Reusing State Machines for Automatic Test Generation in Product Lines

Stephan Weißleder[1], Dehla Sokenou[2], Bernd-Holger Schlingloff[3]

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

[2]GEBIT Solutions
Koenigsallee 75 b, 14193 Berlin, Germany
dehla.sokenou@gebit.de

[3]Fraunhofer Institut FIRST
Kekuléstraße 7, 12489 Berlin, Germany
holger.schlingloff@first.fraunhofer.de

Abstract

In this paper, we deal with the reuse of state machines for automatic test case generation in the context of product lines. We consider a corresponding approach of the Object Management Group and introduce our own approach to reuse state machines. We use OCL expressions to automatically derive test suites. All explanations are supported by the example of a car audio system.

1 Introduction

The primary goal of testing is to find as many errors in the system under test (SUT) as possible with minimal effort. The quality of test suites is assessed by coverage criteria. They are used to keep development and maintenance costs for testing at a reasonable rate to the required safety level of the SUT. Since testing consumes a large amount of the overall development costs, much effort is put into keeping costs for testing as low as possible. The reuse of certain artefacts, e.g., test suites is of use to reduce development costs and maintenance costs. In the field of model-based testing, the reuse of test models also significantly contributes to reducing such costs. In this paper, we aim at reusing test models. Assuming an automatic test generation process, a whole test suite can be generated from a test model. This could lead to reduced costs of model-based testing.

A product line is a set of products with common, optional, and alternative features. Feature models are used to relate the basic product and features. By reusing common features, product lines provide an efficient way to reduce costs for various similar products. Therefore, testing techniques for product lines should strive for the same goal.

In this paper, we present two ways to reuse state machines of the Unified Modeling Language (UML) for test case generation: the one proposed by the Object Management Group (OMG) and our own approach. We sketch advantages and disadvantages of both

approaches in the context of product line testing. Furthermore, we present a prototype implementation for our approach of state machine reuse. The main contribution of this paper is a technique to reuse test models in the context of product lines.

The paper is structured as follows. In the following section, we present related work. In Section 3, we introduce our running example of a car audio system and a corresponding product line. Afterwards, we discuss both approaches to reuse state machines. In Section 5, we sketch the test generation process. We conclude this paper with a discussion in Section 6 and a conclusion and an outlook to future work in Section 7.

2 Related Work

Model-based testing is extensively discussed in [3, 25]. Instances of the UML are often used as test models and as sources for automatic test case generation [4, 23]. For instance, UML state machines are used by Offutt and Abdurazik to derive test cases automatically [18]. In our approach, test cases are generated automatically from UML models with special focus on inheritance as one part of object-orientation. El-Fakih et al. [5] consider testing during product life cycle. They reduce the test effort by testing only the modified elements of a new product version. In contrast, our approach allows to test all products of a product line based on one behavioural test model. The automation of the test case creation reduces the effort for repeated test creation. However, a combination of our approach with the approach of El-Fakih et al. seems to be reasonable.

Liskov [16] defines the substitution principle between types and subtypes. We apply this approach by interpreting a state machine as a property of its context class. This state machine can also be used to describe the behaviour of this class' subclasses.

The Object Constraint Language (OCL) is used to constrain objects and models [22, 30]. Hamie et al. consider OCL in combination with state machines and classes [9]. We apply our approach of transforming OCL expressions to generate test input partitions [28].

Partition testing and boundary testing are testing techniques that are focused on the selection of test input values. Several approaches validate predefined boundaries [10, 13] but do not provide means to derive these boundaries from test models. A prominent approach is the classification tree method (CTM), which enables the tester to define arbitrary partitions. Alekseev et al. [1] propose the reuse of classification tree models. Gomaa [6] introduces the PLUS models and method. He uses Boolean feature conditions to activate product line features. In contrast, our algorithm is not restricted to Boolean values or complete activation of features, but is also able to adapt feature details.

Olimpiew and Gomaa [19] deal with test generation from product lines and sequence diagrams. In contrast to that, we focus on UML state machines, OCL constraints, and inheritance relationships. McGregor [17] points out the importance of a well-defined process for testing software product lines. Kolb [12] discusses the problem of selecting a suitable software product line testing strategy that takes the reuse of variable elements into account. As an extension to that, we focus on the reuse of models for automatic test generation to reduce test development costs. Pohl and Metzger [20] discuss the advantages of software product line testing and emphasise the preservation of variability in test artefacts. As we generate test cases from reused models automatically, this variability is preserved. Gomaa [7] also dealt with an architecture-centric evolution of product lines in which he supports a model-driven approach to develop software product lines.

There are many commercial modelling tools that offer test support. Rhapsody ATG [24] generates and executes test cases with respect to UML state machines. We develop the tool Partition Test Generator (ParTeG) [27], which supports the automatic generation of boundary tests. The tool LTG/UML from Leirios [15] generates test cases from OCL expressions but is not focused on the reuse of test models.

3 Example: Product Lines for Car Audio Systems

A product line consists of products, e.g., software or hardware with essential similar features. The dissimilar features are known as product variation points. Activating or deactivating different variation points results in different product variants. For developing and testing product lines, it is beneficial to deal with their similar features prior to their dissimilar features. The application of product lines to software products results in software product lines (SPLs), which are sets of similar software products. SPLs are often considered in the context of model-based testing [17, 19].

Product lines are typically modelled with feature models. Each feature model contains the features of a product line, their mutual relations, and their relations to the basic product (cf. Figure 1). These relations describe the features as common, optional, or alternative. Other relations between features describe, e.g., dependencies between single features or dependencies inside of a group of features.

In the following, we present our running example: a car audio system. A car audio system has several possible features, many of which are common to all product variants. Therefore, the car audio system is appropriate for a description in a feature model. Figure 1 shows a feature model with some reasonable features of a car audio system.

For instance, the basic controls are common to all radios: A radio must provide an option to switch its mode, e.g., to toggle radio and playback media. It must allow the selection of channels or titles, the change of the volume, and the search for new channels or new titles. Depending on the current mode and received events (e.g., emergency messages via the traffic message channel), the basic controls like forward and backward search can have different meanings. Furthermore, the choice of playback media (CD or cassette player) also influences the behaviour. Whereas a forward event for a cassette player results in winding the tape, a forward event for a CD player results in the immediate

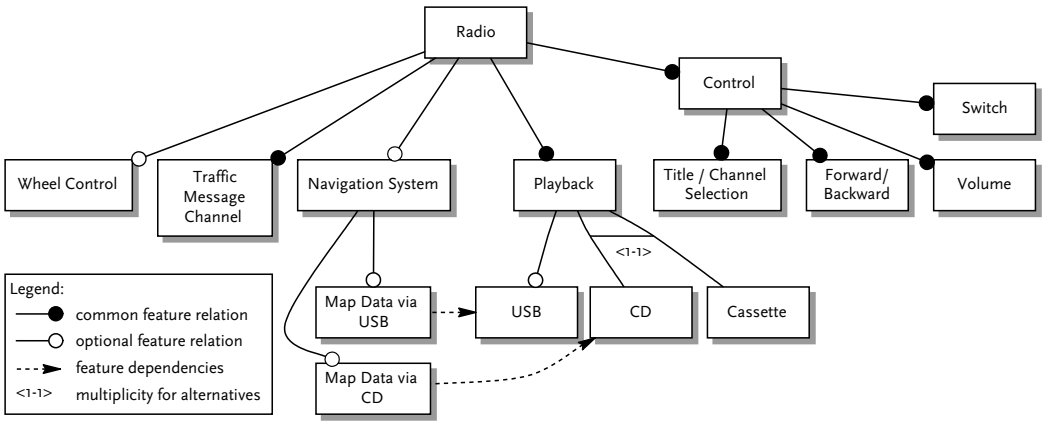

Figure 1: Feature model of car audio systems.

selection of the next track. The traffic message channel is common to all car audio systems. The availability of a USB port, the wheel control, and the navigation system are optional features. Depending on the playback media, the update of map data for the navigation system is an optional feature.

4 Using State Machines to Model Product Variants

State machines [8] are often used to model the behaviour of product variants. In our context, they are used as test models to automatically generate test suites. There are many ways to reuse these state machines as test models in order to reduce the costs for test development and test maintenance.

The OMG proposes to generalize and specialize state machines for UML. Rationale of this is to enable the redefinition of a general classifier's behaviour. So, the OMG defines a generalization relationship between UML state machines [8]: *"A specialized state machine is an extension of the general state machine, in that regions, vertices, and transitions may be added; regions and states may be redefined (extended: simple states to composite states and composite states by adding states and transitions); and transitions can be redefined."* Furthermore, the specification states: *"A submachine state may be redefined. The submachine state machine may be replaced by another submachine state machine, provided that it has the same entry/exit points as the redefined submachine state machine, but it may add entry/exit points. Transitions can have their content and target state replaced, while the source state and trigger are preserved."*

In principle, the specialized state machine is read similarly to "non-specialized" state machines. Interesting features can be added when it is necessary and, consequently, agile development is supported. Nonetheless, some important problems are unsolved. For instance, transitions can be redefined but there is no way to mark them as reused. Consequently, all transitions have to be re-drawn in the specialized state machine. Furthermore, the relations between the general and the specialized state machine are not clarified. Submachine state machines, transitions, and regions can be replaced, added, or redefined. Such changes can have significant effects on the whole state machine. The effects of changes in the general state machine on the specialized state machines are also undefined. To our knowledge, there is no tool support for this approach.

Our approach to reuse state machines leaves the state machine unchanged but changes its context class. As specified in [8], the context of a state machine is a class; transitions of the state machine can refer to properties and operations of this class via events, guards, and effects. Instead of using an inheritance relationship between state machines, we use the inheritance relationship between classes and reuse a state machine as a behavioural description of these classes. For that, we define one state machine for a general class. Since all specialized classes contain the same operations, attributes, and associations, this state machine can also describe the behaviour a specialized class. Another motivation for this approach is Liskov's substitution principle [16]. This principle states that all properties of a class also have to hold for its subclasses. Obviously, the behaviour of a class is such a property. Since a state machine is the model of a class' behaviour, it is also a model of a class' property and can, thus, be a model for the behaviour of each subclass. Consequently, each subclass of the state machine's originally defined context class is a possible context class (see Figure 2).

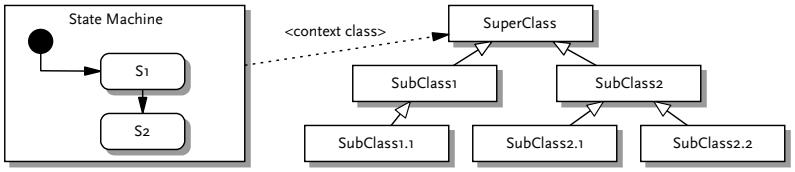

Figure 2: Each subclass of class *SuperClass* can be the context of the state machine.

In our approach, the default values of the referred properties and the pre-/postconditions of the operations in the selected context class influence the behaviour of the state machine. Consequently, the behaviour described by the state machine depends on the selected context class and the modelled behaviour changes with the selection of a new context class. By using submachine states, the clarity of the test model is kept even for large systems. Figure 3 shows a reusable state machine for our example - Figure 4 shows the corresponding context classes. Note that our approach is not only feasible for testing one class, but also for systems with many classes. For each product variant, each active class is then described by a state machine.

This state machine describes a part of the car audio system's complete behaviour. Each car audio system is in one of two states: *On* or *Off*. The state *On* is a composite state and contains two regions. The upper region deals with the common feature *Switch* between sources of the current playback: Users can *switch* between several sources. A *TMCEvent* is triggered if a traffic-relevant message is received. If the traffic message control is activated ($bTMCEnabled = true$), then the input media will change to the traffic channel. If the message broadcast is finished or the radio is turned off and on again, then the radio is reset to the state *SourceSelection*. The lower region describes some of the remaining common control features, like volume control or track control. Both regions are handled in parallel. Depending on the state of the car audio system, the effects of the events differ. For instance, if the radio receives a traffic message, the volume is set to a higher value and the track control is disabled. Furthermore, the handling of the features *Forward* and *Backward* depends on the current product variant - e.g., immediate track selection for a CD or track search for a radio.

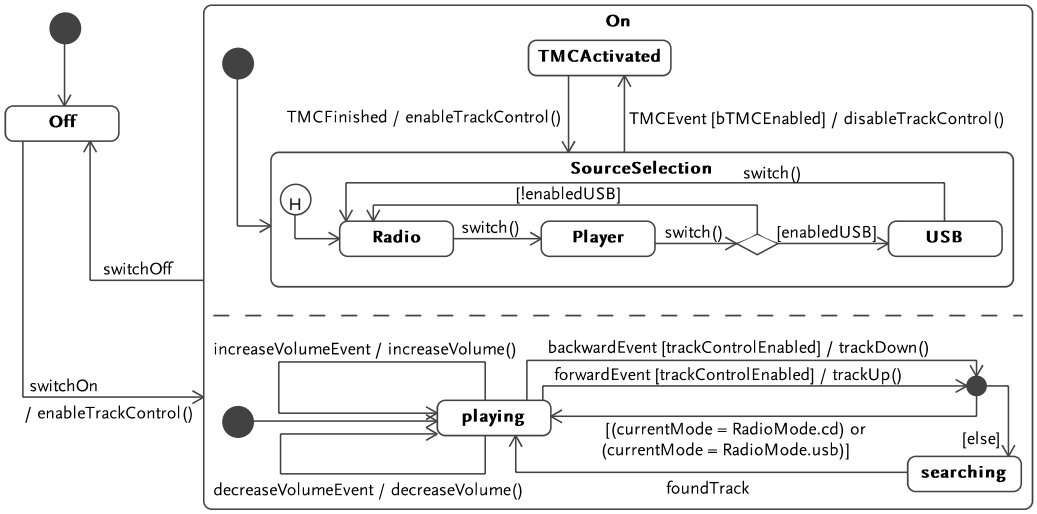

Figure 3: A state machine describing an extract of the general car audio system behaviour.

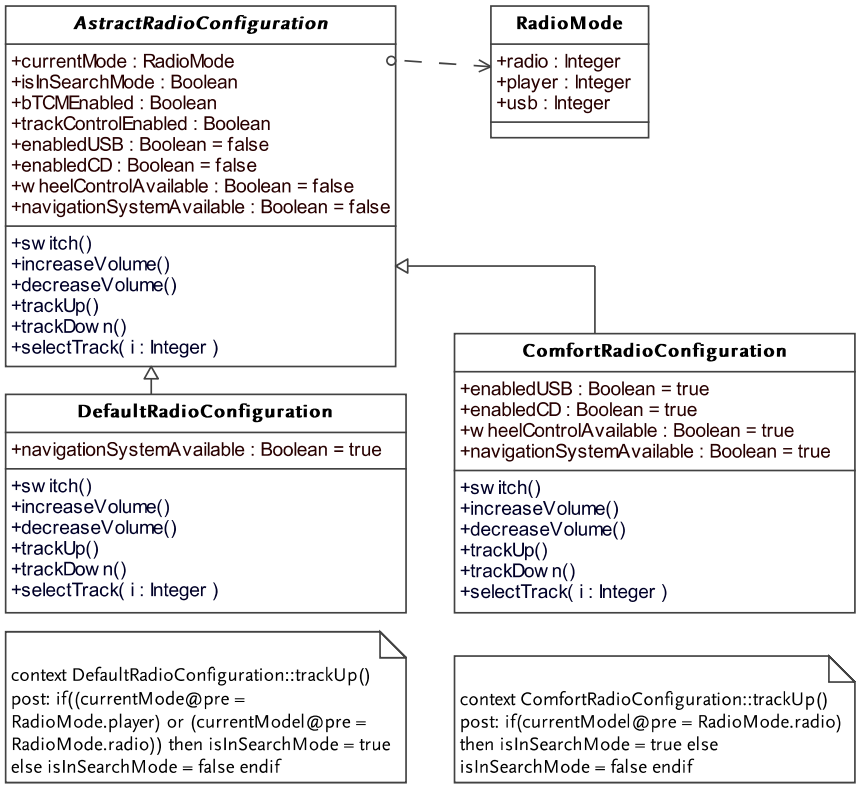

Figure 4: Extract of classes describing two product configurations for a car audio system.

5 Automatic Test Generation

In this section, we examine our automatic test generation process for product lines. In order to derive tests for a product variant, a behavioural description of this variant is required. For instance, Kishi and Noda [11] model one state machine for each feature-supporting component. According to our own approach, we describe the behaviour of all product variants with one state machine. This state machine describes the behaviour of each product variant of the product line. We adapt Liskov's substitution principle [16]. The corresponding configuration classes have to be derived from the feature model. One way to do so is to identify all possible combinations of product features and create corresponding configuration classes. There are certainly more ways to derive class hierarchies from product lines. We discuss some of them in Section 6.

5.1 Test Case Generation Process

The behavioural description of each product variant is given as a state machine. This state machine consists of regions with states and transitions that connect states. Furthermore, the behaviour depends on guard conditions and the effects of transitions. The test generation process evaluates all conditions along the transitions of a state machine [28]. These conditions include, e.g., guard conditions of transitions and pre-/postconditions of the transitions' effects. The elements of these conditions are categorised and subdivided into *dependent* and *independent* variables - according to their dependency on former behaviour (cp. [28]). By this, elements of guards and postconditions can be related to

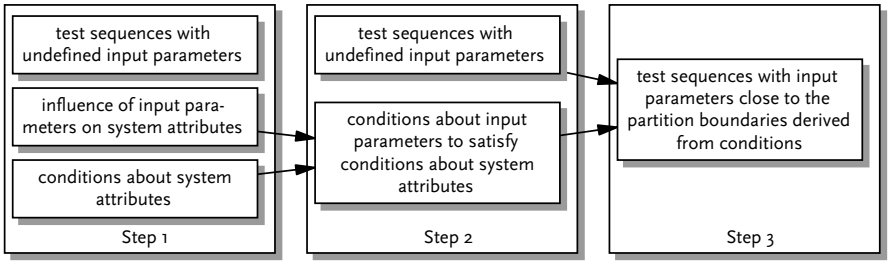

Figure 5: Basic idea of the test generation process.

each other. Using such relations, the effects of a postcondition on the satisfaction of a guard can be detected. The corresponding algorithm is able to identify relations between elements that are contained in inequations, logical disjunctions, and even negations (also in postconditions). This exceeds the ability of other static analysis approaches to generate test suites from OCL expressions. These are often restricted to the evaluation of equations as expressions to change the value of a variable [15]. The main goal of this strategy is to transform conditions concerning system attributes into conditions about input parameters or constants. This results in test suites with concrete input values close to the partition boundaries derived from transformed conditions (see Figure 5). Therefore, the generated test suites satisfy boundary-based coverage criteria [14].

The sketched test generation approach works for one state machine and one context class. Following Liskov, this approach also works for a pair of one state machine and several classes from a class hierarchy (cf. Figure 2). These classes can be derived from product lines. So, our approach is adequate to generate test suites for product lines.

5.2 Generating Test Suites for a Car Audio System

The feature model in Figure 1 contains 14 common, optional, and alternative features. Common features are activated in each product variant. Optional and alternative features reflect differences between product variants. The corresponding configuration classes for our approach result from the combinations of activated optional and alternative features. In the car audio system example, this leads to 26 product variants overall. Probably, not all of these configurations are to be provided to customers. Consequently, fewer configurations might be sufficient. Below, we focus on automatic test generation for the two configurations shown in Figure 4.

ParTeG generates JUnit test suites for both configurations. Thus, the test suites are executable without any further effort. ParTeG allows the user to choose a coverage criterion, which should be satisfied as good as possible by the generated test suite. The test generation process is adapted to the selected coverage criterion. The current version provides three well-known coverage criteria to choose from: *State Coverage*, *Decision Coverage*, and *Modified Condition / Decision Coverage*.

6 Discussion

The presented approach supports the reuse of test models. Nevertheless, there are points left to discuss. For instance, the relative completeness of pre-/postconditions significantly

influences the generated test input partitions based on OCL expressions. Due to the success of random testing, the effectiveness of model-based testing needs discussions [21]. Additionally, the combination of several coverage criteria seems to be promising [29].

For our approach, there are several reasonable adaptations and extensions. For instance, initial configurations can be expressed in object diagrams. Since names of test model elements may differ from the used names of SUT elements, it may be necessary to refactor the test models for each SUT or to adapt the resulting test suite in a subsequent step. The use of just one state machine can also be discussed - especially for distributed systems or distributed development processes: it would be possible to define several active classes of the SUT and define one state machine for each active class. Furthermore, the reuse of test models and the automatic test generation process indicate that a reasonable amount of test effort can be saved by using the presented method. Corresponding case studies have to be carried out.

Another important issue is the derivation of class hierarchies from product lines. Is it possible to derive a class hierarchy similar to the hierarchy between features of a feature model? For the example in Figure 1, it is possible to define a basic class with all optional features deactivated. Therefore, two subclasses can be derived by activating the features *Wheel Control* and *Navigation System*, each. The definition of a class that activates boths features seems to be an issue. Since it is undesirable to derive every class directly from the base class, the new class should inherit the classes defined for the features *Wheel Control* and *Navigation System*. This leads to multiple inheritance, which has some disadvantages connected to the overwriting of class operations. Such problems are well known from C++. Another solution might be the use of aspect-orientation [26], where each feature corresponds to a certain aspect.

An important point for a selected coverage criterion is the actual coverage of the generated test suite. How many percent are sufficient? Andrews et al. [2] investigated this question and discovered a particularly impressive gain in fault-detection in the last 10-20 percent. Consequently, the test process also has to search for the test cases that satisfy the criterion completely. Since full coverage seems to be important, the number of actually feasible test cases is interesting. Static analysis on the test model would be helpful to determine the feasible elements of the coverage criterion.

7 Conclusions and Future Work

In this paper, we proposed the reuse of test models for automatic model-based test generation. We defined an approach to reuse state machines for context classes from a class hierarchy and compared it to the approach to specialize state machines proposed by the OMG. In our approach, the differences between the product variants can be expressed by pre-/postconditions or attribute values of context classes. We sketched how to use ParTeG to generate corresponding test suites.

The main contribution of this paper is a new method to reuse state machines along inheritance relationships between classes. Since class hierarchies can be derived from product lines, our approach is particularly useful for product lines. We admit that the modelling effort for a state machine describing all product line features exceeds the effort for a state machine describing just one product variant. However, it requires considerably less effort than modelling and maintaining an individual state machine for each product

variant of a product line, in particular, for large product lines with dozens of features. Furthermore, the context classes of the state machine are tightly related to the feature model, which eases their specification.

In the future, we plan to perform further case studies to substantiate the advantages of our method. We also plan to extend ParTeG so that other models than state machines can be reused for automatic test generation. Another important aspect is the satisfaction of different coverage criteria that are based on control flow and equivalence class boundaries. Finally, we will investigate combinations of existing coverage criteria to be satisfied by test suites generated by ParTeG.

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

References

[1] S. Alekseev, P. Tollkühn, P. Palaga, Z. R. Dai, A. Hoffmann, A. Rennoch, and I. Schieferdecker. Reuse of Classification Tree Models for Complex Software Projects. In *CONQUEST*, 2007.

[2] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on*, 32:608–624, 2006.

[3] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., 1990.

[4] L. C. Briand, Y. Labiche, and J. Cui. Automated support for deriving test requirements from UML statecharts. *Software and Systems Modeling*, V4, 2005.

[5] Khaled El-Fakih, Nina Yevtushenko, and Gregor von Bochmann. FSM-based Re-Testing Methods. In *TestCom '02: Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, pages 373–390, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[6] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. Addison-Wesley, 2004.

[7] H. Gomaa. Architecture-centric evolution in software product lines. In *Workshop on Architecture-Centric Evolution (ACE 2005)*, 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 *UML 1998, Mulhouse, France*, 1999.

[10] Robert Mark Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. In *Software Testing, Verification and Reliability*, 2002.

[11] T. Kishi and N. Noda. Design Testing for Product Line Development based on Test Scenarios. In *SPLiT*, 2004.

[12] R. Kolb. A Risk-Driven Approach for Efficiently Testing Software Product Lines, 2003.

[13] B. Korel. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[14] 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.

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

[16] B. Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN*, 1988.

[17] J. D. McGregor. Testing a software product line. SEI, Carnegie Mellon, 2001.

[18] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *UML'99*, 1999.

[19] E. M. Olimpiew and H. Gomaa. Model-based testing for applications derived from software product lines. *SIGSOFT*, 2005.

[20] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 2006.

[21] 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: Proceedings of the 27th international conference on Software engineering*, pages 392–401, 2005.

[22] M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *ER*, 1998.

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

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

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

[26] M. A. Wehrmeister, E. P. Freitas, C. E. Pereira, and F. R. Wagner. Applying Aspect-Orientation Concepts in the Model-driven Design of Distributed Embedded Real-Time Systems. In *ISORC'07*, may 2007.

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

[28] 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.

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

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