Deriving Input Partitions from UML Models for Automatic Test Generation

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 UML state machines, UML class diagrams, and OCL expressions to automatically derive partitions of input parameter value ranges for boundary testing. We present a test generation algorithm and describe an implementation of this algorithm. Finally, we discuss our approach and compare it to commercial tools.

1 Introduction

Modeling languages like the Unified Modeling language (UML) [7] are widely used for system development. They are supported by many tools, some of which also provide model-based automatic generation of test suites [10, 22, 25]. This is advantageous compared to conventional test suite generation because the automation increases the efficiency of the test generation process.

We argue that the current approaches neglect the generation of input partitions. Therefore, we present an approach that is focused on the generation of input partitions from UML state machines and UML class diagrams. It derives test input value partitions from expressions of both diagrams, e.g. transition guards or pre-/postconditions of the Object Constraint language (OCL) [6]. The corresponding test suite is focused on detecting errors that result from differences between constraints in the model and constraints in the system under test (SUT).

The quality of test suites created with partition testing and boundary testing depends on satisfied coverage criteria and on the adequate selection of partition boundaries. Usually, the latter is done manually. Therefore, the boundary selection is error-prone and there is a high probability that the test effectiveness is low. In contrast to manual selection of input value boundaries, we derive them automatically from OCL expressions of system models. We statically analyze the interdependence of OCL expressions within the system model and transform the model into a transition tree and investigate the tree's paths. We demonstrate our approach by the example of a sorting-machine. Compared to other approaches, the contribution of this paper is a method to generate test cases by evaluating OCL expressions in postconditions, which are not restricted to equations.

The paper is organized as follows. Sections 2 and 3 contain preliminaries for this paper and the used example system models. Section 4 contains the intermediate transition tree. The test generation approach is described in Section 5. Sections 6 to 8 contain evaluation, related work, and summary.

2 Preliminaries

In this section, we introduce the running example of a sorting machine and describe the use of partition testing and boundary testing.

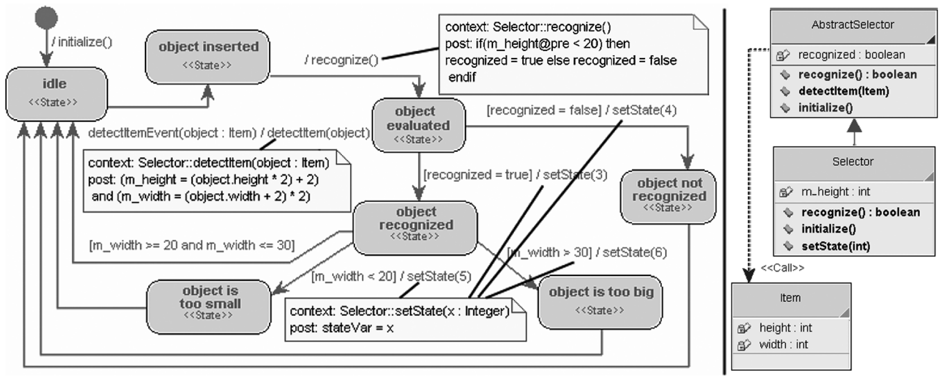
2.1 Example: The Sorting-Machine

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 transport containers.

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

2.2 Partition Testing and Boundary Testing

Partition testing and boundary testing are well-known testing techniques and are often used together: partitioning test input parameters into value domains is a prerequisite of focussing tests on the corresponding domain boundaries. As examples for partition testing we consider control systems for nuclear reactors, Geo-information systems, or sorting machines. In such cases, the exact values of boundaries (sticks in reactors, global position of elements, measures of objects) are important. The corresponding test cases have to contain values that check even small violations of the derived test input parameter boundary values. We deal with the automatic test generation for such kind of applications.



```
context: Selector::recognize()
post: if(m_height@pre < 20) then
recognized = true else recognized = false
 endif
```

```
context: Selector::detectItem(object : Item)
post: (m_height = (object.height * 2) + 2)
 and (m_width = (object.width + 2) * 2)
```

```
context: Selector::setState(x : Integer)
post: stateVar = x
```

Fig. 1. State machine and class diagram for a sorting-machine.

3 The UML Models: Class Diagram and State Machine

Our test generation approach uses UML class diagrams, UML state machines, and OCL to generate test code. A formalization of UML and OCL has been given in [6, 7]. Subsequently, we recall the definitions that are most important for our purpose. For an example, consider the sorting-machine given in Fig. 1.

Class Diagram. A class diagram $cd = (\mathit{CLS}, \mathit{REL})$ consists of classes CLS and relations REL between classes. The right part of Fig. 1 shows boxes depicting classes and arrows depicting relations between them. A class $c \in \mathit{CLS}$ contains a set of attributes AT and a set of operations OP: $c = (\mathit{AT}, \mathit{OP})$. Each operation $op \in \mathit{OP}$ has an optional precondition $op.pre$ and an optional 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 Machine. A state machine sm contains a set of regions REG, which in turn contain a set of vertices VERT and a set of transitions TRS: $sm = (\mathit{REG})$, $\mathit{REG} = (\mathit{VERT}, \mathit{TRS})$. On the left side of Fig. 1, arrows denote transitions, which connect vertices. Each vertex $v \in \mathit{VERT}$ may possess a name $v.sn$, a set of incoming transitions $v.\mathit{INC}$, a set of outgoing transitions $v.\mathit{OUT}$, and an invariant $v.inv$. Each transition $t \in \mathit{TRS}$ has a source vertex $t.sv \in \mathit{VERT}$, a target vertex $t.tv \in \mathit{VERT}$, an event $t.ev$, a guard $t.guard$, and an effect $t.ef$. We interpret events ev solely as call events, since in most object-oriented languages events are realized by operation calls. A *guard* is a Boolean expression without side-effects. The effect ef is of type *Behavior* - in our approach, an operation call of the associated class (see Fig. 1). The example in Fig. 1 does not comprise parallelism and, therefore, contains just one region.

Conditions. The conditions COND are Boolean OCL expressions contained in state machines or class diagrams. They consist of basic predicates like arithmetic conditions, which are connected by Boolean operators. The elements of the predicates are used to 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 collections. To check the test result, OCL expressions are evaluated at run time with respect to the created objects and attributes.

Coupled Models. As shown in Fig. 1, we use models consisting of a pair of state machine and class diagram. We call such pairs *coupled models*. They are connected by references from transitions to operation calls. The constraints of both models are evaluated together. Navigation along inheritance relations helps reusing state machines. According to Liskov's substitution principle [15], properties of a 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 expressions of the subclass).

4 Test Case Tree

In this section, we define a finite tree for test case generation. It contains all necessary information to derive test sequences and test input boundary values. It simplifies the evaluation because all constraints are ordered according to control flow information. This approach also allows to use another source model if an appropriate model transformation is defined.

A **Test Case Tree** tct consists of nodes NOD and directed arcs ARC: $tct = (NOD, ARC)$. Some nodes reference a state in the state machine. They are called anchor nodes $ANOD$. Furthermore, each $n \in tct.NOD$ references incoming arcs $n.IN$, outgoing arcs $n.OUT$, and contains parameter ranges $n.RANGE$: $n = (IN, OUT, RANGE)$. $n.RANGE$ maps each input event parameter used on the path from the root to n to a range of values. Each combination of representatives of these value ranges applied to the current input event sequence parameters results in reaching n. We focus on the boundary values. The tree's root is a node $sroot \in tct.ANOD$ with $sroot.IN = \emptyset$. For all other states $as \in NOD$ it holds that $|as.IN| = 1$. All leaves l are elements of $ANOD$ and satisfy $l.OUT = \emptyset$. The arcs of the tree $arc \in tct.ARC$ possess a precondition $arc.pre \in COND$ (default: $true$), a postcondition $arc.post \in COND$ (default: $true$), and an event $arc.ev$ parameterized with instances of primitive or abstract data types.

An example for the general structure of a test case tree is shown in Fig. 2. The tree contains seven nodes connected by arcs. Each arc contains a transition event, a transition guard, an operation's precondition, or its postcondition. Each 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 and $n \neq m$.

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

Each test case has to satisfy all expressions along its corresponding path. Each arc of the tree contains just one expression. This allows to form the disjunctive normal form (DNF) of the contained expressions and split up the containing arc into several arcs corresponding to the resulting conjunctions. Since this replaces all complex Boolean expressions with conjunctions, the evaluation of all expressions of one path is simplified. We describe the generation algorithm of $l.RANGE$ in *Step 2* of section 5.2.

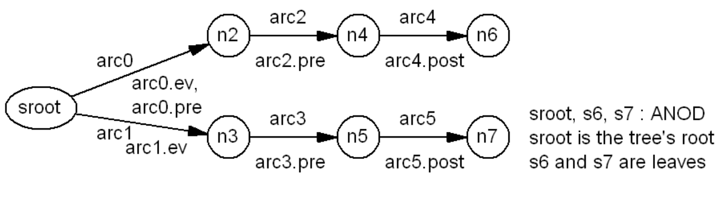


Fig. 2. General structure of a test case tree.

5 Test Generation

This section describes the test generation algorithm. First, a coupled model is transformed into a test case tree. Then, test input partitions are derived by categorizing and transforming OCL expressions of the tree. Afterwards, the algorithm generates concrete test input values from these partitions.

5.1 Classification of variables in OCL expressions

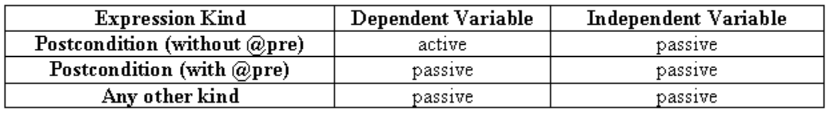
In this section, we present a classification of OCL expressions, which is partially similar to the one used in the Leirios methodology [14]. Predicates in LTG/UML are either *active* or *passive*: Only active predicates can alter the value of attributes, the passive ones can only read. Leirios claims that their tool LTG/UML [24] can evaluate OCL expressions like pre-/postconditions or transition guards. They use an operational interpretation of equations in OCL postconditions. In contrast, our approach is not restricted to equations but can also evaluate inequations. In future work, we will aim at evaluating more complex operations on collections in OCL postconditions. Additionally, Leirios defines new interpretations for OCL constraints. For instance, in active contexts of a postcondition the mere equation $X = Y$ is interpreted as an assignment of the value of Y to X, which can lead to confusions. The OCL specification [6] does not provide such an interpretation. A corresponding assignment would be $X = Y@pre$. In our approach, we stick to the OCL specification without additional interpretations. To recognize the variables that can change and those that can not, we provide a classification of the variables in OCL expressions.

Since we focus on the values of input variables, our classification differs from the one of Leirios: our atomic classification units are variables *var*. They are 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 introduce *dependent* and *independent* variables. As in LTG/UML, we assume that variables not stated in postconditions are unchanged.

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 state that a variable *var* is active or passive depending on *var*'s context predicate. If the context predicate of *var* is a postcondition and no *@pre* is



Expression Kind	Dependent Variable	Independent Variable
Postcondition (without @pre)	active	passive
Postcondition (with @pre)	passive	passive
Any other kind	passive	passive

Fig. 3. Active and passive variables.

attached to *var* then the value of *var* can be changed - it is active. In all other cases, the value of *var* can not be changed - *var* is passive. Fig. 3 shows the corresponding classification.

Using Definition 1, we are able to 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 is said to contain *var*.

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

Definition 3 (Defined Variables). *Independent variables are* defined. *Active variables are* defined *if all remaining variables contained in its context predicate are* defined. *Each dependent passive variable* depvar *contained in a condition* cond $\in$ *COND used in an arc* a2 $\in$ *tct.ARC is* defined *iff the next preceding arc* a1 $\in$ *tct.ARC of* a2 *w.r.t.* depvar *exists and the corresponding active variable is* defined.
A condition cond *used in an arc* a2 *is* defined *iff each variable in* cond *is* defined. *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* defined condition set *of* cond.

These definitions exclude the existence of two active variables in one atomic condition. Obviously, variables can be defined in atomic conditions connected by conjunctions, respectively. Consequently, the presented definitions are applicable to conjunctions. All conditions in DNF are only connected by conjunctions. Hence, expressing OCL conditions in DNF is necessary for their evaluation.

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

Proof. A variable *var* is defined iff *var*'s context predicate of the next preceding arc *nparc* w.r.t. *var* contains only passive defined variables besides the corresponding active variable *var*. These variables are dependent or independent. Since all variables are defined (Definition 3), such defined variables *initvar* can only be dependent as long as there is a next preceding arc w.r.t. *initvar*. Otherwise, *initvar*'s context predicate in *nparc* contains only independent variables. Each outgoing arc $oarc \in sroot.OUT$ has no next preceding arc, because $sroot.IN = \emptyset$. Consequently, *oarc* contains no dependent passive initialized variables. Since all paths are of finite length, all dependent variables of an defined condition set depend directly or indirectly on independent variables. □

For instance, in a postcondition $X > X@pre + Z@pre$, the values of X and $X@pre$ are different if $Z@pre \geq 0$. Roughly speaking, we consider X and $X@pre$ as different variables with different values. X is initialized iff $X@pre$ and $Z@pre$ are also initialized, which in turn depends on their next preceding arcs. In the following, we assume that all variables are initialized.

5.2 Creating the Test Case Tree

The presented algorithm is similar to existing transformation approaches presented in [2]. Within each step of the transformation, the algorithm evaluates the OCL expressions of the test case tree *tct*. We split up the transformation in two steps: the creation of the test case tree *tct* in *Step 1* and the creation of input value partitions in *Step 2*.

Step 1. The algorithm starts at the root *sroot* of *tct* and at the state *S1* after the initial pseudostate of the state machine (see Fig. 4). For each $t \in S1.OUT$, we insert a node *n1* into $tct.NOD$ and an arc *arc1* into $tct.ARC$, so that $arc1 \in sroot.OUT$ and $arc1 \in n1.IN$. The triggering event of t and t's guard are attached to *arc1*: $arc1.ev = t.ev$; $arc1.pre = t.guard$. Subsequently, we insert the new node *n2* and the state s into $tct.ST$ and add the arcs *arc2* and *arc3* into $tct.ARC$, so that $arc2 \in n1.OUT$, $arc2 \in n2.IN$, $arc3 \in n2.OUT$, and $arc3 \in s.IN$. The conditions of t's effect $t.ef$ are assigned to *arc2* and *arc3*: $arc2.pre = t.ef.pre$ and $arc3.post = t.ef.post$. We copy $sroot.RANGE$ to $n1.RANGE$, $n2.RANGE$, and $s.RANGE$ and let s refer to t's target vertex $t.tv$.

Subsequently, we transform the added expressions into DNF and split up the test case tree corresponding to the resulting conjunctions (see Fig. 5). This is reasonable because a path selection is similar to a disjunction. Dealing just with conjunctions simplifies the evaluation process, because we do not have to consider dependencies between expressions like in conditioned constraints. For instance, in *if (A) then B else C endif* we have to evaluate the value of A before knowing whether to consider B or C. Fig. 4 shows the creation process for one transition. It terminates if the transformed transitions form a circle. Although this process seems to be similar to simple unfolding, the test case tree makes the test generation algorithm in Step 2 independent of UML. It could be reused for other formalisms.

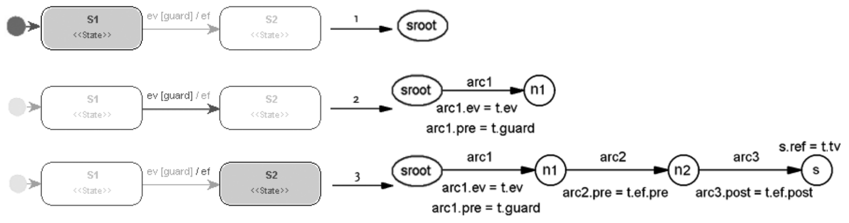


Fig. 4. Creation of the test case tree for one transition of the coupled model.

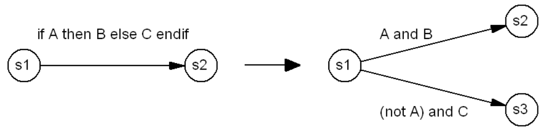


Fig. 5. Exemplary transforming an expression in DNF and adapting the test case tree.

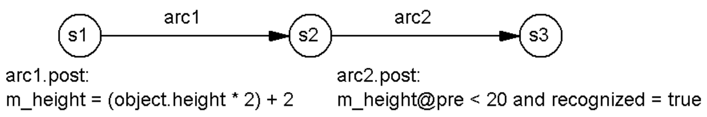
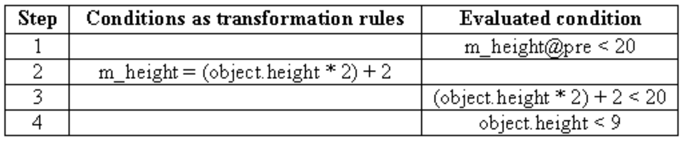

arc1
arc2
s1
s2
s3
arc1.post:
m_height = (object.height * 2) + 2
arc2.post:
m_height@pre < 20 and recognized = true

Fig. 6. Part of the test case tree for the sorting machine.



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

Fig. 7. Stepwise transformation of the evaluated condition.

Step 2. In this step, we evaluate the expressions of each $arc \in tct.ARC$ that was just inserted into the test case tree. Starting from *arc*'s condition *cond*, we compute the initialized condition set of *cond*. We use this condition set to transform the condition *cond* until it just contains independent variables. The postconditions of *cond*'s initialized condition set are used as the corresponding transformation rules.

The test generation process identifies the active variables in these postconditions. The values of active variables depend on the values of passive ones. Consequently, the conditions on dependent variables can be expressed as conditions on independent variables (see Fig. 3). This results in partitions of the value ranges of input parameters.

In the example in Fig. 1, we consider a short path from *idle* via *object inserted* to *object evaluated*. We insert the postcondition of *Selector::recognize()* and evaluate it using the postcondition of *Selector::detectItem()*. We split up the postcondition of *Selector::recognize()* in DNF conjunctions and just consider the case that $m\_height@pre < 20$ is true. Fig. 6 shows a part of the corresponding test case tree. The algorithm transforms $m\_height@pre < 20$ by using the postcondition of the next preceding arc as transformation rule (see Fig. 7).

The transformed condition contains new restrictions for the value ranges of the input parameter *object.height* to reach the corresponding target node. Such 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 $l \in tct.ANOD$. For each leaf l, we create a test input sequence corresponding to a path from *tct*'s root to l and parameterize each event with representatives of the value range $l.RANGE$. We use the boundary testing method [19] to select these representatives: e.g., boundary values, next inner values, and random values from within the value range. The result is a test suite that satisfies boundary-based coverage criteria [13]. Expected and actual system behavior are compared by evaluating all conditions available along the path. The resulting test cases comprise just the deducible input partitions of the variables in our system model (see *completeness* in [23]).

6 Evaluation of ParTeG

The described algorithm is implemented in the prototype *ParTeG* (Partition Test Generator) [18]. This tool is capable of handling arithmetic and Boolean operations within OCL expressions. The SUT is on the level of source code. In the current version, the generated test code is a JUnit test suite.

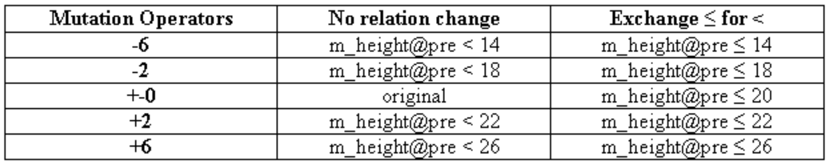
We use the example in Fig. 1 to compare ParTeG with Rhapsody's ATG and Leirios' Test Designer, which are popular tools in the field of automated test generation. We compare all generated test suites via mutation analysis. For that, we first define mutation operators to inject errors in the SUT. After that, we compare the generated test suites of all tools by the amount of killed mutants. At the end of this section, we discuss advantages and limitations of ParTeG.

Mutation Operators. Since we use mutation testing to compare the generated test suites, the selection of the mutation operators is critical for the quality of the comparison. We put emphasis on the recognition of changes to the OCL expressions. Consequently, our mutation operators change such expressions in the SUT. We define two mutation operators that exchange relation symbols ($\leq$ for $<$ and $\geq$ for $>$) respectively shift the boundaries of the conditions by small values like 2 or 6. 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). For instance, mutating the inequation $m\_height@pre < 20$ results in the mutated conditions of the SUT that are shown in Fig. 8.

Comparison to Commercial Tools. We modeled the example of Fig. 1 in Rhapsody's ATG, in Leirios' Test Designer, and in ParTeG. The latter two support all OCL expressions needed for this example. Since ATG is restricted to the domain of C++ and does not support pre-/postconditions of OCL, we added all expressions as implementation code to the model.

Rhapsody's ATG generated 4 loop-free test cases that cover all transitions but killed only 10 out of 24 mutants (see Fig. 9). Leirios Test Designer also generated 4 loop-free test cases that cover all transitions and killed 10 of 24 mutants. Interestingly, both sets of killed mutants differ from each other.

ParTeG's implementation can generate test suites that satisfy different test criteria like *State Coverage* [24] and *Multi-Dimensional* [12]. With minimal configuration, ParTeG created 5 loop-free test cases overall that killed all 24 mu-



Mutation Operators	No relation change	Exchange ≤ for <
-6	m_height@pre < 14	m_height@pre ≤ 14
-2	m_height@pre < 18	m_height@pre ≤ 18
+-0	original	m_height@pre ≤ 20
+2	m_height@pre < 22	m_height@pre ≤ 22
+6	m_height@pre < 26	m_height@pre ≤ 26

Fig. 8. Mutated conditions in the SUT.

Tool	Generated Test Cases	Indentified Mutants
Rhapsody ATG	4	10 / 24
Leirios Test Designer	4	10 / 24
ParTeG	5	24 / 24

Fig. 9. Results of the comparison.

tants. Manual inspection showed that the generated test suite contains no redundant test cases. For this example, ParTeG killed all mutants with the minimum number of loop-free test cases. Higher effectiveness could be reached by, e.g., concatenating all test cases.

Discussion. The above result shows the prospective strengths of our approach. Furthermore, there are many interesting points to discuss. For instance, the effect of a triggered transition can trigger other transitions. Since all events of state machines are handled in a pool, the triggered event can simply be added to this pool. Furthermore, the relative completeness of pre-/postconditions strongly influences the quality of the generated test suite. As any other model-based approach, our apporoach can only transform expressions into input value partitions if the model comprises the corresponding dependencies. Another important aspect is the effectiveness of our approach. The selected coverage criterion plays an important role for the overall costs: satisfying All-One-Loop-Paths is definitely more costly than satisfying All-States. Furthermore, it seems reasonable to combine boundary-based coverage criteria with transition-based coverage criteria. The size of the modeled systems is another important aspect. Until now, we have not performed larger case studies. The generation of test suites that satisfy criteria like All-One-Loop-Paths for state machines with many parallel regions can exceed the available memory. On the one hand, this might be a problem of ParTeG's memory management. However, for future versions of ParTeG we are planning further improvements. On the other hand, this seems to be a problem related to the selected coverage criterion: we are currently investigating the impact of the coverage criterion on the size and fault detection capabilities of the test suite. Since there is no proof that coverage criteria have an impact on the number of identified faults, the effect of coverage criteria needs to be examined.

7 Related Work

References to model-based testing and partition testing can be found in [1, 2, 24]. Hierons et al. [9] use conditioned slicing to check given input partitions. Dai et al. [5] use partition testing and rely on the user to provide input partitions. Our approach differs in that we create input partitions instead of relying on predefined ones. Legeard, Peureux and Utting [13] develop a method for automated boundary testing from the textual languages Z and B based on set-oriented constraint technology. They execute all operations with all input boundary values on each reachable boundary state. In contrast, our approach uses the languages UML and OCL. It is based on transformations instead of constraint solving.

OCL is object of many studies [17, 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. [8] 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, 4]) also support automatic test generation but are not designed for object-oriented systems. Offutt and Abdurazik [16] generate test cases from state machines. However, they focus on single transitions and random source state initialization paths. Sokenou [20] alters the initialization by using sequence diagrams. Furthermore, she translates OCL constraints of the model into Java code to use them as a test oracle. Our algorithm deviates in that we also evaluate OCL constraints and use them to derive test input value partitions.

To derive test cases, we create an intermediate control-flow tree: the test case tree. In [11], Kansomkeat and Rivepiboon introduce a Test Flow Graph generated from a UML statechart diagram. Their generated test suites satisfy state coverage and transition coverage. In contrast, our tree also contains conditions from class diagrams; the nodes contain input value boundaries. This allows to generate test suites that also satisfy boundary-based coverage criteria.

Many commercial tools support testing. The Conformiq Test Generator [25] supports parallelism and concurrency in UML state machines but input values are created manually. The algorithm of the tool AETG [21] also depends on user-defined values and boundaries. In contrast, we derive input partitions automatically. The tool Rhapsody ATG [22] is based on UML state machines. It generates and executes test cases with respect to coverage criteria like MC/DC. The tool LTG/UML [24] from Leirios [14] evaluates OCL expressions to generate test cases. It interprets equations in postconditions as assignments, which allows to perform a symbolic execution of the model. In contrast to that, our approach is not restricted to active equations but can also evaluate active inequations in OCL postconditions. In future work, we also aim at evaluating more complex active OCL expressions in OCL postconditions. To our knowledge, no commercial tool creates test cases by explicitely deriving input partitions from conditions.

8 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 will evaluate a broader range of constraints in OCL postconditions, and we will use our method in the domain of Geo-information systems. We will also examine the satisfied coverage criteria of the generated test suite and, if necessary, define new ones.

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

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., 1999.
3. 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.
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. 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.
6. Object Management Group. Object Constraint Language (OCL), version 2.0, 2005.
7. Object Management Group. Unified Modeling Language (UML), version 2.1, 2007.
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. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. In *Software Testing, Verification and Reliability*, 2002.
10. Reactive Systems Inc. Reactis. http://www.reactive-systems.com.
11. S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *SAICSIT '03*, pages 296–300, 2003.
12. 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.
13. B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *FME*, pages 21–40, 2002.
14. Leirios. LTG/UML. http://www.leirios.com.
15. B. Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN*, pages 17–34, 1988.
16. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML'99*, pages 416–429, 1999.
17. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *ER*, pages 449–464, 1998.
18. S. Weißleder. ParTeG (Partition Test Generator). http://parteg.sourceforge.net.
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. Y. Le Traon. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, 2006.
24. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., 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. In *FMICS'2003*, 2003.