Cause-Effect Graphs for Test Models Based on UML and OCL

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

Dehla Sokenou
GEBIT Solutions
Koenigsallee 75b, 14193 Berlin, Germany
dehla.sokenou@gebit.de

Abstract

In this paper, we discuss how to transform UML state machines with OCL expressions into cause-effect graphs. This transformation is necessary to keep test models consistent and understandable. We substantiate all explanations by an example model, which is part of a model for a freight elevator control.

Keywords: *automatic test generation, model-based test specification, cause-effect graph, UML, OCL*

1 Introduction

The primary goal of testing is to detect as many faults of a system under test (SUT) as possible. For that, the behavior of the SUT is compared to a behavior specification. In model-based testing, such specifications are models (e.g. instances of the UML [7]). For automatic test generation, models can be used for the generation of test oracles, test sequences in order to satisfy a specified coverage criterion, and concrete test input data.

The quality of test suites is usually measured with coverage criteria. They can be applied at the model-layer or at the layer of the SUT. Both measured values can differ considerably. The completeness of the model influences the relation between the measured coverage at both layers. Since the model is created manually, the clearness of the model is important for its consistency. The clearness of models significantly depends on the used modeling language. The use of several different modeling languages usually decreases the model's clearness.

A cause-effect graph [2, 4] is "a graphical representation of inputs or stimuli (causes) with their associated outputs (effects), which can be used to design test cases" (BS 7925-1.British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST)). Furthermore, cause-effect graphs contain directed arcs that represent logical relationships between causes and effects. Each arc can be influenced by boolean operators. Such graphs can be used to design test cases, which can directly be derived from the graph [4], or to visualize and measure the completeness and the clearness of a test model for the tester.

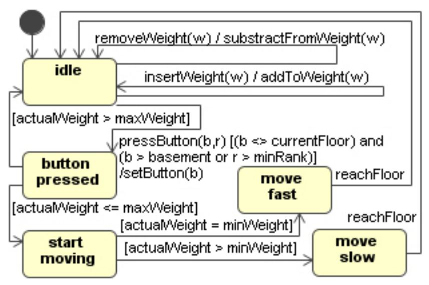
Two widely-adopted modeling languages are the Unified Modeling Language (UML [7]) and the Object Constraint Language (OCL [6]). UML state machines with OCL expressions are often used as test models, which describe all execution paths of the SUT [3, 11]. This combination of graphical and textual notation causes a high model complexity because, e.g., OCL expressions and the state machine both influence the behavior. Consequently, there is a higher probability for an erroneous test model. An explicit description of the interdependencies between OCL and UML can reduce this probability. Cause-effect graphs are one way to describe these interdependencies. After introducing the running example and cause-effect graphs, we will describe a way to transform combined UML state machines with OCL expressions into cause-effect graphs.

2 Example

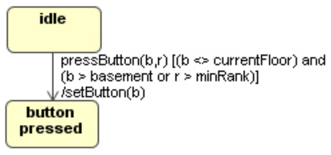
In this section, we introduce an example of a freight elevator. Figure 1(a) shows an UML state machine with OCL expressions that describes a part of a corresponding elevator model. As presented in [11], one algorithm to generate test cases from such models comprises the following steps: classify OCL expression in order to find changeable variables; select guard conditions that must be satisfied in order to satisfy certain coverage criteria; and find preceding postconditions for each guard in the state machine that influence the satisfaction of this guard condition to build a path for a test case.

This algorithm enables the tester to derive test cases that satisfy all guard conditions of the state machine. However, the quality of testing is measured with coverage criteria [10], some of which are only satisfied if a guard condition is violated (e.g. Decision Coverage [1]). As presented in [12], one way to solve this is to add corresponding OCL expressions to the test model that describe the violated guard conditions according to certain coverage criteria. For instance, Figure 1(b) shows one transition of Figure 1(a), which contains a guard condition that references parameters of the triggering event. To satisfy Decision Coverage, at least one of the additional expressions shown in Figure 1(c) also has to be satisfied.

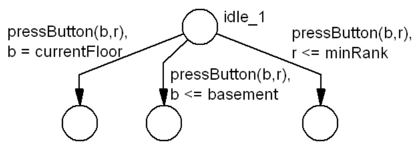
There are some tools that generate executable test cases from UML state machines and OCL expressions [3, 11]. However, the used test models still need


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]
reachFloor
[actualWeight = minWeight]
start moving
[actualWeight > minWeight]
move slow

(a) Complete state machine.

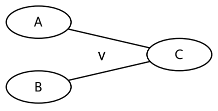

idle
pressButton(b,r) [(b <> currentFloor) and (b > basement or r > minRank)] /setButton(b)
button pressed

(b) Guard conditions.

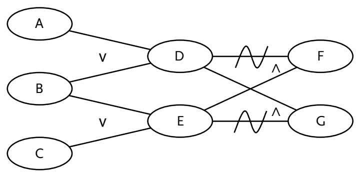

idle_1
pressButton(b,r), b = currentFloor
pressButton(b,r), r <= minRank
pressButton(b,r), b <= basement

(c) Additional expressions.

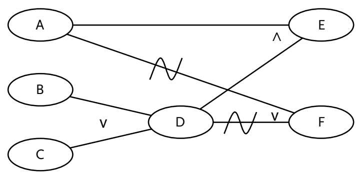
Figure 1: Model of a freight elevator control.



(a) Simple disjunction.



(b) Concatenation.



(c) Expressing Guard Condition.

Figure 2: Cause-effect graphs.

to be validated. There is no reference specification for that model to compare with and the tester himself has to validate the test model. For that task, the tester has to understand the test model. A test model can be understood if its presentation is plausible. As stated above, the use of several modeling languages in one model makes the model hard to understand.

In our approach, the used modeling languages are UML and OCL. For making the corresponding models understandable, they should be transformed into instances of a different modeling language that provides a better understanding. In this paper, we propose cause-effect graphs to express the causes and the corresponding effects of UML models with OCL expressions and to ease the calculation of preceding preconditions.

3 Cause-Effect Graphs

Cause-effect graphs are directed graphs with causes and effects represented as nodes and connections between them represented as arcs. Each node is labeled with a unique number or letter referencing a certain condition. Arcs can be negated and connected to other arcs with boolean operators. Cause-effect graphs are often used because they are easy to understand and intuitive to use.

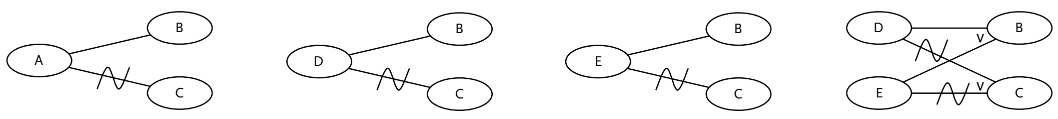
Figure 2(a) shows a cause-effect graph with two causes A and B and one effect C. The arcs in this graph mean that A or B has to be satisfied in order to satisfy C. The vee means a disjunction of both arcs. There are more possible connectors derived from boolean expressions like *AND*, *NOT*, or *REQUIRES*. For instance, Figure 2(b) shows several connected causes and effects: the nodes labeled D and E are effects of the nodes A, B, and C. D and E are also causes for the nodes F and G. The wedge means a conjunction and the tilde means a negated connection: e.g., the node G is satisfied if D is satisfied and E is not satisfied.

In this paper, we apply cause-effect graphs to depict UML state machines with OCL expressions. We demonstrate it with our example of a freight elevator. For instance, Figure 2(c) shows such a graph for the guard condition attached to the transition shown in Figure 1(b): The node A stands for $b <> currentFloor$, B stands for $b > basement$, and C for $r > minRank$. The node D is an intermediate node that is an effect of B and C but that is also a cause for E and F. The nodes E and F represent the positive respective negative result of the evaluation of the guard condition.

4 Transform UML-OCL-Models to Cause-Effect Graphs

In this section, we show how to transform UML state machines with OCL expressions to cause-effect graphs. For instance, Figure 2(c) shows a cause-effect graph for a single guard condition. Our approach, however, is not restricted to transforming single OCL expressions to cause-effect graphs. Instead, we want to transform several OCL expressions that influence each other. For that, we need a way to connect OCL expressions contained in the state machine.

One solution is the classification of variables contained in OCL expressions [11]. This classification identifies *active* and *passive* variables: the value of active variables can change in OCL expressions; passive variables, however, cannot change their value. The use of this classification allows to relate different OCL expressions of the state machine: for instance, a post-



(a) Guard expression. (b) Transformed expression for insertWeight. (c) Transformed expression for removeWeight. (d) Combined transformed expressions.

Figure 3: Cause-effect graphs for OCL expressions.

```
createCauseEffectGraphs(model) {
  Graphs = Set();
  classify all OCL expressions contained in model;
  foreach guard g from model {
    add a cause-effect graph for all input parameters in g to Graphs;
    foreach active variable v in g {
      find a preceding postcondition p that influences the value of v;
      transform g with p to a new expression e;
      repeat evaluation of e for remaining active variables;
      add a cause-effect graph for e to Graphs;
  }}
  return Graphs;
}
```

Figure 4: Pseudo code definition for the creation of cause-effect graphs.

```
combineCauseEffectGraphs(allGraphs) {
  foreach pair of graphs g1, g2 from allGraphs {
    if(the set of effects in g1 and g2 are overlapping){
      combine both graphs by adding all causes and elements to one graph and unite the equal ones;
  }}
  return allGraphs;
}
```

Figure 5: Pseudo code definition for the combination of matching cause-effect graphs.

condition contains an active variable referencing the same system attribute a as a passive variable in a succeeding guard condition; by influencing the value of a, this postcondition also influences the satisfaction of the succeeding guard condition. In our example, the postcondition of the effect *addToWeight(w)* influences the guard condition $actualWeight > minWeight$ (cf. Figure 1(a)). Consequently, the value of the input parameter w is a cause for the satisfaction of the guard condition, which is the effect.

4.1 Combining Cause-Effect Graphs

In the following, we show how to combine cause-effect graphs that describe somehow related OCL expressions In our running example, we consider two expressions: the guard condition $actualWeight > minWeight$ and the postcondition of *addToWeight(w)*, which is $actualWeight = actualWeight@pre + w$. All variables of the guard condition and all variables but the first of the postcondition are passive. The left hand use of $actualWeight$ in the postcondition is active. According to the transformation presented in [11], we can connect both expressions to the new expression $actualWeight@pre + w > minWeight$. This expression has to be satisfied in the postcondition in order to satisfy the following guard condition. How do the corresponding cause-effect graphs look like? Figure 3(a) shows the cause-effect graph for the guard condition: the node A represents the guard condition $actualWeight > minWeight$, the nodes B and C in Figures 3(a)-3(d) stand for the positive respectively negative evaluation of this guard condition. Figure 3(b) shows a cause-effect graph for the transformed expression: node D stands for the expression $actualWeight@pre+w > minWeight$ when the event *insertWeight(w)* occurs. A similar OCL expression can be created for the postcondition of the effect *substractFromWeight(w)*. For this case, the transformed expression is: $actualWeight@pre - w > minWeight$. The corresponding cause-effect graph is shown in Figure 3(c). Both presented transformed expressions represent possible causes for the satisfaction of the considered guard condition. Consequently, we can combine both causes in one cause-effect graph: Figure 3(d) shows the combined graph (all letters mean the same as in the previous figures). The pseudo codes in Figure 4 and Figure 5 sketch the creation of cause-effect graphs from UML state machines with OCL expressions and the subsequent combination of existing cause-effect graphs.

5 Conclusion and Discussion

In this section, we conclude and discuss the presented approach. In the last sections, we dealt with the transformation of UML state machines with OCL expressions into cause-effect graphs. This approach is motivated by the following facts: UML state machines with OCL expressions are often used as test models. There are solutions to automatically create test cases from such models [3, 11]. However, the mix of two modeling languages makes it hard for the tester to understand the test model. Therefore, we need a way to present the implications of the current model. Cause-effect graphs are often used to model test cases because they just confront causes and effects and are easy to understand. Consequently, we used cause-effect graphs to display the test models.

According to this, the presented approach is reasonable. However, there are open issues left to discuss. For instance, the completeness of the transformation of OCL expressions depends on the power of the logical comparison of OCL expressions. We presented a way to compare OCL expressions depending on a classification of active and passive variables. The inclusion of operations on sets seems to be manageable but needs further investigation. The used state machines are flat state machines. We believe that constructs like history or substates bear no general limitation to our approach but this also needs further investigation. We omitted details of the presented process. For instance, first, all expressions are transformed into disjunctive normal form. Since this form contains just disjunctions and conjunctions, these transformed expressions are easier to compare than the original ones. We believe that there is more potential for improvements.

6 Related Work

Cause-effect graphs were developed by Elmendorf [2]. Test case derivation from cause-effect graphs is described in [4] and [8]: The algorithm in [4] guaranteed condition coverage. The approach presented in [8] combines cause-effect graphs with a test strategy called boolean operator (BOR) which derives test cases that satisfy decision-based coverage criteria like MC/DC. By contrast, we propose the use of cause-effect graphs to make the UML-OCL-models easier to understand. This approach is not focused on test case generation from cause-effect graphs: The test cases are derived from the UML-OCL-model. The corresponding generation process supports several coverage criteria and is not limited to decision-based coverage criteria.

An informal specification of the combination of state machines and OCL is already found in [7]. Using this semantics, UML state machines and OCL are combined in [9]. Here, the focus lies on test oracles not on test case generation. In [9] and [5], initializing transitions are calculated that need to be performed in order to reach states that represent different test scenarios. Both approaches do not consider OCL conditions for automatic calculation of the start conditions. The Leirios Test Generator [3] also uses UML state machines and OCL expressions to derive test cases. However, the corresponding algorithm only supports a subset of OCL (cf. [11]).

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

References

[1] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage Criteria for Logical Expressions. In *ISSRE '03*, Washington, DC, USA, 2003. IEEE Computer Society.

[2] W. R. Elmendorf. Cause-Effect Graphs in Functional Testing. Technical Report TR-00.2487, IBM Systems Development Division, Poughkeepsie, NY, 1973.

[3] Leirios. LTG/UML. http://www.leirios.com.

[4] Glenford J. Myers. *The Art of Software Testing*. 1979.

[5] Roman Nagy. Bedeutung von Ausgangszuständen beim Testen von objektorientierter Software. In *CoMaTech '04*, Trnava, Slowakei, 2004.

[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] Amit Paradkar, K. C. Tai, and M. A. Vouk. Specification-Based Testing Using Cause-Effect Graphs. *Annals of Software Engineering*, 4:133–157, 1997.

[9] D. Sokenou. *UML-basierter Klassen- und Integrationstest objektorientierter Programme*. PhD thesis, Technische Universität Berlin, Germany, 2006.

[10] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[11] S. Weißleder and B.-H. Schlingloff. Deriving Input Partitions from UML Models for Automatic Test Generation. In *LNCS Volume on Models in Software Engineering (MoDELS 2007)*, 2007.

[12] S. Weißleder and B.-H. Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *ICST*, April 2008.