The basic problem in software testing is choosing a subset from the near infinite number of possible test cases. Testers must select test cases to design, create, and then execute. Due to limited test resources, it is important to select the best possible set of tests. In this article, the authors present test case design with the Classification Tree Editor (CTE XL), a popular tool for systematic black-box test case design of classification tree-based tests. They show how to integrate weighting factors into classification trees and automatically generate prioritized test suites. In addition to "classical" approaches such as minimal combination and pair-wise, weighted counterparts have been added, along with statistical testing and new generation rules. CTE XL supports prioritization by occurrence probability, error probability, or risk.

Key words

classification tree method, pairwise testing, prioritized testing

Automated Test Case Generation Using Classification Trees

PETER M. KRUSE AND MAGDALENA LUNIAK
Berner & Mattner Systemtechnik GmbH

INTRODUCTION

Software testing is unfortunately one of the activities in which too few resources are invested. Considering a common industry-driven software development, there is often not enough time for testing, since the software under test may not be finished in time and the release date cannot be delayed. Ideally project teams should readjust the planning for their software projects to allow more time to be spent on testing; however, the product launch would be delayed too.

In the real world, software testers have to deal with multiple challenges: Their resources are limited, deadlines arrive quickly, and the system under test is sometimes finished too late. Using the software testing period as a buffer for any kind of delay in other phases of software and product development, as well as shortening testing efforts to fulfill the planed product launch date, is not uncommon.

Testers have to handle this situation. If there is not enough time to carry out all test cases, a tester can try to finish as many tests as possible. If promising test cases have been determined from previous software projects, a tester could start with these test cases, perhaps accompanied by the first n test cases from a global test list. While there is no guarantee the selected test cases are more important than any of the other test cases, there is no way to identify the most important test cases from a given test suite.

In this article the authors present an approach for test suite optimization using the classification tree method. They introduce

the weighting of classification tree elements, allowing them to generate prioritized test suites. These test suites can be optimized by selecting only subsets of test cases. They introduce coverage criteria for identifying the importance of any single test case under given test aspects.

FOUNDATIONS

Classification Tree Method

The classification tree method (Grochtmann and Grimm 1993) aims for systematic and traceable test case identification for functional testing over all test steps (for example, component test, system test). It is based on the category partition method (Ostrand and Balcer 1988), which divides the test domain into disjunctive classes representing aspects of the importance of the test object. Applying the classification tree method involves two steps—designing the classification tree and defining test cases.

Design of the classification tree

The classification tree is based on the functional specification of the test object. Figure 1 shows an example tree for the adaptive cruise control test object. For each aspect of interest (classifications), the input domain is divided into disjoint subsets (classes). In the authors' example, classifications are preceding vehicle, speed, and daylight. Classes for speed are low, medium, and high. The class car is further refined into different shapes, which are limousine and cabriolet. Refinements allow test objects to be modeled with any preferred granularity.

Definition of test cases

Having composed the classification tree, test cases can be defined by combining classes from different classifications. Figure 2 shows four test cases for the system under test. In Testcase 1, the adaptive cruise control is tested with a motorcycle as the preceding vehicle, the driving speed is low, and the test is run during the day. Since classifications only contain disjoint values—obviously speed cannot be low and high at the same time—test cases cannot contain several values of one classification.

Classification Tree Editor

The Classification Tree Editor (CTE XL) was introduced together with the classification tree method (Grochtmann, Grimm, and Wegener 1993). A screenshot of CTE XL is provided in Figure 3. Current versions of the CTE XL support automated test case generation, user-defined dependency rules, and the integration of requirement and test management tools (for example, IBM Rational DOORS and HP Quality Center) (Lehmann and Wegener 2000).

Current test case generation offers four different modes:

- *Minimal combination* creates a test suite that uses every class from each classification at least once in a test case.
- *Pairwise combination* creates a test suite that uses every class pair from disjunctive classifications at least once in a test case.
- *Threewise combination* ("triple-wise") creates a test suite that uses every triple of classes from disjunctive classifications at least once in a test case.
- *Complete combination* creates a test suite that uses every possible combination of classes from disjunctive classification in a test case.

FIGURE 1 Classification tree for the adaptive cruise control test object

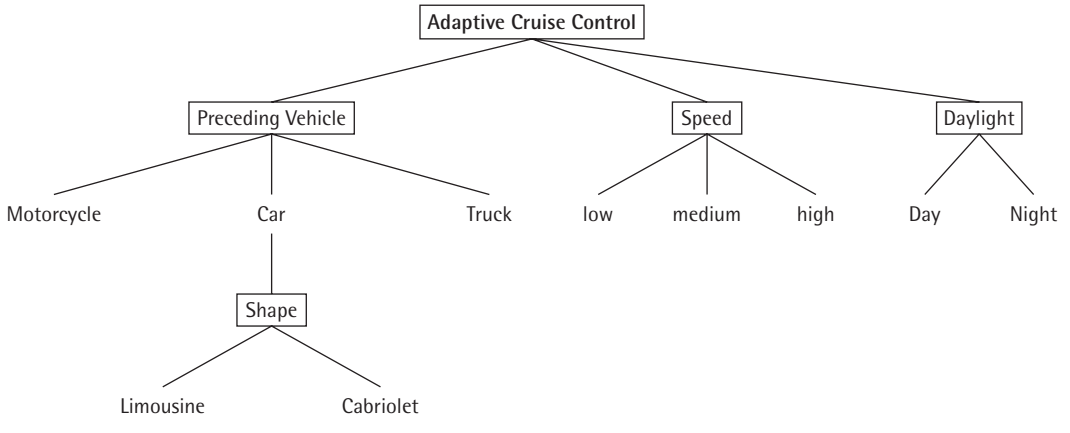


FIGURE 2 Classification tree with test cases

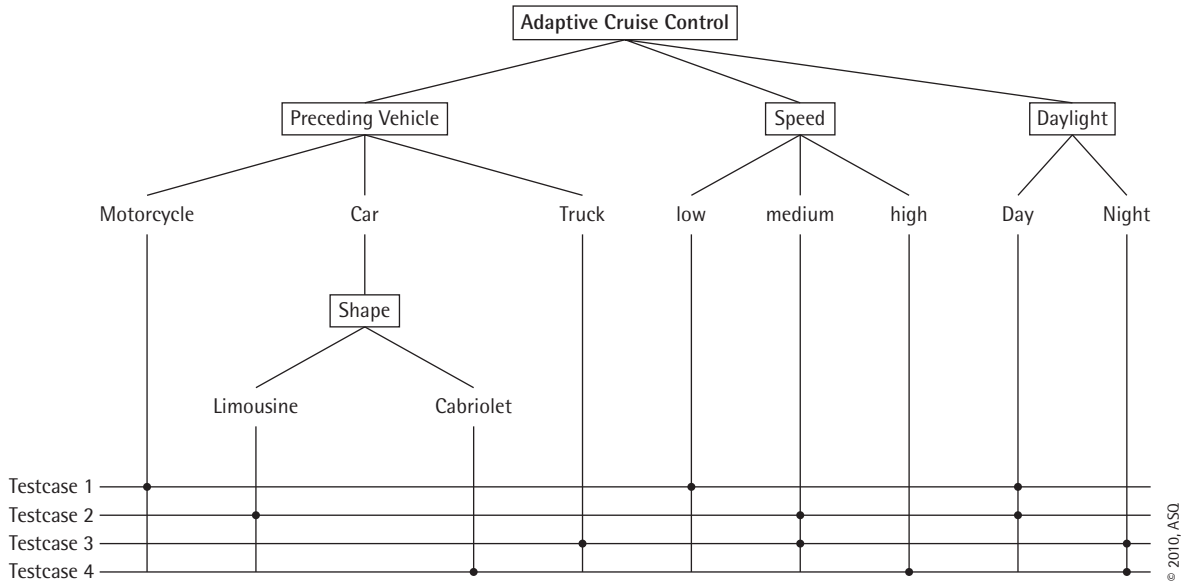
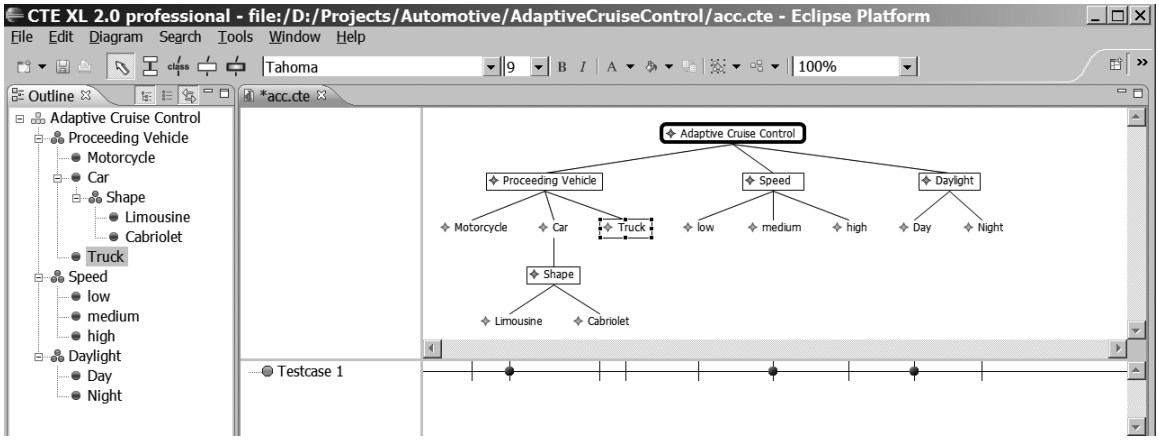


FIGURE 3 Screenshot of CTE XL



Limitations

Earlier versions of the CTE XL only support size reduction of test suites by applying different combination rules, for example, using the pairwise combination if the complete combination results in too many test cases. Combination rules do not support prioritization of certain test aspects (for example, costs of an error). The classification tree method does not yet support the weighting of tree elements according to these test aspects. Since they are not taken into account, these aspects cannot be used for comparing the test cases. Additionally, test cases are generated in an arbitrary order; thus, there is no means of selecting an optimized test suite.

IMPROVEMENTS

The authors have identified the following enhancements:

1) Integration of weights into the classification tree to allow the prioritization of certain test aspects
2) Definition of prioritized combination rules using tree weights
3) Introduction of coverage criteria
4) Test suite optimization

Prioritization

Prioritization is used to assign values of importance to classification tree elements. These values of importance are called *weights*. To cover all kinds of test aspects, these weights can differ. Higher and lower weights should reflect higher and lower importance, respectively. Consequently, one is able to compare the elements of the classification tree to determine their importance under a given test aspect and to prioritize test aspects during test case generation.

The authors analyzed several existing prioritization techniques. Elbaum et al. provide good overviews of existing approaches (Elbaum, Malishevsky, and Rothermel 2002; Elbaum et al. 2004). The following three models were selected to provide a basis for prioritization:

- Prioritization based on a usage model (Walton, Poore, and Trammel 1995): This prioritization tries to reflect usage distribution of all classes in terms of usage scenarios. Classes with a high occurrence have higher weights than classes with a low occurrence.
- Prioritization based on an error model (Elbaum, Malishevsky, and Rothermel 2002): This prioritization aims to reflect distribution of error probabilities of all classes. Classes with a high probability of revealing an error have higher weights than classes with a low probability.
- Prioritization based on a risk model (Amland 2000): This prioritization is similar to prioritization based on an error model but also takes error costs into account. Risk is defined as the product of error probability and error costs. Classes with a high risk have higher weights than classes with a low risk.

Example

The system under test, the adaptive cruise control, is shown with assigned occurrence values in Figure 4. All classes were assigned values of importance. As the figure shows, medium is the most probable speed. Low and high are the subsequent values in descending order of importance.

The weights of all classes composed within one classification sum to 1 in the occurrence model. They are independent of each other, however, in both the error model and the risk model. The class car has an occurrence rate of 0.7 of all preceding vehicles. If the preceding vehicle is a car, limousines have an occurrence rate of 0.9 and cabriolets have an occurrence rate of 0.1. Refinements are interpreted as conditional probability in the occurrence model. The resulting occurrence probability for a limousine being the preceding vehicle is 0.63 (= 0.7 * 0.9), and for a cabriolet it is 0.07, accordingly.

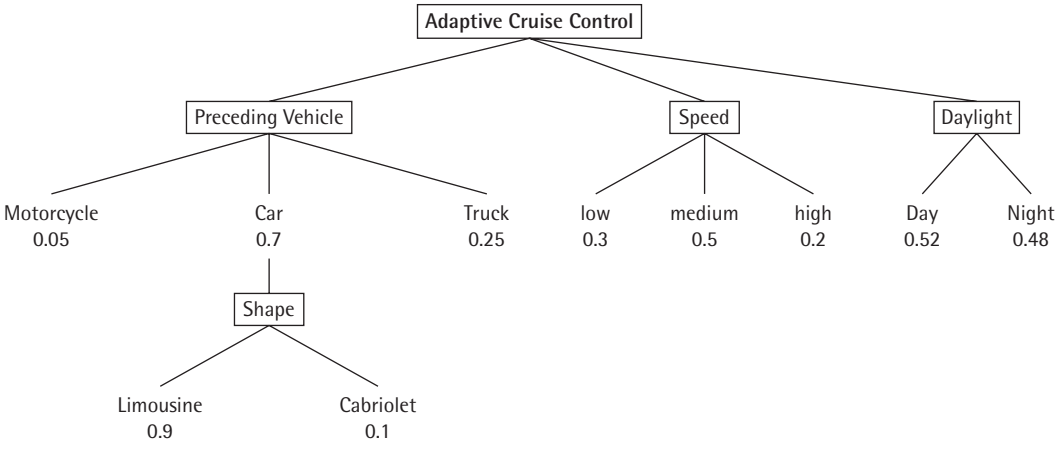
Since error and risk values are independent of each other, an interpretation as conditional error or risk fails in both the error model and the risk model. All values in these two models are absolute values.

New Combination Rules and Automated Test Case Generation

New combination rules are applied to the weighted classification tree. Existing combination rules are extended and completely new rules are designed. Having implemented combination rules, test suites can be generated automatically.

The existing minimal combination and pairwise combination rules have been extended into prioritized

FIGURE 4 Test object with assigned occurrence values



minimal combination and prioritized pairwise combination rules. The authors have added class-based statistical combination to enable statistical testing with the classification tree method. All combination rules are applicable to occurrence, error, and risk models.

Prioritized Minimal Combination

The prioritized minimal combination (PMC) is based on the minimal combination rule. The minimal combination creates a test suite that covers each class in at least one test case, but does not take into account the order of class coverage.

Therefore, the authors have extended this combination to PMC. Test cases are generated in descending order of importance. Each new test case contains the most important class remaining from all classes in the tree plus as many still uncovered classes as possible. If there are only covered classes left and the test case is not yet completed, the most underrepresented classes are taken. Underrepresentation is calculated by comparing the weight of each class with the ratio of already-generated test cases containing this class. This ensures that classes with a high weight are used more often during test case composition than classes with a low weight.

PMC guarantees coverage of at least the n most important classes by the n first test cases. The authors accept that test suites may be slightly larger than generated by plain minimal combinations. As a matter of course, dependency rules are taken into account. For identical trees, the PMC is deterministic and always generates the same test suite.

Prioritized Pairwise Combination

The plain pairwise combination creates a test suite that covers each class pair in at least one test case. The authors extended this combination to a prioritized pairwise combination (PPC), which uses class pairs. The combined weight for class pairs is calculated by multiplying either the occurrence or error probabilities. By contrast, the combined risk is the product of the summed individual costs and the multiplied individual errors. Test cases are generated in descending order of importance.

The most important class pair from all class pairs still uncovered is identified for composing each new test case. Furthermore, the authors determined all candidate test cases containing this class pair and calculated index values for all candidates. This index value includes the weights and the number of newly covered class pairs. It emphasizes class pairs with a high weight. In this way, PPC selects the test case with the highest assigned index value.

PPC guarantees coverage of at least the n most important class pairs by the n first test cases. The generated test suite may be slightly larger than the result of the plain pairwise combination because of weights and dependencies taken into account. The generation process using PPC is deterministic: the same test suite is generated for identical trees.

Example

The combined occurrence values for class pairs are shown in Table 1. The pair with the highest value is limousine and day, followed by limousine and night. The authors would therefore expect to see these pairs in the first two test cases of a generated test suite. Since classes from the same classification cannot be combined, there are cells without values in the table.

The expected result of a test suite generated with PPC with respect to the occurrence model is given in Table 2. The first test case covers the pair limousine

TABLE 1 Combined class pair values for occurrence model

		Day	Night	low	medium	high
		0.52	0.48	0.3	0.5	0.2
Motorcycle	0.05	0.026	0.024	0.015	0.025	0.01
Limousine	0.63	0.3276	0.3024	0.189	0.315	0.126
Cabriolet	0.07	0.0364	0.0336	0.021	0.035	0.014
Truck	0.25	0.13	0.12	0.075	0.125	0.05
low	0.3	0.156	0.144	/	/	/
medium	0.5	0.26	0.24	/	/	/
high	0.2	0.104	0.096	/	/	/

TABLE 2 Test cases for PPC occurrence including weights calculation

#	Preceding Vehicle	Speed	Daylight	Vehicle-Speed	Vehicle-Daylight	Speed-Daylight	Sum
1	Limousine	medium	Day	0.315	0.3276	0.26	0.9026
2	Limousine	medium	Night	0.315	0.3024	0.24	0.5424
3	Limousine	low	Day	0.189	0.3276	0.156	0.345
4	Truck	low	Night	0.075	0.12	0.144	0.339
5	Truck	high	Day	0.05	0.13	0.104	0.284
6	Limousine	high	Night	0.126	0.3024	0.096	0.222
7	Truck	medium	Night	0.125	0.12	0.24	0.125
8	Cabriolet	medium	Day	0.035	0.0364	0.26	0.0714
9	Cabriolet	low	Night	0.021	0.0336	0.144	0.0546
10	Motorcycle	medium	Day	0.025	0.026	0.26	0.051
11	Motorcycle	low	Night	0.015	0.024	0.144	0.039
12	Cabriolet	high	Night	0.014	0.0336	0.096	0.014
13	Motorcycle	high	Night	0.01	0.024	0.096	0.01

and day as expected, followed by a test case containing limousine and night.

The table gives all class weights along with the respective test case. Bold numbers indicate the remaining most important class pair. The pair limousine and day has the highest weight, so it is selected first. Then the algorithm tries to complete the test case by selecting the highest remaining value from speed. It selects medium, as limousine and medium has the highest class pair weight and it is not yet covered by any test case. As there are no more uncovered classifications, the test case is complete.

In the second run, the algorithm selects the remaining most important class pair, limousine and night. Then, the test case is completed again by selecting the pair with the highest weight that is still not covered. The algorithm selects medium and night, as it fulfills both criteria. The test case is complete now, since there are no more uncovered classifications. Italic numbers are repeated selections. They are not taken into account for sum calculation. In test case no. 2, this applies for limousine and day, since this class pair was part of an earlier test case.

The algorithm finishes when all class pairs have been covered at least once. In the authors' example, the algorithm created 13 test cases.

Class-Based Statistical Combination

In contrast to the limited resources problem introduced at the beginning of this article, there can be situations in software testing where an extended set of tests is desired, for example, for testing nondeterministic systems. As weights have already been annotated to the classification tree, they can be reused for statistical testing. Here, the repeated execution of test cases increases the chance of revealing errors.

The class-based statistical combination (CSC) rule is a completely new combination rule for statistical testing in the classification tree method. CSC generates test suites reflecting the distribution of classes defined by a priority model. The tester obtains a test suite with a given size that reflects the test aspect well. Test cases are generated using a random process. For each classification in the tree, one class is randomly selected according to its distribution within the classification. Test suites created using CSC may contain redundant test cases. With CSC, the classification tree method allows for testing of nondeterministic systems.

COVERAGE CRITERIA AND OPTIMIZATION OF TEST SUITES

To assess the test progress, coverage criteria are commonly used in the testing process. Therefore, the authors introduce coverage criteria for prioritized combination rules. The each-used coverage (EUC) for PMC measures the coverage of classes in a test suite. It only takes classes into account that can be covered by test cases and are not excluded by dependencies.

$$EUC_{PMC} = \frac{number\ of\ covered\ classes}{number\ of\ coverable\ classes}$$

The weight coverage (WC) measures the coverage of weights in a test suite. Again, the authors only take into account classes that are coverable.

$$WC_{PMC} = \frac{sum\ of\ weights\ of\ covered\ classes}{sum\ of\ weights\ of\ coverable\ classes}$$

For the PPC, the authors defined the pair coverage metric, measuring the coverage of new pairs in the test suite.

$$EUC_{PPC} = \frac{number\ of\ covered\ class\ pairs}{number\ of\ coverable\ class\ pairs}$$

The weight coverage can be calculated in the same way.

$$WC_{PPC} = \frac{sum\ of\ weights\ of\ covered\ class\ pairs}{sum\ of\ weights\ of\ coverable\ class\ pairs}$$

Both metrics are relative since they ignore class pairs that are not coverable due to user-defined dependencies.

The authors do not define coverage criteria for the class-based statistical combination since test generation is based on a random process, and they cannot assure any coverage. Nevertheless, they offer a chi-square test to measure the quality of the generated test suite. It compares the expected distribution of classes or class pairs with the resulted actual distribution. The expected distribution of classes results from the respective weights. For example, the expected distribution of classes in the classification daylight would be 52 percent for day and 48 percent for night.

Example

For prioritized pairwise combination, Table 3 and Figure 5 show that the WC grows faster than the EUC for the first six test cases. In the last five test cases, the

TABLE 3 Test cases with coverage values

#	Preceding Vehicle	Speed	Daylight	EUC	WC
1	Limousine	medium	Day	0.12	0.3
2	Limousine	medium	Night	0.19	0.48
3	Limousine	low	Day	0.27	0.6
4	Truck	low	Night	0.38	0.71
5	Truck	high	Day	0.5	0.8
6	Limousine	high	Night	0.58	0.88
7	Truck	medium	Night	0.62	0.92
8	Cabriolet	medium	Day	0.69	0.94
9	Cabriolet	low	Night	0.77	0.96
10	Motorcycle	medium	Day	0.85	0.98
11	Motorcycle	low	Night	0.92	0.99
12	Cabriolet	high	Night	0.96	0.99
13	Motorcycle	high	Night	1	1

WC increases slower since the class pairs with a high priority have already been covered at the beginning of the generation process (as shown in Table 2). The whole test suite guarantees both 100 percent of the EUC and the WC.

The test suite optimization allows one to achieve a defined coverage, while minimizing the size of the test suite. Only using the first test case, the authors already obtain 30 percent of the WC. The first three test cases reach 60 percent of the WC. Figure 5 illustrates the relation between potential subsets of test suites and the resulting coverage levels.

IMPLEMENTATION

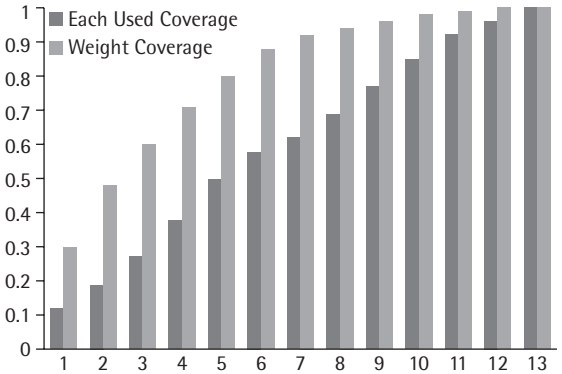
After identifying improvements to the classification tree method, the authors integrated them into the new version of the CTE XL.

Prioritization

To include weights for classes, the tester has to assign values to the classification tree. Models can be used independently of each other; it is possible to only use occurrence probabilities by assigning only occurrence values or to use all three models consecutively.

To enable the use of a priority model, the tester assigns values to all classes, as shown in Figure 6. The class truck has an occurrence probability of 25 percent. In the screenshot, error and risk values have been entered as well: the error probability is 5 percent and the costs of an error are 40,000, resulting into a risk of 2000.

FIGURE 5 Each used coverage vs. weight coverage



New Combination Rules and Automated Test Case Generation

With all values assigned, the user can start prioritized test case generation. The test case generation dialog provides all available combination rules (see Figure 7). Priority models are available only if respective values have been assigned to all classes.

Prioritized combinations do not need any further parameters. For statistical combinations (for example, CSC), the tester can specify the number of test cases to generate and the number of tries. The complete result for PPC occurrence can be found in Figure 8. The order of test cases conforms to Table 2.

Coverage Criteria and Optimization of Test Suites

Using the optimization dialog (see Figure 9), the tester can optimize the generated test suite by selecting the preferred coverage criteria and adjusting the coverage level. A preview of the resulting test suite is shown, together with the number and ratio of test cases.

CONCLUSIONS AND FUTURE WORK

The authors developed and implemented the integration of weights into the classification tree to allow prioritization with respect to test aspects. Prioritized combination rules using tree weights were defined and implemented. Thanks to the introduction of coverage criteria, test suite optimization was enabled.

FIGURE 6 Prioritization (Screen Shot)

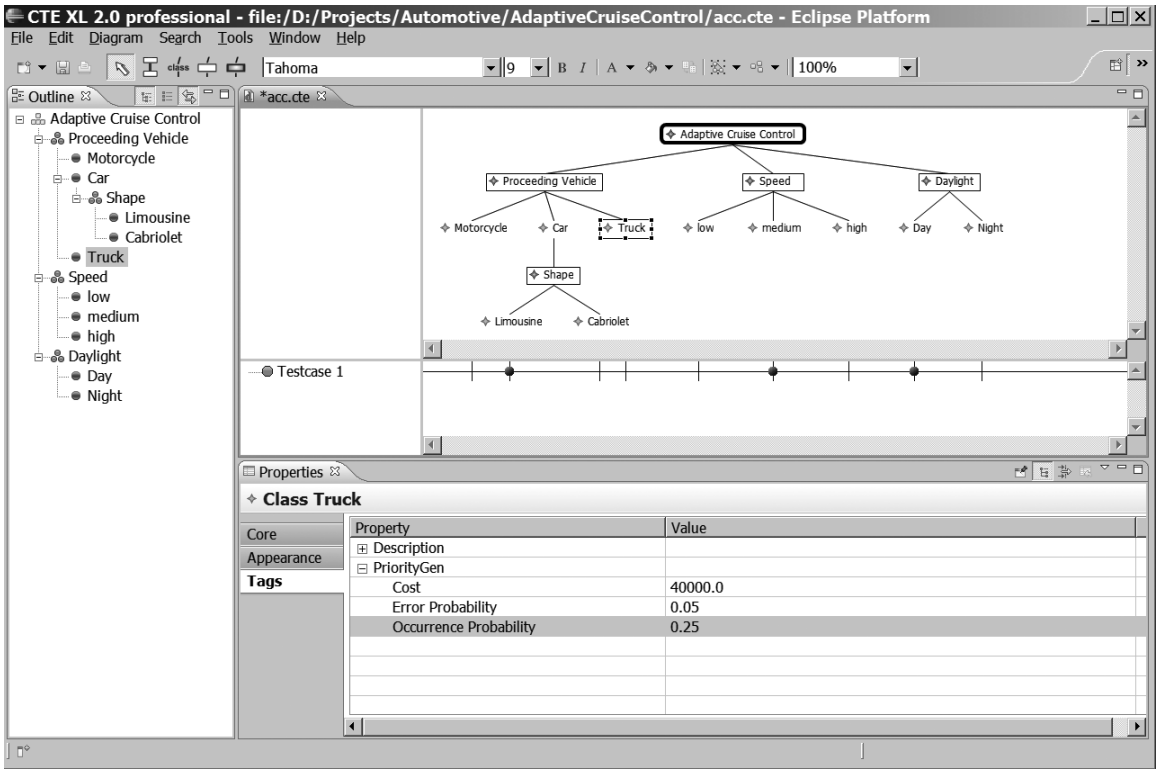
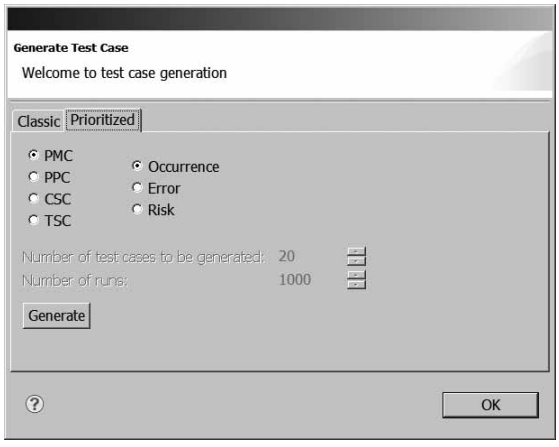


FIGURE 7 Combination rules (Screen Shot)



As a result, testers are now able to prioritize and weight tree elements according to test aspects (for example, costs of an error). Test aspects are used for comparing the test cases; test cases are generated in a defined order. Testers can now select optimal test suites meeting their individual requirements.

The use of weights for statistical testing for nondeterministic systems is available, too. All of these enhancements are valuable extensions to the classification tree method. They support testers to focus on their most important test aspects. Subsets of test suites can be used as an incoming inspection in initial tests.

Future work will concentrate on further integration with testing tools. Moreover, the authors will extend the remaining plain combinations (threewise and complete combination) in such a way that they take into account priority values and develop further combinations for statistical testing.

References

Amland, S. 2000. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software* 53, no. 3:287–295.

Elbaum, S., A. G. Malishevsky, and G. Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 28, no. 2:159–182.

Elbaum, S., G. Rothermel, S. Kanduri, and A. G. Malishevsk. 2004. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* 12:185–210.

Grochtmann, M., and K. Grimm. 1993. Classification trees for partition testing. *Software Testing, Verification & Reliability* 3, no. 2:63-82.

Grochtmann, M., K. Grimm, and J. Wegener. 1993. Tool-supported test case design for black-box, testing by means of the classification-tree editor. In *Proceedings of EuroSTAR'93*, London.

FIGURE 8 Test suite generated for PPC using occurrence values

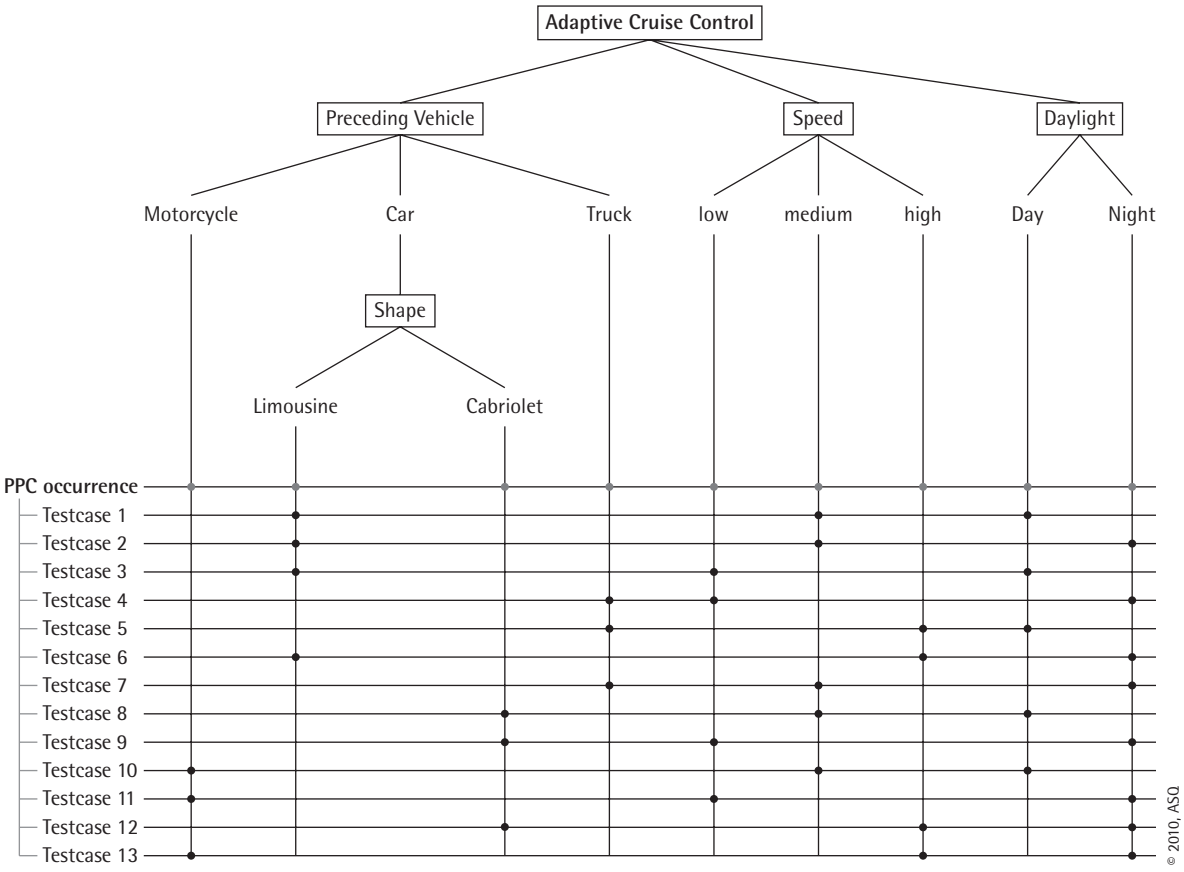
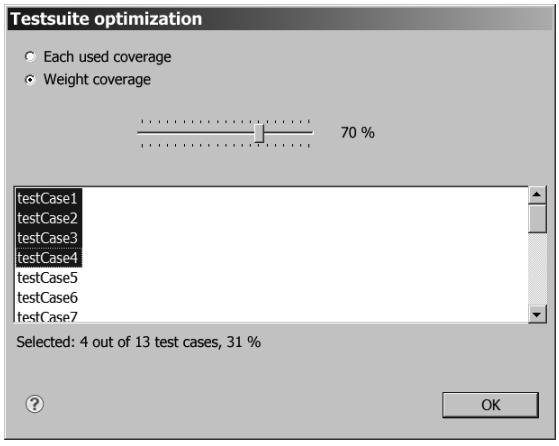


FIGURE 9 Testsuite optimization in CTE XL



Lehmann, E., and J. Wegener. 2000. Test case design by means of the CTE XL. In Proceedings of the 8th European International Conference on Software Testing, Analysis & Review, EuroSTAR 2000, Kopenhagen, Denmark.

Ostrand, T. J., and M. J. Balcer. 1988. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31, no. 6:676–686.

Walton, G. W., J. H. Poore, and C. J. Trammell. 1995. Statistical testing of software based on a usage model. *Software Pract. Exper.* 25, no. 1:97–108.

Biographies

Peter M. Kruse is a software engineer working in the domain of testing, including evolutionary testing and the classification tree method. He is an experienced software developer and tester in the German automotive industry. Kruse's project experience includes Hardware-in-the-Loop Testing (HiL), model driven-development (MDD), and evolutionary structural and functional testing. He is responsible for development of CTE XL, a very popular test design tool. Kruse can be reached at peter.kruse@berner-mattner.com.

Magdalena Luniak's project experience includes development of statistical analysis tools. She has transferred her experience from data analysis into the field of software testing, working on the enhancement of the classification tree method at Berner & Mattner. Luniak has been involved in software development at StataCorp LP in Texas, as well as teaching computer science at the Technical University Berlin, Germany.