Automatic Test Generation from Coupled UML Models using Input Partitions

Stephan Weißleder and Bernd-Holger Schlingloff

Humboldt-Universität zu Berlin, 12489, Germany,
{weissled, hs}@informatik.hu-berlin.de

Abstract. In this paper, we deal with model-based automatic test generation. We show how to use coupled models consisting of UML state machines, class diagrams, and OCL expressions to automatically derive partitions of input ranges for boundary testing. We present a test generation algorithm, describe an implementation of this algorithm, and compare this implementation to Rhapsody's ATG.

1 Introduction

Modeling languages like UML [8] are widely used for system development. They are supported by many tools, some of which also provide model-based automatic generation of test suites [22, 11, 25]. This is advantageous compared to conventional test suite generation, because the automation increases efficiency during product evolution.

We argue that the current approaches neglect the generation of input partitions. Therefore, we present an approach that focuses on the generation of input partitions from UML state machines and class diagrams. We use the OCL expressions [7] of both diagrams to generate test input value partitions, which can be used to find deviations of these constraints in the system under test.

The effect of partition testing and boundary testing depends on appropriate coverage criteria (e.g. *all-edges* [13]) and on the adequate selection of partition boundaries. Usually, this selection is done manually. Therefore, it is error-prone and there is a high probability that the tests are ineffective. In contrast to the manual selection of input value boundaries, we derive them automatically from OCL expressions of system models. First, we statically analyze the interdependence between the elements of different OCL expressions within the system models. Then, we transform system models into a transition tree and investigate the tree's paths. We demonstrate our approach by the example of a sorting-machine. The main contribution of this paper is the model-based automatic generation of input value partitions.

The paper is organized as follows. In section 2, we introduce the system models which are used in this paper. We introduce the intermediate transition tree in section 3, and describe our test generation algorithm in section 4. We evaluate our approach in section 5 and summarize our results in section 6.[1]

[1] Our work is part of the Graduate School METRIK and founded by the Deutsche Forschungsgemeinschaft DFG.

Related Work. References to model-based testing and partition testing can be found in [1, 2, 24]. Hierons et al. [10] use conditioned slicing to check given input partitions. Dai et al. [6] use partition testing and rely on the user to provide the input partitions. Our approach differs in that we create input partitions, instead of relying on given ones. Legeard, Peureux and Utting [14] develop a method for automated boundary testing from the textual languages Z and B. In contrast, we focus on the graphical language UML.

OCL is the object of many studies [18, 26]. It can be used for contract-based design for which Traon [23] also defines vigilance and diagnosibility but does not use it for test case generation. Hamie et al. [9] consider OCL in the context of state machines and classes. Our approach analyzes OCL expressions to automatically generate test input value partitions.

Formalisms from outside the UML (e.g. extended finite state machines [3, 5]) also support automatic test generation but are not designed for object-oriented systems. Offutt and Abdurazik [17] generate test cases from state machines. However, they focus on single transitions and random source state initialization paths. Sokenou [20] alters this random approach and uses sequence diagrams. Our algorithm deviates in that we derive test input value partitions. In [4], Briand et al. consider data flow for testing criteria. We also extract the data flow along paths of the state machine, but use it for test generation.

To derive test cases, we create an intermediate control-flow tree, the test case tree. Kansomkeat and Rivepiboon introduce a Test Flow Graph in [12]. Our tree contains also pre/post conditions of operations, all its transitions may possess conditions. Besides that, each node of our tree holds information about the input values that are necessary for reaching the respective node.

Many commercial tools support testing. The Conformiq Test Generator [25] focuses on UML state machines. Its main strengths are parallelism and concurrency, but input values are created manually. The tool AETG [21] also examines input values, but its boundary testing algorithm depends on user-defined values and boundaries. In contrast, we derive input partitions automatically. Rhapsody ATG [22] is a tool with which test cases can be generated and executed with respect to UML state machines. These test cases are selected according to certain coverage criteria like MC/DC. The tool LTG/UML [24] from Leirios [15] also includes OCL expressions to generate test cases. To our knowledge, no commercial tool creates test cases by explicitely deriving input partitions from conditions.

1.1 Example: The Sorting-Machine

Here, we briefly introduce our reoccuring example of a sorting-machine. The context of this machine is a post office where incoming items are wrapped up. Due to this packing, the original width of the object is doubled by foam plus two extra size units for each side of a plastic box ($m\_width = (object.width + 2) * 2$). The height is handled likewise. If wrapped-up items violate the necessary sizes for the standard shipping container, extra containers are needed. Our sorting-machine's task is to sort incoming items depending on the size after their wrapping so that they fit into given containers.

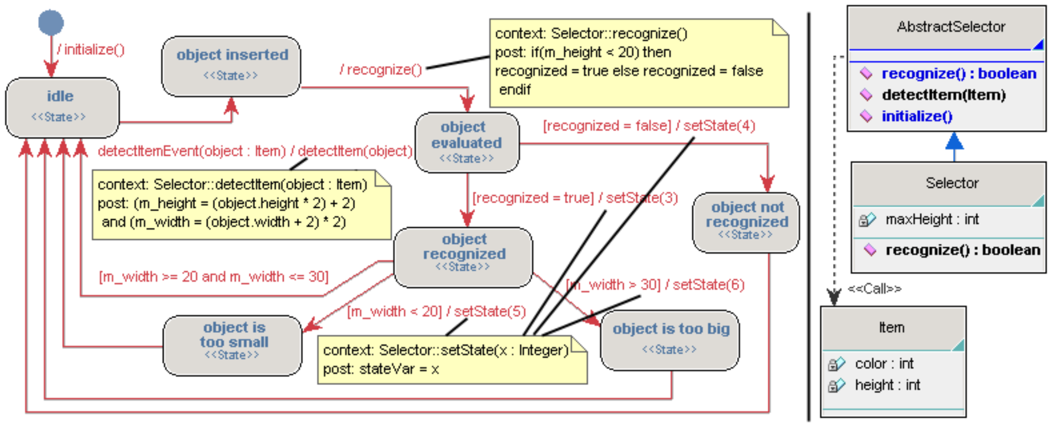

/ initialize()
idle
<<State>>
object inserted
<<State>>
/ recognize()
context: Selector::recognize()
post: if(m_height < 20) then
recognized = true else recognized = false
endif
object evaluated
<<State>>
[recognized = false] / setState(4)
detectItemEvent(object : Item) / detectItem(object)
context: Selector::detectItem(object : Item)
post: (m_height = (object.height * 2) + 2)
and (m_width = (object.width + 2) * 2)
[recognized = true] / setState(3)
object not recognized
<<State>>
object recognized
<<State>>
[m_width >= 20 and m_width <= 30]
[m_width > 30] / setState(6)
[m_width < 20] / setState(5)
object is too small
<<State>>
context: Selector::setState(x : Integer)
post: stateVar = x
object is too big
<<State>>
AbstractSelector
recognize() : boolean
detectItem(Item)
initialize()
Selector
maxHeight : int
recognize() : boolean
<<Call>>
Item
color : int
height : int

Fig. 1. Sorting-machine

Fig. 1 shows the state machine and class diagram of such a sorting-machine. The sorting is realized once in the postcondition of *recognize*() and once in the guard conditions of the outgoing transitions of state *object recognized*.

Automating the comparison of expected and actual behavior of a system is important for automatic test generation. It can be realized by checking the output of behavior or by checking its corresponding conditions. Unfortunately, these checks often do not hold true for just one operation and, thus, the satisfaction of a condition does not identify an operation (i.e. the system behavior) unambiguously. We fix this by extending model and system (e.g. with aspect-oriented techniques). In Fig. 1, we inserted an additional operation $setState(x : Integer)$ whose effect identifies the corresponding target state unambiguously.

1.2 Partition Testing and Boundary Testing

Partition testing and boundary testing are well-known testing techniques and are often used together: partitioning test input into domains is a prerequisite of focussing tests on the corresponding domain boundaries. There are many applications for partition testing (e.g. control systems for nuclear reactors, Geo-information systems, sorting-machines). In such cases, the exact values of boundaries (sticks in reactors, global position of elements, measures of objects) are important. We deal with test generation for such kind of applications.

2 The UML Models: Class Diagrams and State Machines

Our test generation approach uses UML class diagrams, UML state machines, and OCL. A formalization of UML and OCL has been given in [7, 8]. Subsequently, we recall those definitions that are most important for our purpose. For an example, consider the sorting-machine given in Fig. 1.

Class Diagrams. A class diagram $cd = (CLS, REL)$ consists of classes CLS and relations REL between classes (right part of Fig. 1: boxes depict classes, arrows depict relations). A class $c \in CLS$ contains a set of attributes AT and a set of operations OP: $c = (AT, OP)$. Each operation $op \in OP$ has a precondition $op.pre$ and a postcondition $op.post$. The condition $op.pre$ must be met before the op's execution, $op.post$ defines the condition that is met after op's execution.

State Machines. A state machine sm consists of a set of regions r, which consist of a set of vertices $VERT$ and a set of transitions TRS: $r = (VERT, TRS)$. On the left side of Fig. 1, arrows denote transitions, which connect vertices (e.g. *idle*). Each vertex $v \in VERT$ may possess a name $v.sn$, a set of incoming transitions $v.INC$, a set of outgoing transitions $v.OUT$, and an invariant $v.inv$. Each transition $t \in TRS$ has a source vertex $t.sv \in VERT$, a target vertex $t.tv \in VERT$, an event $t.ev$, a guard $t.guard$, and an effect $t.ef$. We interpret events ev solely as call events, because most object-oriented programming languages have to realize them by operation calls anyway. A *guard* is a boolean expression without side-effects. The effect ef is of type *Behavior* - in our approach, an operation call of the assciated class (see Fig. 1).

Conditions. The conditions $COND$ in our approach are Boolean OCL expressions. They consist of basic predicates (e.g. (in-)equations or Boolean attributes), which are connected by Boolean operators (e.g. *and*, *or*). The elements of the predicates can navigate along association relations between classes. In Fig. 1, the folded boxes contain OCL expressions. The attached lines show their assignment to effects of transitions. Furthermore, OCL provides expressions on operation calls and on collections. Due to space restrictions, these OCL expressions are not examined here.

Coupling Models. As shown in Fig. 1, we use models consisting of both state machines and class diagrams. We call a pair of state machine and class diagram a *coupled model*. This pair is connected by references from the effects of transitions to operation calls. Navigation along inheritance relations helps reusing state machines. According to Liskov's substitution principle [16], properties of a base class also hold for its subclasses. State machines are behavioral properties of a class. Thus, they can be reused in the subclass of a class (this time referencing the operations and the attached OCL conditions of the subclass).

3 Test Case Trees

In this section, we define a finite tree for test case generation.

A **Test Case Tree** tct consists of a set NOD of nodes and a set ARC of directed arcs: $tct = (NOD, ARC)$. Each node is either a state ST or an intermediate state IST. Furthermore, each $n \in tct.NOD$ contains a set of incoming arcs $n.IN$ and a set of outgoing arcs $n.OUT$, and a set of parameter ranges $n.RANGE$, which maps input event parameters to ranges of values: $n = (IN, OUT, RANGE)$. Each $s \in tct.ST$ contains an additional reference to a corresponding state in the state machine (to refer to the state invariant). The tree's root is a node $sroot \in tct.ST$ with $sroot.IN = \emptyset$. For all other states

$as \in ST \cup IST$ it holds that $|as.IN| = 1$. All leaves l are elements of ST and satisfy $l.OUT = \emptyset$.

Each $arc \in tct.ARC$ connects a state and an intermediate state. It may possess a precondition $arc.pre \in COND$, a postcondition $arc.post \in COND$ (default: *true*), and a parameterized event $arc.ev$ (default: instant state change). A small example of a test case tree is shown in Fig. 2.

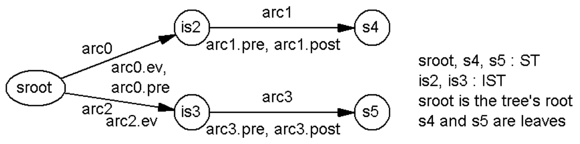


Fig. 2. Example of a test case tree

A tree's path leads from *sroot* to a leaf. All nodes on a path are ordered. So, $m \in tct.ARC$ is a ***preceding arc*** of arc $n \in tct.ARC$ iff n can be reached from m by following the arcs. The paths of the test case tree are used to derive test cases.

Each **Test Case** corresponds to a path from *sroot* to a leaf $l \in ST$. The input for a test is a parameterized operation call sequence corresponding to the operation call sequence and a representative of the input parameter range $l.RANGE$. Expected and actual system behavior is compared by evaluating the conditions along the path.

We describe the generation algorithm of *l.RANGE* in *Step 2* of section 4.2.

4 Test Generation

The test generation algorithm transforms a *coupled model* into a test case tree and creates test input partitions by evaluating OCL expressions. Afterwards, the algorithm derives concrete test input values from these partitions.

4.1 Classification of OCL expressions

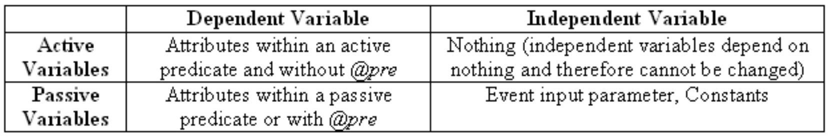
In this section, we present a classification of OCL expressions, which is related to the one used in the Leirios methodology [15]. The corresponding Leirios tool LTG/UML [24] can evaluate OCL expressions like pre/post conditions or transition guards. All atomic predicates in LTG/UML are either *active* or *passive*: Only active predicates can alter the value of attributes, the passive ones can only be read. Active predicates are only allowed within postconditions of operations.

Because we focus on the values of input variables, our classification is slightly different from the one of Leirios. Our basic unit is a variable *var*. It is part of an atomic predicate, which is in turn the *context predicate* of *var*. Each predicate consists of variables, relations between them, and operations on them. We classify the system model's variables (attributes, input parameters, or constants) and use Leirios' classification to introduce dependent and independent variables. As in LTG/UML, we assume that variables not stated in postconditions are unchanged. In contrast to LTG/UML, we do not take equations between two active predicates (or variables) as assignments. In OCL, the meaning of such equations is restricted to the fact that both predicates are equal - the concrete way to reach this equality is undefined.

Subsequently, we define kinds of variables and their mutual relations.

Definition 1 (Independent and Dependent Variables). *An **independent variable** is either an event input parameter or a constant class attribute. Its value is constant. A **dependent variable** is a non-constant class attribute.*

We define that a variable *var* is either active or passive corresponding to Leirios' definition concerning *var*'s context predicate: If *var*'s context predicate is active and @*pre* [7] is not attached to *var*, then *var* is active and can be changed in this expression. If either *var*'s context predicate is passive or *@pre* is attached to *var*, then *var* is passive and cannot be changed. Fig. 3 shows the relation between active/passive variables and dependent/independent variables.



	Dependent Variable	**Independent Variable**
Active Variables	Attributes within an active predicate and without *@pre*	Nothing (independent variables depend on nothing and therefore cannot be changed)
Passive Variables	Attributes within a passive predicate or with *@pre*	Event input parameter, Constants

Fig. 3. Classification of variables

With definition 1, we can now describe the dependency between the OCL expressions along a given path. If an arc $a \in tct.ARC$ contains a condition consisting amongst others of a variable *var*, then *a* contains *var*.

Definition 2 (Next Preceding Arc). *If an arc $a1 \in tct.ARC$ contains a dependent passive variable var, then the **next preceding arc** of a1 w.r.t. var is a1's preceding arc $a2 \in tct.ARC$, that is closest to a1 and contains var as an active variable.*

Definition 3 (Initialized Variables). *Each independent variable is **initialized**. A dependent passive variable depvar of a condition $cond \in COND$ assigned to an arc $a2 \in tct.ARC$ is **initialized** iff the next preceding arc $a1 \in tct.ARC$ of a2 w.r.t. depvar exists and, except for depvar, depvar's context predicate in a1.post contains only initialized passive variables.*
*A condition cond assigned to an arc a2 is **initialized** iff each variable in cond is initialized. The set consisting of cond and all conditions of all next preceding arcs w.r.t. each dependent variable along the path from the root to a2 is the **initialized condition set** of cond.*

Theorem 1 (Reducible Variables). *In an initialized condition set of a condition, each included dependent variable can be reduced to independent variables.*

Proof. A variable var is initialized iff var's context predicate of the next preceding arc $nparc$ w.r.t. var contains besides var only passive initialized variables. These variables are dependent or independent. Because all variables are initialized (definition 3), a dependent variable $depvar$ can only occur as long as there is a next preceding arc w.r.t. $depvar$. Otherwise, $depvar$'s context predicate in $nparc$ contains only independent variables. Each outgoing arc $oarc \in sroot.OUT$ has no next preceding arc, because $sroot.IN = \emptyset$. So, each $oarc$ contains no dependent passive initialized variables, and all dependent variables of an initialized condition set depend directly or indirectly on independent variables. □

In the following, we assume that all variables are initialized.

4.2 Creating the Test Case Tree

The algorithm to create the test case tree is similar to existing transformation approaches in [2]: it iterates along the state machine's transitions and adds newly created nodes and arcs to the test case tree. Within each step of the iteration, the algorithm evaluates the OCL conditions in the newly created arcs of the test case tree tct: all of $cond$'s variables are classified into independent and dependent variables (definition 1).

We detail the transformation process in two steps: the creation of the test case tree tct (*Step 1*) and the evaluation of the OCL conditions (*Step 2*) to create input value ranges $RANGE$ in tct's nodes. At first, tct consists only of the root node $sroot$.

Step 1. The algorithm starts at the root $sroot$ of tct and at the state $S1$ following the initial pseudostate of the state machine (see Fig. 4). For each transition $t \in S1.OUT$, we insert an intermediate state is into $tct.IST$ and an arc $arc1$ into $tct.ARC$, so that $arc1 \in sroot.OUT$ and $arc1 \in is.IN$. Subsequently, t's triggering event $t.ev$ and t's guard $t.guard$ are attached to $arc1$: $arc1.ev = t.ev$; $arc1.pre = t.guard$. After this, we insert a new state s into $tct.ST$ and another arc $arc2$ into $tct.ARC$, such that $arc2 \in is.OUT$ and $arc2 \in s.IN$.

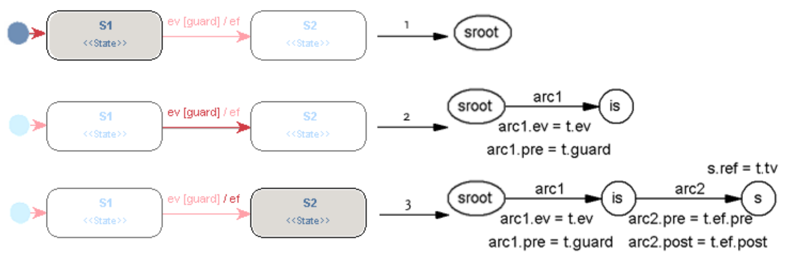
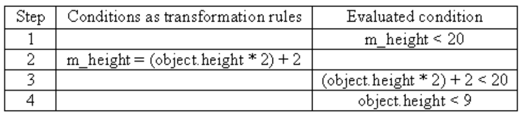


Fig. 4. Creation of the test case tree for one transition of the coupled model

The conditions of t's effect $t.ef$ are assigned to *arc2*: $arc2.pre = t.ef.pre$ and $arc2.post = t.ef.post$. We copy $sroot.RANGE$ to $is.RANGE$ and to $s.RANGE$ and let s refer to t's target vertex $t.tv$. Subsequently, we execute *Step 2* for *is* and for s successively. The whole procedure is repeated for all outgoing arcs and their target states. Fig. 4 shows the creation process for just one transition. The branching in the resulting test case tree depends on the branching of the source state machine's transitions. The current termination criterion is forming a circle in the control flow of the original state machine.

Step 2. In this step, we evaluate all conditions of each $arc \in tct.ARC$ that was just inserted into the test case tree. Starting from each just added condition *cond*, we gather the initialized condition set of *cond*. We use this condition set (except *cond*) together with transformation rules to transform the condition *cond* until it solely contains independent variables. These transformation rules define patterns for *cond* and for the conditions of the condition set, which must be matched to create a transformed *cond*. The transformation rules are solely expedient if they use the conditions from the condition set to replace dependent variables in *cond* with independent variables reasonably. For instance, a is a dependent variable and *cond* is $a < b$. Then a predeceding postcondition $a = c$ could be used to transform $a < b$ into $c < b$, whereas a preceding postcondition $a < c$ is useless to relate c and b.

In our example (Fig. 1), we consider a short path from *idle* via *object inserted* to *object evaluated*. We just insert the postcondition of *Selector::recognize()* and evaluate it using the postcondition of *Selector::detectItem*. We first consider that the if-condition in the newly inserted condition is evaluated to *true*. So, we transform the if-condition with the help of the conditions in the corresponding initialized condition set (see the steps in Fig. 5).



Step	Conditions as transformation rules	Evaluated condition
1		m_height < 20
2	m_height = (object.height * 2) + 2	
3		(object.height * 2) + 2 < 20
4		object.height < 9

Fig. 5. Stepwise transformation of evaluated condition

The transformed guard conditions contain new restrictions for the value ranges of input parameters to reach *arc*'s target node $n \in tct.NOD$. These new conditions are intersected with $n.RANGE$, which was created in *Step 1*.

Creating the Test Cases. To create test cases from the test case tree *tct*, we iterate over all leaf nodes $l \in tct.ST$ and gather the parameter ranges $l.RANGE$. Within each step, we iterate over all nodes from *tct*'s root to l and trigger each input event along the arcs with parameters corresponding to the value ranges within $l.RANGE$: for the example in Fig. 5, we choose a value smaller than 9 for the input attribute *object.height*. We use the boundary testing method [19] to select the exact values: the boundary values itself, the next inner values, and some random values from within the value range. Expected and actual system behavior are compared by evaluating all conditions available along the path: all referred state invariants, guards, preconditions, and the results of the postconditions (depending on the actual input parameter values). The resulting test cases cover the input partitions of those variables that also appear in our system model (see *completeness* in [23]).

5 Comparison with ATG.

The described algorithm is realized in the prototype *PARTEG* (Partition Test Generator). We use the example in Fig. 1 to compare *PARTEG* with Rhapsody's ATG, which is a popular tool in the field of automated test generation. We compare both tools via mutation analysis. For the commercial tool Leirios LTG/UML, the available documentation claims that it can also generate test cases from UML models similar to the ones in our approach. Due to licensing restrictions, however, this tool could not be compared to our prototype.

Mutation Operators. We define two mutation operators. Both fit to boundary testing and to the kind of expressions that our algorithm can evaluate: they exchange relation symbols ($\leq$ for $<$ and $\geq$ for $>$) respectively shift the boundaries of the conditions with arbitrary values (here: 2 and 6). So, the resulting mutants are completely killed iff the test set checks violations of the conditions given in the model. We combine both mutation operators and receive 24 distinct, identifiable mutants (3 inequations with 9 mutants each: 4 for shifting boundaries, 5 for also changing the relation symbol; 3 mutants overall are not distinguishable from the original).

Comparison. We model the example of Fig. 1 in *PARTEG* and in Rhapsody's ATG. In ATG, we can not use pre/post conditions. Additionally, ATG is restricted to the domain of C++ and cannot evaluate OCL. So, we add the postconditions as operation implementation code. ATG generates 4 test cases that cover all transitions but it identifies only 10 out of 24 mutants.

PARTEG's implementation realizes different test criteria like *all-boundaries* and *all-edges* [13]. With minimal configuration, *PARTEG* creates 5 test cases overall that identify all 24 mutants. Manual inspection shows that no test case of the generated test suite is redundant. For this example, our approach identifies all mutants with the minimum number of test cases necessary. Although we still need more elaborate studies, this result shows the prospective strengths of our approach.

6 Conclusion and Future Work

In this paper, we used UML state machines and class diagrams to derive test input partitions automatically. We pointed out the importance of partition testing when dealing with numeric data, named application fields, and showed the potential of our approach with a prototype. In the future, we aim at less restrictive approaches to use OCL and UML, and we want to use our method in the domain of Geo-information systems.

References

1. B. Beizer. *Software Testing Techniques.* John Wiley & Sons, Inc., 1990.
2. R. V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
3. C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols.
4. L. C. Briand, Y. Labiche, and Q. Lin. Improving statechart testing criteria using data flow information. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 95–104, Washington, DC, USA, 2005. IEEE Computer Society.
5. 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.
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, 2005.
7. Object Management Group. Object Constraint Language (OCL), version 2.0, 2005.
8. Object Management Group. Unified Modeling Language (UML), version 2.1, 2007.
9. A. Hamie, F. Civello, J. Howse, S. J. H. Kent, and R. Mitchell. Reflections on the object constraint language. In *Proc. International Conference on the Unified Modelling Language (UML) 1998, Mulhouse, France.* Springer-Verlag, 1999.
10. R. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing, 2002.
11. Reactive Systems Inc. Reactis. http://www.reactive-systems.com.
12. S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *SAICSIT '03*, pages 296–300, 2003.
13. 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 Computer Society.
14. B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B, 2002.
15. Leirios. LTG/UML. http://www.leirios.com.
16. Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, 1988.
17. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML'99*, volume 1723, pages 416–429. Springer, 1999.
18. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *Proc. 17th International Conference on Conceptual Modeling (ER)*, volume 1507, pages 449–464. Springer-Verlag, 1998.
19. P. Samuel and R. Mall. Boundary Value Testing based on UML Models. In *ATS '05*, pages 94–99. IEEE Computer Society, 2005.
20. D. Sokenou. Generating Test Sequences from UML Sequence Diagrams and State Diagrams. In *INFORMATIK 2006*, pages 236–240, 2006.
21. Telcordia Technologies. AETG. http://aetgweb.argreenhouse.com.
22. Telelogic. Rhapsody Automated Test Generation. http://www.telelogic.com.
23. Yves Le Traon. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, 2006. Member-Benoit Baudry and Member-Jean-Marc Jezequel.
24. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
25. VerifySoft Technology. Conformiq Test Generator. http://www.verifysoft.com/.
26. P. Ziemann and M. Gogolla. Validating OCL specifications with the USE tool — an example based on the BART case study, 2003.