Quality of Automatically Generated Test Cases based on OCL Expressions

Stephan Weißleder
Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
weissled@informatik.hu-berlin.de

Bernd-Holger Schlingloff
Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
hs@informatik.hu-berlin.de

Abstract

In this paper, we deal with coverage criteria for boundary testing. We focus on the automatic generation of boundary tests based on OCL expressions and evaluate the quality of these tests with established coverage criteria like MC/DC. We define and apply new coverage criteria, evaluate their efficiency via mutation testing, and substantiate all explanations by an example, part of a model of an elevator control.

1 Introduction

In specification-based testing, each test run validates the system under test (SUT) corresponding to reference specifications, e.g. models [19]. Additionally, these specifications can be used to provide information about the generation of test input sequences, expected behavior of the SUT, or both. Commercial tools are available which automatically generate test sets based on models [10, 18, 20].

In order to quantify the quality of test sets, coverage criteria are necessary [5, 11, 14]. There are many widely approved coverage criteria, each of which is focused on certain aspects of the specification or the SUT. Whereas most coverage criteria have been defined and investigated in the area of code-based test generation, we focus on coverage of the specification. We investigate coverage criteria that either focus on conditions [5] or partition boundaries [11, 12] of the specification. To combine the advantages of both kind of criteria, we also define new coverage criteria. We generate test sets which satisfy such coverage criteria automatically [21] and introduce a prototype [16] that realizes our approach based on the Object Constraint Language (OCL).

This paper is structured as follows. We introduce the example in Section 2. Following, we sketch the intermediate test case tree. In Section 4, we review existing coverage criteria and define new coverage criteria. Then, we evaluate the criteria's efficiency via mutation testing. We summarize our results in Section 6.

Related Work. A detailed survey about model-based testing can be found in [19]. The Unified Modeling Language (UML) is a well-known means for specifying test models [2, 17, 19]. For example, Offutt and Abdurazik derive test cases from UML state machines [13]. Our approach deviates in that we focus on deriving test input value partitions. Other formalisms than the UML (e.g. extended finite state machines [1, 4]) also support test generation but do not adequately support object-oriented systems.

The Object Constraint Language (OCL) is a language to express constraints on models [15, 23]. Hamie et al. consider OCL in combination with state machines and classes [8]. We analyze and transform OCL expressions of a test case tree [21] to generate test input value partitions and corresponding test sets.

Many approaches validate given input partitions and their boundaries [6, 9]. However, they provide no means to also derive them from models. As an exception, Legeard, Peureux, and Utting develop a method for automated boundary testing from the textual languages Z and B [12]. In contrast to this, our work about the automatic generation of test input partitions is based on UML and OCL [21].

Gutjahr [7], Weyuker, and Jeng [22] compare the quality of random tests and partition tests. In [3], Briand et al. consider data flow for testing criteria. Kosmatov et al. [11] define coverage criteria for boundary testing. Additionally, we also consider the relationship of such criteria to coverage criteria that handle conditions (e.g. MC/DC [5]). We do not restrict ourselves to finite partitions. Furthermore, we focus on how to achieve partitions and corresponding coverage criteria for boundary testing from UML models.

2 Example

As an example, we consider part of the functionality of an freight elevator control (figure 1). The elevator can check the weight inside. Accordingly, it gains extra speed to reach its destination earlier if the elevator is empty and it refuses to move if it is overloaded. The full version of the model can be found at [16].

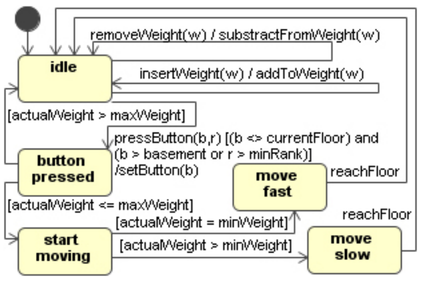
removeWeight(w) / substractFromWeight(w)
idle
insertWeight(w) / addToWeight(w)
[actualWeight > maxWeight]
pressButton(b,r) [(b <> currentFloor) and (b > basement or r > minRank)] /setButton(b)
button pressed
move fast
reachFloor
[actualWeight <= maxWeight]
[actualWeight = minWeight]
reachFloor
start moving
[actualWeight > minWeight]
move slow

Figure 1. Model of an elevator control.

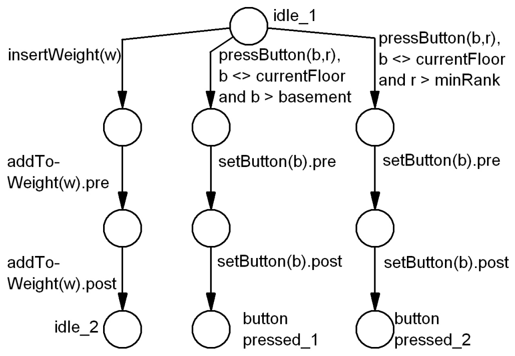

Figure 3. Part of the test case tree.

3 Test Case Tree

In this section, we quickly recall the test case tree structure from [21]. The tree is automatically generated from UML state machines and UML class diagrams. It contains all expressions from both models (e.g. operation pre-/postconditions or transition guards). Each path in the tree corresponds to a logical test case for a path in the UML state machine. The expressions of each path in the tree are transformed into conditions on input values. These conditions correspond to input value partitions for logical test cases. Values near the boundaries of these partitions are selected for the concrete test cases. Further on, we use the expressions of a path as an oracle for the generated test cases.

The branching of the tree results from the transition branches of the UML state machine and the conjunctions of the disjunctive normal form (DNF) of the included expressions: Consider the fragment of the state machine in figure 2 and the part of the test case tree in figure 3. The test case tree's left path outgoing from *idle_1* in figure 3 corresponds to the outgoing transition of state *idle* triggered by *insertWeight(w)* in figure 2. The DNF of the guard condition of the transition leading from state *idle* to state *button pressed* in figure 2 contains two conjunctions. Consequently, the test case tree contains two corresponding paths (middle and right one). A complete description of the construction algorithm can be found at [16]. The tree size will grow exponentially depending on the number of transitions and the number of conjunctions in DNF. It's size is potentially infinite. Therefore, we plan to substitute the presented tree by a finite graph that unites the nodes that represent the same state of the state machine.

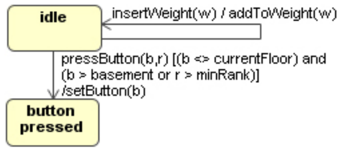
idle
insertWeight(w) / addToWeight(w)
pressButton(b,r) [(b <> currentFloor) and (b > basement or r > minRank)] /setButton(b)
button pressed

Figure 2. Fragment of the elevator model.

4 Quality of Generated Test Set

Coverage criteria can be categorized in groups that focus on certain aspects like conditions or partition boundaries, each. Criteria inside such groups are connected via the subsumption relation. For instance, *Multiple Condition Coverage* subsumes *MC/DC* [5] and *Multi-Dimensional* subsumes *One-Boundary* [11]. Criteria from different groups are often unconnected. To our knowledge, there is no approach to combine condition-based and boundary-based coverage criteria in the area of specification-based testing.

In the following, we introduce coverage criteria from both groups and combine them to new coverage criteria. The generated test sets satify such criteria. However, their concrete test values are generated from input value partitions of the test case tree. Consequently, the tree has to provide corresponding input value partitions. Such partitions are said to be *generated to satisfy a condition-based criterion*. In the following, we show how to adapt the tree in figure 3 to generate test sets for condition-based criteria.

4.1 Boundary-based Coverage Criteria

Test suites satisfy boundary-based coverage criteria like *All-Boundaries (AB)*, *All-Edges (AE)*, or *Multi-Dimensional (MD)* [11] iff the concrete values are selected close to the boundaries of a given partition. For specification-based testing, however, we do not restrict ourselves to discrete or finite domains.

Our approach is focused on the generation of partition boundaries [16]. The generated concrete test suites are aimed at satisfying MD. In [11], MD coverage has the highest fault-detection effectiveness for given partitions. However, MD does not include condition-based coverage criteria because boundary-based criteria do not consider the origin of the underlying partitions. Consequently, we define new criteria to combine the advantages of both groups of criteria. We use the tree in figure 3 to demonstrate the necessary adaptions to our test generation process.

4.2 Condition Coverage (CC)

The condition coverage criterion is focused on atomic conditions of expressions. To satisfy this criterion, a test suite has to satisfy and violate each atomic condition at least once, respectively. The test suite generated from the test case tree in figure 3 does not satisfy condition coverage. As each generated test case of the tree satisfies the expressions of one path of the tree, e.g. the condition $b <> currentFloor$ is never false if the event *pressButton(b,r)* is triggered.

For the generated test suite to satisfy condition coverage, the test tree generation process adds expressions to the tree that violate the contained atomic conditions. It is not sufficient to generate test cases with values outside of partitions because in general it is undecidable, which conditions and expressions of the control flow will be violated. Figure 4 shows the new expressions at additional outgoing arcs from the node *idle_1* in figure 3. With these new expressions, the generated concrete test cases also contain input values that violate the conditions. Consequently, the generated test set satisfies condition coverage. This is of vital importance for the selection process of input partition boundaries. It enhances the quality of our automatically generated test suites which already satisfy boundary-based coverage criteria.

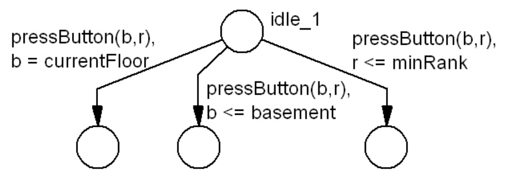

Figure 4. Additional conditions in the tree.

4.3 Decision Coverage (DC)

The decision coverage criterion is focused on values of expressions. A test suite that satisfies this criterion has to satisfy and violate each expression at least once. The test suite generated from the test case tree in figure 3 does not satisfy this criterion. For instance, no test case has input values that result in the violation of the guard condition for the event *pressButton(b,r)* in figure 2. Adding the negated existing expressions to the tree is sufficient to modify the coverage of the generated test set correspondingly. Figure 5 shows the additional expression that conflicts with the value of the existing guard condition in figure 2. With such additional expressions, the test case tree contains all original expressions and their conflictive counterparts. Since each generated test case satisfies the expressions of one path, the whole test set satisfies decision coverage. Again, this enhances the quality of the test suite generated from the test case tree.

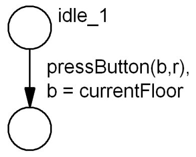

Figure 5. Additional decisions in the tree.

4.4 Modified Condition/Decision Coverage (MC/DC)

A test suite that contains every permutation of atomic condition values for each expression satisfies Multiple Condition Coverage (MCC). This criterion is considered inefficient because it usually contains many unnecessary test cases with value permutations, for which small deviations have no impact on the expression value (decision).

The Modified Condition/Decision Coverage criterion (MC/DC) [5] only requires that relevant condition value combinations are included. A condition value combination is relevant if changing one certain condition value leads to a different decision value. After adding all condition value permutations to the tree like in section 4.2 and 4.3, the generated test set satisfies MCC. Without the irrelevant value combinations, the remaining test suite satisfies MC/DC. MC/DC is very important and, for instance, required for software quality in airborne systems and equipment certification (standard RTCA/DO-178B). Since MC/DC is so important, the satisfaction of this criterion is of high importance for the selection of input partitions. This is another quality improvement of the test suite generated from the test case tree.

4.5 Definition of New Coverage Criteria

In the previous sections, we showed how to incorporate well-known coverage criteria into our test-generation process. For specification-based testing, however, these criteria are only partially adequate. Therefore, we combine both boundary-based coverage criteria and condition-based coverage criteria. This is motivated by the lack of condition-based coverage criteria to focus on boundaries and by the lack of boundary-based coverage criteria to consider the origin of partitions.

We define that a test set satisfies the *Multi-Dimensional Condition Coverage (MDCC)* if it contains all boundary values satisfying MD for partitions generated to satisfy CC. Likewise, we define *Multi-Dimensional Decision Coverage (MDDC)* for partitions generated to satisfy DC and *Multi-Dimensional Modified Condition/Decision Coverage (MDMC/DC)* for MC/DC. The advantage of the test sets that satisfy such criteria is the focus on the boundaries for partitions that are relevant for the model's conditions.

5 Evaluation

In this section, we evaluate the efficiency of the referenced coverage criteria via mutation testing. We use the example model and evaluate the criteria's efficiency with ParTeG [16]. For that, we define mutation operators to inject errors in the SUT. They change the relation symbol of conditions (e.g. $b > basement$ to $b < basement$) or change the constant values of *basement*, *currentFloor*, and *minRank* by 1, which results in 10 mutants.

We compare **(1)** MD for partitions that just satisfy Transition Coverage, **(2)** MC/DC without boundary testing, and **(3)** MDMC/DC.

The test suite generated for **(1)** contains 7 test cases and kills 4 out of 10 mutants. Errors that imply a wrong value for *minRank* cannot be found with this test set (e.g. $minRank = 3$ in the model, $minRank = 2$ in the SUT).

The test suite satisfying **(2)** contains 9 test cases with values randomly selected from the generated partitions. Since these values are chosen randomly, the test set cannot guarantee to find especially the changes near the boundaries. In our test run, this test suite kills 6 mutants.

The generated test suite that satisfies **(3)** contains 10 test cases and kills all 10 mutants. The used input values are closely above and closely below the values changed by the mutants. Consequently, even small value deviations are detected and MDMC/DC has the highest error-finding efficiency of the referenced coverage criteria.

The support of MC/DC for our prototype is not finished yet. For instance, the test suites would not detect changes of the values of *minWeight* or *maxWeight*. Currently, we are extending our tool ParTeG to support all mentioned control-flow coverage criteria.

6 Conclusions and Future Work

In this paper, we handled coverage criteria that are focused on boundaries and conditions. We sketched how to adapt the test case tree such that the automatically generated test sets satisfy these coverage criteria. Furthermore, we defined new coverage criteria which are more appropriate to specification-based test generation.

We also presented the current state of the prototype tool ParTeG [16] with its support of the evaluated coverage criteria.. In the future, we plan to extend this tool to support all presented combinations of coverage criteria. Since there are many possible combinations of boundary-based and control-flow coverage criteria, we evaluate further criteria in future work.

Acknowledgements. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

References

[1] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. In *IWTCS'97*, pages 75–90, 1997.

[2] L. C. Briand, Y. Labiche, and J. Cui. Automated support for deriving test requirements from UML statecharts. *Software and Systems Modeling*, V4(4):399–423, November 2005.

[3] L. C. Briand, Y. Labiche, and Q. Lin. Improving statechart testing criteria using data flow information. In *ISSRE '05*, pages 95–104, 2005.

[4] K. T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *DAC'93*, pages 86–91. ACM Press.

[5] J. Chilenski and S. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. In *Software Engineering Journal*, 1994.

[6] Z. R. Dai, P. H. Deussen, M. Busch, L. P. Lacmene, T. Ngwangwen, J. Herrmann, and M. Schmidt. Automatic Test Data Generation for TTCN-3 using CTE. In *ICSSEA*, 2005.

[7] W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 1999.

[8] A. Hamie, F. Civello, J. Howse, S. J. H. Kent, and R. Mitchell. Reflections on the object constraint language. In *UML 1998, Mulhouse, France*, pages 162–172, 1999.

[9] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. In *Software Testing, Verification and Reliability*, 2002.

[10] R. S. Inc. Reactis. http://www.reactive-systems.com.

[11] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *ISSRE '04*, pages 139–150. IEEE, 2004.

[12] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *FME*, pages 21–40, 2002.

[13] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML'99*, pages 416–429, 1999.

[14] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05*.

[15] M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *ER*, pages 449–464, 1998.

[16] S. Weißleder. ParTeG (Partition Test Generator). http://parteg.sourceforge.net.

[17] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In *PSI03*. Springer-Verlag, 2003.

[18] Telelogic. Rhapsody Automated Test Generation. http://www.telelogic.com.

[19] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

[20] VerifySoft Technology. Conformiq Test Generator. http://www.verifysoft.com/.

[21] S. Weißleder and B.-H. Schlingloff. Automatic Test Generation from Coupled UML Models using Input Partitions. In *MoDeVVa*, 2007.

[22] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.

[23] P. Ziemann and M. Gogolla. Validating OCL specifications with the USE tool — an example based on the BART case study. In *FMICS'2003*, 2003.