to generate tests at the unit level.

Requirements based testing [23] of embedded systems is another approach. While this addresses the issue of traceability, the paper does not compare the test generation technique with other approaches. In principle the approach we recommend (based on our experimentation) can be used in their overall process. Other techniques such as usage based testing [24] are out of scope of our study; because for these embedded systems no usage profiles are available and the aim is to achieve MCDC based coverage to meet the required standards.

The other issue is that of test effectiveness. Coverage based metrics are standard in industry, but mutation testing [11] can often provide a better measure. Mutation is a way of deliberately modifying the code by introducing a potential error. The modified function is called a mutant. Mutation operators define the way in which a particular programming construct gets modified. For instance, a mutation operator can be defined which replaces the operator in a randomly selected relational condition with a different one. Depending on which program unit is selected and how it is mutated, different mutants of the same function can be created. Each mutant can have either one fault or several faults.

In comparison with the existing work on automatic test generation, we have conducted experiments mainly for reactive systems using an automotive example. We have evaluated the impact of long test sequences and also the impact of combining unit tests with system tests. We also compare manual test cases, black box testing and randomly generated test cases. To our knowledge there have been no other experiments to compare these effectiveness.

We use mutations to evaluate effectiveness. In our experiments we primarily consider mutants with exactly one fault [11], [25]. We use the tool Proteum [26] to measure the effectiveness of test sets. Proteum supports all the relevant mutation operators for C programs and is able to analyse large programs which usually result in a large number of mutants. We use Proteum in the standard way and have not made any changes to its features. The functionality of the system under test is compiled separately from the rest of the system so that Proteum mutates only that part of the program which corresponds to its functionality. This ensures that only behaviour errors are introduced and test suites are expected to detect only these types of errors.

A. Testing Terminologies

The definitions as used in this paper for the different terms are as follows.

A System is a C program including the function *main*.

A Unit is any function f other than *main* that is part of the system.

Inputs of a unit or system is the union of all parameters to the unit and all global variables that are referred to by the unit before getting assigned to within the unit itself. Intuitively, the input set includes all variables that may get a value from either the environment or a different unit. Note that, by this definition, a unit could have more inputs than the system.

System	LoC	#Units	#Mutants
Sys_1	120	3	1736
Sys_2	1000	10	9540

Table I
CHARACTERISTICS OF INITIAL EXAMPLES.

Test Vector: for a unit or a system is an assignment of values to all the inputs of that unit or system.

Test Case: for a unit or a system is a sequence of test vectors.

Test Set: (also called a test suite) is a set of test cases which could be at the unit level or system level.

System test sets generated by different algorithms are compared using mutation analysis where:

Killing a mutant: A test set is said to *kill* a mutant if the output of the mutant is different from the output of the original program when run for the test set.

The number of mutants killed by a test set is a measure of effectiveness of the test set. Mutation kill ratio represents this effectiveness. More precisely -

Mutation kill ratio: Let TS represent a test set, and let tc represent a test case. A mutant f' of the program f is said to be killed by a test set TS if and only if there exist a test case tc belonging to TS such that the output of f and f' are not equal when run with the input tc.

Test effectiveness: for the mutation kill ratio of a test set TS is the fraction of mutants killed by TS in a given set of mutants.

B. Test Case Generation

For experiments involving unit testing the unit test cases are generated to achieve MCDC coverage. We use the tool AutoGen [5] to automatically generate these test cases and hence are able to achieve MCDC wherever it is theoretically possible.

In the case of random test case generation since our aim is to compare a randomly generated test case against test cases generated using other techniques like AutoGen or black box or manual, we use a deterministic approach for random test case generation by using the same initial seed. This is because our aim here is not to compare different random generation schemes or select the best possible seed.

III. EXPERIMENTATION

The initial set of experiments were conducted on two small code fragments implementing two subsystems derived from of a power delivery system for an automobile. As these are subsystems of a larger system no manual test is available. The characteristics of the two subsystems are specified in Table I.

The two questions we wish to answer in this part of the experimentation are:

System	Number of Iterations	Random Test	MCDC Test
Sys_1	mc_1 (31)	0.310	0.388
	500	0.345	0.591
	1000	0.346	0.671
	1500	0.347	0.746
	2000	0.347	0.764
Sys_2	mc_2 (35)	0.250	0.433
	500	0.258	0.677
	1000	0.279	0.681
	1500	0.286	0.684
	2000	0.314	0.685

Table II
RESULTS OF INITIAL EXPERIMENTATION.

1) Is a test suite that achieves MCDC coverage more effective than using randomly generated values?
2) Does increasing test case lengths by adding test vectors arbitrarily have an impact on effectiveness?

The results of the initial experimentation are given in Table II. Columns labeled *Number of Iterations*, *Random Test* and *MCDC Test* give the length of the test case, the effectiveness of the random test cases and the effectiveness of the test cases generated to achieve MCDC coverage respectively. Entries tagged mc_1 and mc_2 represent the length of a test case that is sufficient to achieve MCDC coverage for Sys_1 and Sys_2 respectively, which are 31 and 35 test vectors. To obtain test cases of longer sequences the MCDC test case is extended by randomly selecting the required number of test vectors from the existing test case and adding them to the end of the sequence. Thus if $< t_1...t_k >$ is a test case the test case extended to a length of 500 is $< t_1...t_k t_{k+1}...t_{500} >$ where $t_{k+1}...t_{500} \in t_1, ..., t_k$. A random test case is extended by randomly adding additional test vectors.

For both Sys_1 and Sys_2 the test suite satisfying the MCDC criterion was found to have better effectiveness than the test suite generated randomly. This is particularly surprising for extended test cases because in the case of MCDC the test cases are extended with a limited combination of input values, whereas in the case of randomly generated test cases the length is increased randomly allowing for more combinations in theory. This could be because the specific combination of input values required to achieve MCDC is important in improving effectiveness.

Another interesting finding is that extending the test case length has a significant impact on the effectiveness. For Sys_1 the test sets generated by AutoGen showed a 15% improvement while the randomly generated values did not really improve. For Sys_2 both the test sets showed only marginal improvement especially after 500 iterations. When 2000 iterations are considered, the test suite satisfying the MCDC criterion kills at least 20% more mutations than the randomly generated values. Thus it appears to be beneficial to use MCDC test generation with extended test cases.

This is similar to the use of covering arrays [27] to generate long test sequences. Although we have not arrived at an ideal number of iterations, we decided to use 2000 test vector for the next experiment. This is because, the manually written test case was approximately 2000 iterations long.

A. Large Experiment

The next experiment we performed compares the effectiveness of test cases generated using three different techniques - manual test case generation by domain experts, random test case generation and automatic test case generation from precise specifications. We also study the impact of unit tests achieving MCDC coverage on the effectiveness of test cases generated by each of the three techniques.

The experiment was conducted on a medium sized application that implements the wiper control functionality of a car. It consists of approximately 10,000 lines of code and 28 key units. The wiper control system includes several components that control the wiper, the washing mechanism, the rain sensor and the various interactions between these components. It is naturally reactive with the wiper speed being determined by various environmental inputs such as ignition switch, washing mechanism switch, vehicle speed, rain sensor data amongst others.

For generating unit test cases, the units of behaviour we consider include controlling the front and rear wiper, setting the wiping mode to intermittent, high-speed or low-speed, automatically calculating the wiping speed based on the speed of the vehicle, the ambient temperature and input from the rain sensor. There are also units of behaviour which handle a limited number of failure conditions (e.g., battery voltage being low).

This program was subjected to unit tests, tests derived from a specification, manual tests developed by the client and a test set that consists of randomly generated values.

We present some details of the specification and the implementation in the Sections III-A1 and III-A2.

1) The Specification: To enable model-based automatic test case generation, we wrote the specification as a small C Program of about 300 lines. The C code captured all the conditions (e.g., decisions made based on the speed of the vehicle, ambient temperature, presence of rain) listed in the requirements document and did not concern itself with any of the implementation related issues. So it ignores all the low level and hardware related aspects of the implementation. The specification has 17 inputs (12 Boolean, two inputs which can take values in the range 0-15, one input which can take values in the range 0-3 and two variables with ranges 0-2000) and a single output variable - the wiper speed.

Test cases were generated from the specification using AutoGen to satisfy the MCDC criterion. That is, the various decisions in the specification itself were covered. These test cases were then translated into black box test cases for the implementation using an appropriate translation.

Test	Mutation Kill Ratio	Branch Coverage
Only Unit Tests	0.476	85.92
Only Hand Tests	0.255	69.72
Only Black-Box	0.203	68.96
Only Random	0.219	71.08

Table III
RESULTS OF SINGLE TEST SUITES.

Test	Mutation Kill Ratio	Branch Coverage
Hand with Unit Tests	0.573	99.24
Black Box with Unit	0.559	99.55
Random with Unit	0.562	99.55

Table IV
RESULTS OF COMPOSITE TEST SUITES.

2) The Implementation: The implementation of the wiper control unit is a C application generated from a hierarchical state transition specification using a code generator. The hierarchical state transition specification is not a specification in the classical sense. It has many implementation level details and is used by the engineers to generate the actual code. The hierarchical state transition system is, therefore, the implementation with the tool associated with it being used as the code generator. The system itself is a parallel composition of several state transition systems. The code consists of a function corresponding to each transition system. Therefore, for the experiment, the function corresponding to each transition system is considered as a unit and the entire code as the system.

As the C code is automatically generated it has a uniform and repetitive structure. The code consists of several switch cases to implement the state transition systems and enum declarations corresponding to the states in each state transition system. We discuss the relevance of this later.

There are in all 21 parameters that can be categorised as configuration parameters and input parameters. The value of the only output, wiper speed, is determined by the combination of the values of these 21 parameters. There is a direct map between these 21 inputs and the 17 inputs identified at the specification level. Proteum generates 29,446 mutants for this program.

Since we are measuring relative effectiveness of the various testing strategies, an absolute adequacy criterion is not of direct interest to us. Thus we only report the percentage of mutants killed. However, we analyse the mutations that are not killed to check if our testing has missed any significant class of mutations. This analysis is more detailed when the kill ratio falls below 50%. We also use `gcov` to measure the branch coverage of our system test suites as an additional measure of quality of test suite.

IV. Results and Analysis

The results are presented in Tables III and IV. The first observation from Table III is that system tests on the implementation by themselves do not kill many mutants. Even the specially crafted manual tests are able to kill only around 25% of the mutants. The automatically generated unit tests when combined (for the 28 units) killed close to 48% of the mutants. This is because the unit tests achieve MCDC and hence exercise more branches than the high level system tests. It is standard software engineering practice to use both unit tests and system tests. So to reflect the practice, system level tests need to be combined with the unit tests to measure the effectiveness. Our experimentation clearly demonstrates the importance of unit tests.

The manual tests were generated by domain experts and the specification based black box tests were generated systematically from a formal specification. We therefore expected the manual designed tests and specification based black box tests to perform much better than the tests using randomly generated values. To understand why the results varied we analysed the implementation of the system and discovered the role of the types of variables used and the conditions they are involved in. As the main implementation had only 2 non-boolean variables the conditions expressed in the program were not very complex. So obtaining MCDC coverage did not require the identification of interesting values for these variables. Manually calculating the values for MCDC coverage is not trivial. Hence the manually designed tests did not perform as well as expected.

Therefore, we hypothesise, that tests that are extended from the sequences generated by AutoGen will work better on systems that have several integer and float value variables that are involved in conditions especially in comparisons against constant values. This is based on a preliminary analysis of another subsystem from the automotive domain. This subsystem had 109 input variables out of which 100 were integer or float type. The findings are presented in Table V. This hypothesis needs to be further validated through more experiments.

While the overall mutation kill ratio appears to be low, a manual examination of about 7000 alive mutants indicated that at least 35% of mutants were not killable.

A. Equivalent mutants analysis

Trying to identify non-killable (including equivalent) mutations is very tedious. However we have examined a large variety of mutations by hand. This is consistent with the findings reported in [28]. They look at only about 40 mutations while we have examined close to 17,000 mutations across various programs in our experiments (i.e., the two subsystems and the actual implementation).

We now describe some of the reasons for the equivalent mutations at the implementation level. There are two main categories, those related to lack of strong type checking in C and those related to the program structure.

As the programs are written in C, there is no direct Boolean type. As Boolean values are simulated by integers, program fragments of the type `x == 1` (where one is checking if `x` is true) results in mutations of the form `x >= 1`, `x == 5`, `x == y` where `y` is an integer variable which takes only positive values. These mutations are not killable by the various test vectors. Similarly, replacing the logical operators with bit wise operators leads to non-killable mutants. A suggestion (for the purposes of mutation based testing evaluation) is to rewrite `x == 1` as `x` to avoid such mutations. About 15% of all mutations are of this nature.

Another example of lack of strong type checking is in the use of enumerated types. This leads to equivalent mutations as shown by the following example declarations.

```
enum {v1, v2, v3} x;
enum {v4, v5, v6} y;
```

Mutants of the form `x = v4`, `x = 0`, `y = v3`, `y = 2` are generated from the original statements `x = v1` and `y = v6`. Clearly the mutants cannot be killed. The implementation has 7 such enumeration types with about 4 values. So for each assignment statement, about 7 mutations are not killable. More than 10% of the overall mutations are of this nature.

We now give an example of the program structure leading to equivalent mutants. Given the reactive nature of the system, most of the functions have a `switch` statement that is used to determine the value of the output based on the current state and the input values. Most of this has the following structure.

```
void function f() {

switch (inp) {
 case x1: S1; break;
 case x2: S2; break;
 case x3: S3; break;
 default: break;
} // end of switch
} // end of function
```

Now mutating any of the `break` statements to a `return` statement results in an equivalent mutant. Deleting the last two `break` statements also results in equivalent mutants. About 5% of the mutations are of this type.

Then there are the usual equivalent mutations such as replacing `0` by `-0`. which occurs quite frequently as we are dealing with many Boolean values. Similarly, erasing the final `return` statements in functions that return `void` has no effect. These account for at least 5% of the mutations. So the net result is that about 35% of all mutations are not killable.

System	Number of Inputs	Number of inputs with integer/float type	Number of expressions involving integer/float constants	MCDC effectiveness	Random effectiveness
Original System	21	2	13	0.212	0.219
New Sub system	109	100	109	0.685	0.43

Table V
VARIABLE TYPES AND OCCURRENCE IN COMPARISONS

B. Threats to Validity

Although we have reported the results using realistic systems there are a number of limitations that can impact the validity. We now discuss these in detail.

The first threat is related to the choice of our system for testing. It is not clear if all embedded systems in the automotive domain have the characteristics of the system we have studied. While it has been suggested to us by our clients that this is a typical example, there has been no rigorous study to justify this assumption. This entire experiment took about 7 man months of effort and it is very unlikely that we can repeat the same study on another example in an industrial setting. It would also be very hard to replicate the formal black box testing as creating a formal specification is non-trivial. It is more likely that the results reported here will be validated only by conducting similar experiments on other examples in the field.

The second threat to validity is related to the metric we have used for the comparison of the testing algorithms. The limitations of using mutation based testing [29] is well recognised. This is because we have not validated if the mutations generated correspond to actual defects in the system. It would have been good to get access to an older version of the programs used in this experiment and see how many real defects were identified by our tests. But this was not possible given the requirements of our client. Also, we have generated and run only one instance of the randomly generated tests.

Furthermore, although we have looked at a large number of mutations by hand, there are many mutations that have not been examined. So our estimate of 35% non-killable mutations for the implementation while conservative is still only an estimate.

The third threat to validity is related to the information we received for manual testing. We are comparing the effectiveness of the automatically generated tests with manual tests. This could be dangerous as our approach could appear to be very effective if the manual tests are of poor quality. While the automatically generated tests could provide better results, the results are not generalisable. For the system we studied, the manual tests were deemed to be effective by the field engineers and hence cannot be viewed as having poor quality. In this context our experiments, can be viewed as a validation of such claims.

Other threats to validity include the bound of 2000 iterations (it is not clear if that would suffice for all implementations – although there is a pragmatic reason for its choice) and the use of MCDC criterion for the tests generated from the specification (it may be the case that other test generation techniques such as all round trip paths may perform better than our chosen technique).

V. CONCLUSIONS AND FUTURE WORK

Based on our findings from the experiments conducted on representative pieces of code from the automotive domain we conclude that for simple embedded control systems -

- Effective testing requires sequences that are longer than those that just satisfy a coverage requirement.
- Automatically generated system tests have a fairly good mutation adequacy ratio (after accounting for non-killable mutants) when combined with unit tests.
- When most inputs have either boolean type or take values from a small domain random testing works as well as other techniques.
- Our experiment also shows that the effectiveness of automatically generated test cases is comparable to manually produced test cases.

In the large example we considered a test set that consists of randomly generated values appears to be comparable to manual testing. Whether this holds for all other software systems needs to be further validated. Similarly the hypothesis that random testing is in the presence of mainly boolean inputs and perhaps fares badly when inputs take values from larger domains needs further validation.

Specifically, we have also used the upper bound of 2000 iterations to generate large test sets. Further experimentation is needed to determine a lower upper bound on the number of iterations.

ACKNOWLEDGEMENT

Padmanabhan Krishnan was supported by a grant from Tata Consultancy Services.

REFERENCES

[1] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," in *Proceedings of the IEEE*. IEEE, Feb 2007, pp. 356–373.

[2] X. Wu, J. J. Li, D. Weiss, and Y. Lee, "Coverage-Based Testing on Embedded Systems," in *International Workshop on Automation of Software Test (AST)*. IEEE, 2007, pp. 64–70.

[3] M. P. E. Heimdahl and D. George, "Test-suite reduction for model based tests: Effects on test quality and implications for testing," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 176–185.

[4] "DO-178B: software considerations in airborne systems and equipment certification," RTCA, 1992.

[5] P. Bokil, P. Darke, U. Shrotri, and R. Venkatesh, "Automatic Test Data Generation for C Programs," in *SSIRI '09: Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 359–368.

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.

[7] J. Hu, Z. Ding, and G. Pu, "Path-based approach to integration testing," *Secure System Integration and Reliability Improvement*, vol. 0, pp. 445–446, 2009.

[8] L. Borner and B. Paech, "Using dependency information to select the test focus in the integration testing process," in *Testing: Academic & Industrial Practice And Research Techniques*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 135–143.

[9] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[10] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE '05: Proceedings of the 27th international conference on Software Engineering*, 2005, pp. 402–411.

[11] J. A. Offutt and R. H. Untch, "Mutation 2000: uniting the orthogonal," in *Mutation testing for the new century*. Norwell, MA, USA: Kluwer Academic Publishers, 2001, pp. 34–44.

[12] M. Rappl, A. Pretschner, C. Salzmann, and T. Stauner, Eds., *3rd Intl. ICSE workshop on software engineering for automotive systems*. ACM, 2006.

[13] E. Bringmann and A. Kramer, "Model-based testing of automotive systems," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2008, pp. 285–493.

[14] A. Arcuri, M. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *Testing Software and Systems*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6435, pp. 95–110.

[15] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner, "Finding faults: Manual testing vs. random+ testing vs. user reports," in *International Symposium on Software Reliability Engineering*, 2008, pp. 157–166.

[16] R. Hamlet, "When only random testing will do," in *RT '06: Proceedings of the 1st International Workshop on Random Testing*. New York, NY, USA: ACM, 2006, pp. 1–9.

[17] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[18] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ISSTA*. ACM, 2011, pp. 265–275.

[19] D. Owen, D. Desovski, and B. Cukic, "Random testing of formal software models and induced coverage," in *RT '06: Proceedings of the 1st international workshop on Random testing*. New York, NY, USA: ACM, 2006, pp. 20–27.

[20] B. Schätz and C. Pfaller, "Integrating Component Tests to System Tests," in *International Workshop on Formal Aspects of Component Software (FACS)*, ser. ENTCS. Elsevier, 2008, vol. 260,1, pp. 225–241.

[21] P. Godefroid, "Compositional dynamic test generation," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2007, pp. 47–54.

[22] D. Beyer, A. J. Chlipala, and R. Majumdar, "Generating tests from counterexamples," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 326–335.

[23] T. Bauer, F. Bohr, D. Landmann, T. Beletski, R. Eschbach, and J. Poore, "From requirements to statistical testing of embedded systems," in *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, ser. SEAS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 3–.

[24] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *Software: Practice and Experience*, vol. 25, no. 1, pp. 97–108, 1995.

[25] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[26] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. Simao, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero, "Proteum: a family of tools to support specification and program testing based on mutation," *Mutation testing for the new century*, pp. 113–116, 2001.

[27] X. Yuan, M. B. Cohen, and A. M. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing," in *Proceedings of Automated Software Engineering (ASE)*. ACM, 2007, pp. 405–408.

[28] B. J. M. Gruen, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *International Workshop on Mutation Analysis*. IEEE, 2009, pp. 192–199.

[29] A. S. Namin, J. H. Andrews, and J. D. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 351–360.