Derivation of Domain Test Scenarios from Activity Diagrams[1]

Andreas Reuys, Sacha Reis, Erik Kamsties, Klaus Pohl

Software Systems Engineering, University of Duisburg-Essen, Germany
{reuys, kamsties, reis, pohl}@sse.uni-essen.de

Abstract

Requirements are often reported as not suitable for testing, because they are, for instance, incomplete. We argue in this paper for early steps in requirements engineering to ensure the testability of requirements in the context of product families. This paper describes the early derivation of test scenarios from use cases represented as activity diagrams. Use cases are often supplemented with activity diagrams if the control structure of the use case includes loops or branches. The use of activity diagrams allows defining a coverage criterion to ensure a particular degree of completeness of the test scenarios. The approach described in this paper is intended for use cases at the domain engineering level. It is discussed how variability in these use cases can be captured in activity diagrams, and, most important, how to address variability while deriving test scenarios so that a particular degree of completeness is reached. For this purpose, we adapt the existing branch coverage criterion to the needs of product families and provide an operational procedure that helps in deriving a set of test scenarios that fulfills our extended coverage criterion. Eventually, the derivation of test scenarios gives an early feedback to the requirements engineer when performed from the tester's perspective. This increases the requirements quality.

1 Introduction

Testing is an important activity to ensure a certain software quality in single product development as well as in product family engineering. Testing in software product family engineering thereby faces the additional problems of handling the domain test artifacts and variability in development documents. Recent testing approaches (e.g. our ScenTED approach [5]) consider the derivation of test cases from use cases. We suggest the early development of *test scenarios* as an intermediate step to the creation of test cases. A test scenario is a particular sequence of steps to be tested, which does not contain any test data (i.e., input or expected outputs). A test scenario can be determined early when a use case is specified and reasoning about test scenarios helps spot gaps in the use cases.

The derivation of test scenarios should be performed in a repeatable and traceable way to increase the traceability between the development artifacts. A structured derivation can be done with model-based approaches and a corresponding coverage criterion. We propose to use activity diagrams as the model to derive test scenarios. Activity diagrams often occur as part of use case documents to represent the use case scenarios (see [4] for a case study). An activity diagram is able to reflect all possible scenarios for one use case. The requirements engineer does not necessarily perform the creation of such a complete activity diagram. Testers may complete that activity diagram for the test case generation.

Coverage criteria were originally suggested for testing code and were recently adapted for testing use cases [11], since use cases often contain alternatives and loops similar to a program. We use in this paper branch coverage (i.e., each branch of the control flow structure behind a use case should be covered by at least one test scenario), because this coverage criterion leads to a reasonable number of test scenarios, even if the control flow structure contains alternatives and loops.

The branch coverage criterion does not work properly in the case of variability. In product family engineering, variability in the flow of events in a use case usually manifests in addi-

[1] This work was partially funded by the CAFÉ project "From Concept to Application in System Family Engineering"; Eureka Σ! 2023 Programme, ITEA Project ip00004 (BMBF, Förderkennzeichen 01 IS 002 C) and the state Nord-Rhein-Westfalia.

tional decisions and branches in the underlying activity diagram as shown in Figure 1 (i.e., variant 1 of the variation point is realized by branch B1 and variant 2 by B2 – we will introduce the notation later in detail). The branches that reflect variants of a variation point cannot be treated using branch coverage, because this would result in incomplete application testing (i.e., not all branches would be tested for each application). In the example, two test scenarios would result from branch coverage, i.e. {A, B1, C, D1, E} and {A, B2, C, D2, D3, E}. If only variant 1 is chosen for a customer-specific product, then the second scenario is not applicable. This leads to the fact that the common activities D2 and D3 are not tested, because they are not part of the first scenario. The path coverage would work in this case and lead to four scenarios (additional {A, B1, C, D2, D3, E} and {A, B2, C, D1, E}). These four scenarios are not always required and are therefore overhead: If only variant B2 has been taken, than two of four scenarios (50%!) have been created and possibly implemented without a defined use. Therefore the branch criterion is more efficient.

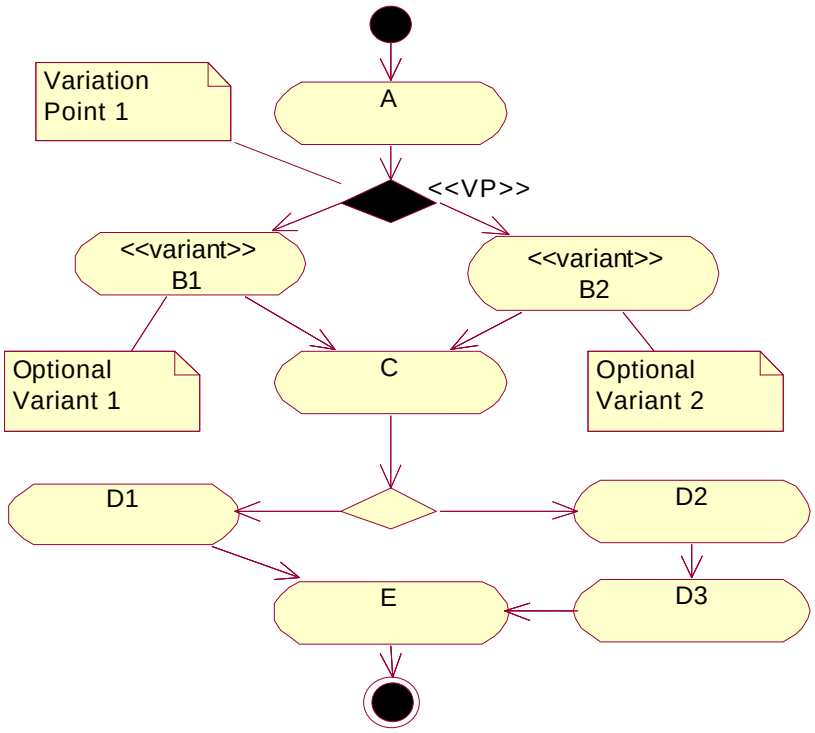

Figure 1: Example activity diagram

This paper aims at defining a branch coverage criterion for software product families that overcomes this problem.

We discuss the related work on activity diagrams and coverage criteria and show that there is no sufficient consideration of variability (see Chapter 2). Afterwards we define how variability can be handled in activity diagrams (see Chapter 3).

Based on this we define a product family specific branch coverage criterion that we explain with several examples and a recommended derivation approach (Chapter 4). The summary and outlook on future work is given in Chapter 5.

2 Related Work

The role of activity diagrams in the UML has been discussed in [9]. Paech shows that activity diagrams are well suited to show user interactions in context of use cases. Activity diagrams have already been considered for use in software product family development. Riebisch et al. describe how to make variants explicit [10]. However, variation points are not made explicit

as required for deriving test scenarios [3]. Moreover, test approaches consider activity diagrams, for example, the approach by Briand [2]. However, these test approaches do not take variability into account.

Coverage criteria are sets of rules, which allow determining whether a program is adequately covered or not. The coverage criteria described in [8] are the most considered ones. Myers distinguish between statement, branch, and path coverage (or condition coverage). Depending on the criterion every statement, branch or path of the program's code must be executed. Kim et. al. propose coverage criteria concerning the use of UML state diagrams [7]. These criteria are adaptations of the Myers' criteria on this type of model. The same holds for Winter who adopts the coverage criteria to use case based development [11]. On a control flow graph that reflects the structure of a textual use case, Winter defines five coverage criteria: use case step coverage, use case branch coverage, use case scenario coverage, use case boundary body coverage, and use case path coverage. However, variability in the control flow graph is not considered.

Summarizing, activity diagrams have proven useful in software and product family engineering. Their value to formalize use cases is well known, but variability in use cases cannot be completely captured in activity diagrams. Coverage criteria have been defined for various UML models. However, these criteria do not take variability into account.

3 Modeling Variability in Activity Diagrams (V-Activity Diagrams)

We propose a simple and pragmatic extension to UML activity diagrams in order to allow capturing variability, which can be used with all standard tools for UML modeling (e.g., Rational Rose). We have defined a new stereotype for decision points that represent the variation point (<<VP>>). The variation point is emphasized using black color. Furthermore, a variation point is annotated with a name.

The variants are the possible choices for a variation point. This is reflected with another stereotype for the variant (<<variant>>). The variants get also a unique identifier in an associated note. We will call these activity diagrams that contain variability *V-activity diagrams*.

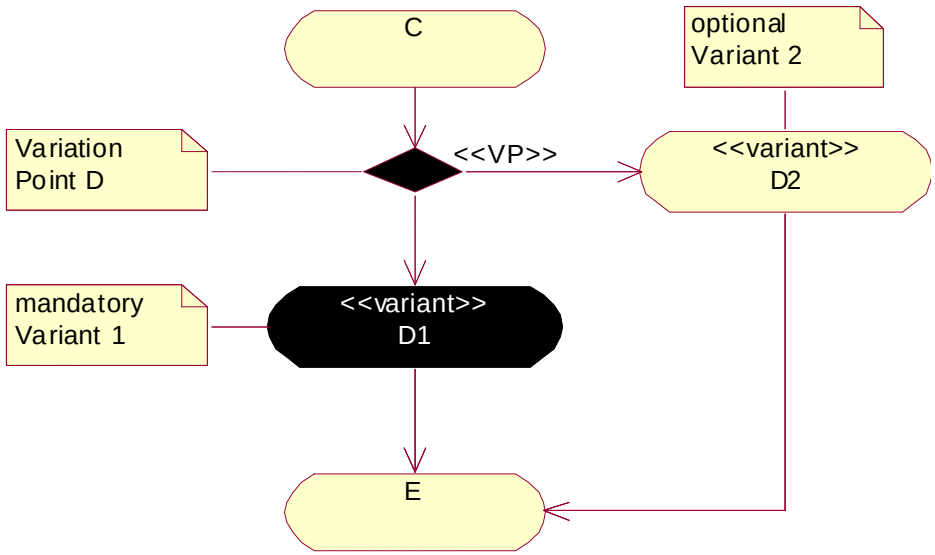

Figure 2: V-activity diagram

Figure 2 shows an example for a V-activity diagram: The variation point D has been made explicit with the stereotype <<VP>> and the corresponding appearance. The ID of this varia-

tion point (Variation Point D) has been written in the annotation. The variation point offers the two variants *D1* and *D2*. The variant identifiers are written in the corresponding note and the stereotypes stress their property of being a variant.

Variability may occur in several types as discussed in [1] and [3], e.g. variability concerning functionality, quality, system environment, etc.. Each type has its own set of attributes whereas some invariant attributes exist (see [3] for a list of type-independent attributes). One of the attributes is called *Existence*: The variants may be mandatory (1 of 1) or optional (0..1 of 1). Our pragmatic extension for the activity diagram is to extend the note for the variant description with the corresponding existence attribute. Moreover, we recommend filling the mandatory variants with the color black (see mandatory variant 1 in Figure 2). This causes simplifications in the test scenario derivation as we will see later in this paper.

Variation points may consist of alternative variants (1 out of *n* variants) and multiple co-existing variants (*l..m* out of *n*) – see [3]. Alternative variants mean that an application may contain only one of the offered variants. Co-existing describes that several variants may exist in parallel. The annotation of the variation point can easily be extended with the corresponding attribute. For co-existing variants the number of variants to be chosen is written in the annotation field. Figure 3 shows both possibilities with two different annotations. For the following examples, we consider the alternative attribute as default. Whenever there is no attribute at the variation point, this means that the variants have to be used co-existing (1..n out of *n*).

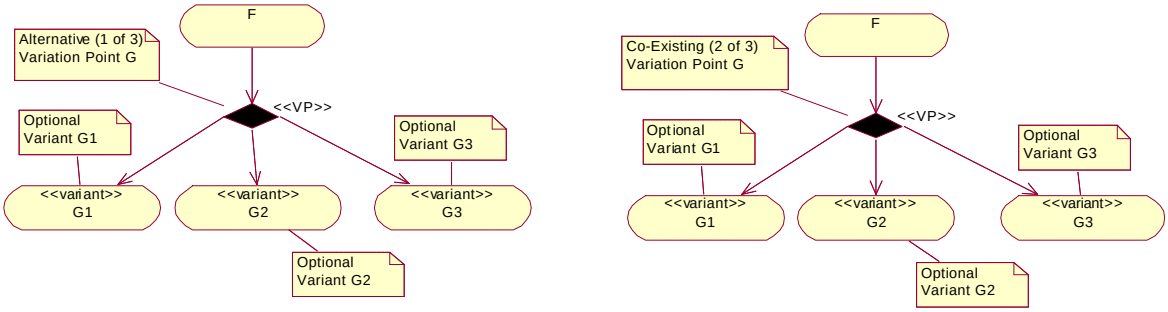

Figure 3: Alternative and co-existing variants

4 Coverage Criteria for V-Activity Diagrams

We aim at providing a criterion that has the same advantages of branch coverage, but overcomes the before-mentioned shortcomings. The problem was that the choosing/binding of variants led to incomplete application test cases. We are strongly convinced that branch coverage should not be weakened for product family testing. Therefore, the straight-forward extension to overcome this drawback is:

> **Each branch of each possible application must be covered by at least one test scenario.**

Obviously, this criterion would lead to a high number of test scenarios. Therefore, also test scenarios must contain variability and we need a procedure for deriving test scenarios that supports this variability. Based on the criterion we suggest the following operational procedure to derive test scenarios from V-activity diagrams:

Step 1: Derive test scenarios so that each branch that does not reflect an optional variant of the V-activity diagram is covered at least once (i.e., conventional branch coverage). Leave an empty variation point in the scenarios for the other kinds of branches.

Please note: “each branch that does not reflect optional variant” imply all branches that have nothing to do with variability as well as all branches that contain mandatory variants!

Step 2: Add to the scenarios the steps that cover the optional variants of a variation point. All branches of all variants must be covered. If necessary, create additional scenarios to reach full coverage of a variant, for example, if a variant contains a decision among alternative branches.

In the following we will give examples how to use this procedure to derive domain test scenarios for several fundamental cases. We call the test scenarios *domain test scenarios*, because the scenarios contain variability. During the test of an application all variants are bound and therefore variability must not appear in the *application test scenario*.

The first case has been shown in Figure 1. Here, variability appears in a bottleneck of the activity diagram. Two test scenarios can be derived using the procedure. Based on the first step one would end up with {A, $\{\}_{vp1}$, C, D1, E} and {A, $\{\}_{vp1}$, C, D2, D3, E}). The second step leads to the domain test scenarios {A, $\{\{B1\}_{v1}$, $\{B2\}_{v2}\}_{vp1}$, C, D1, E} and {A, $\{\{B1\}_{v1}$, $\{B2\}_{v2}\}_{vp1}$, C, D2, D3, E}). The variation point {B1, B2} must be completely covered in each scenario, because it is in the bottleneck of the activity diagram.

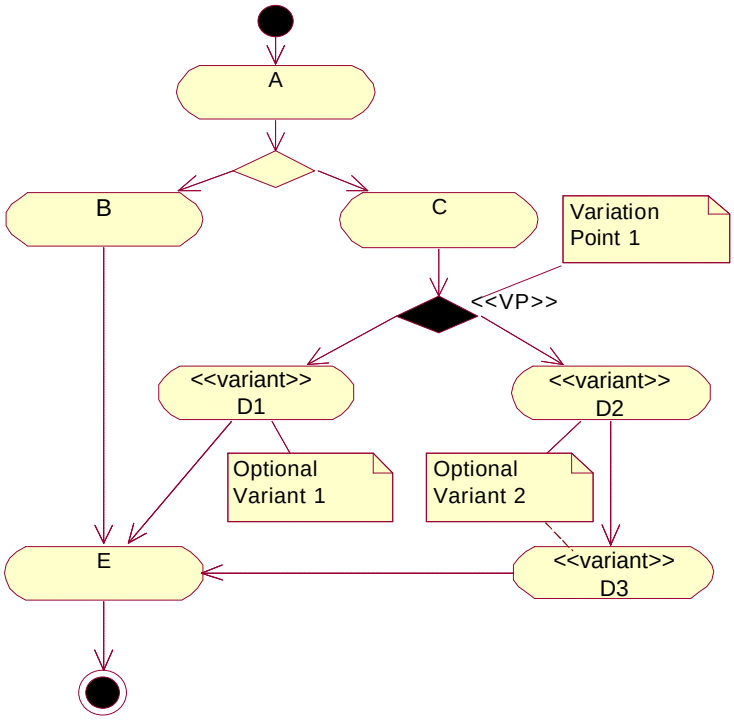

Figure 4: Variation point in a branch of a V-activity diagram

The second case is that if a variation point does not appear in a bottleneck, as shown in Figure 4, then the other non-variant scenarios do not need to contain a variation point. That is, variation points in a use case do not necessarily affect all derived scenarios. From the activity diagram in Figure 4, two scenarios would result. The first step leads to a test scenarios without variability {A, B, E} and one scenario with a variation point {A, C, $\{\}_{vp1}$, E}. The later is extended in the second step to {A, C, $\{\{D1\}_{v1}$, $\{D2, D3\}_{v2}\}_{vp1}$, E}.

Third, a variant may include normal decisions are well, as shown in Figure 5. In this V-activity diagram, the decision between {D1} and {D2, D3} is such an ordinary decision. They are marked as <<variant>>, because they require the selection of the corresponding variant B. If a variant contains ordinary decisions among different branches, then conventional branch coverage is applied to these branches. From the activity diagram in Figure 5, one scenario would result in the first step: {A, $\{\}_{vp1}$, E}. This scenario contains variability and is therefore refined in the second step. This step leads to two domain test scenarios: {A, $\{\{B, D1\}_{v1}$, $\{C\}_{v2}\}_{vp1}$, E} contains a variant and covers the {D1} branch, and {A, $\{B, D2, D3\}_{v1,vp1}$, E} covers

the {D2, D3} branch. The subscript indicates that the latter scenario is relevant only for variant 1 of variation point 1.

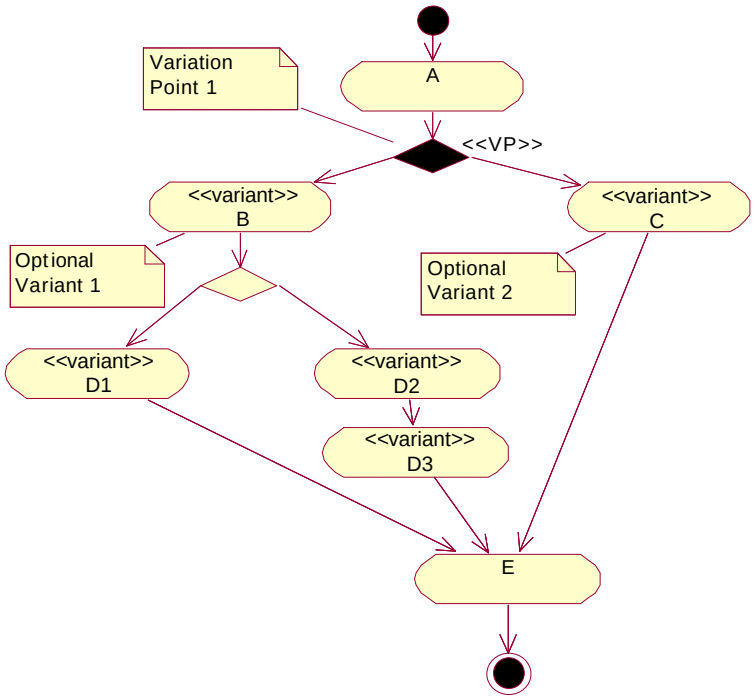

Figure 5: Decision in a variant branch of a V-activity diagram

The fourth case appears with mandatory variants. Scenarios are derived for the mandatory activities. The scenarios with the mandatory activities are always part of later test cases and should therefore explicitly be defined. The example given in Figure 6 would result in the two scenarios {A, B1, C} and {A, $\{\}_{,vp1}$, C} in the first step. The second step refines the second scenario to {A, $\{B2\}_{v2,vp1}$, C}. This step does not really contain variability, but the indicator shows that the scenario is only applicable if variant 2 from the variation point 1 has been chosen.

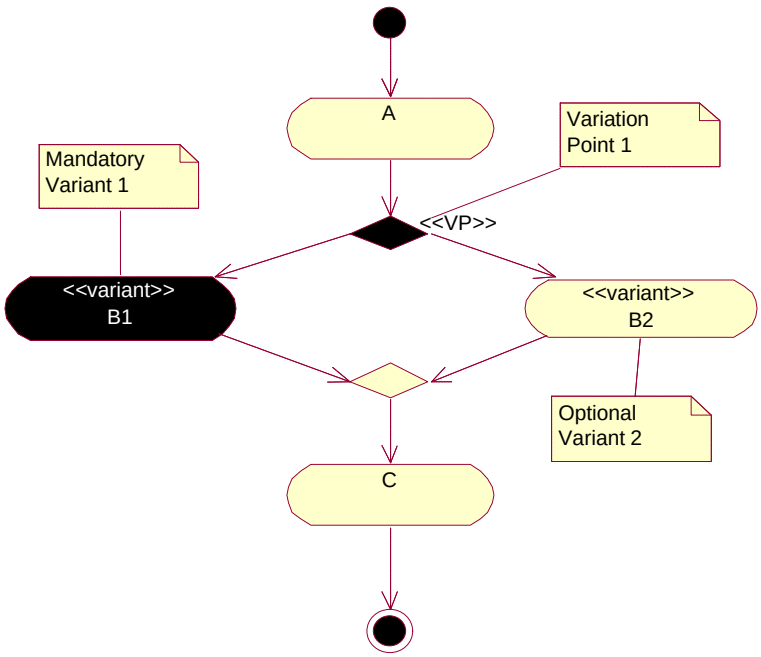

Figure 6: Derivation with mandatory variant

The suggested procedure worked nicely on the investigated cases. The application of the steps on the cases led to scenarios that fulfilled the criterion. All other cases can be reduced to these or considered as a combination of them.

5 Conclusion and Future Work

The product family branch coverage criterion and the operational procedure allow the structured derivation of domain test scenarios based on activity diagrams that contain variability. These scenarios cover all branches for each possible application. Therefore, the domain test scenarios form an adequate basis for later test cases. A possible representation of domain test cases by sequence diagrams is described in [6]. The derivation of test scenarios can be carried out early in requirements engineering and help spot gaps in requirements if performed by a test engineer.

We have followed the same thoughts for representing variability in activity diagrams as proposed in [10]. Our representation has the advantage that variation points have been made explicit. We have further introduced more concepts of variability in the diagram, as we have shown with mandatory and optional variants as well as with the alternative and co-existing variants in the variation point. These extensions are used during the derivation of domain test cases. For instance, the mandatory variants lead to invariant test scenarios.

We aim at validating the branch coverage criterion and the procedure for deriving test scenarios also in industrial case studies. Moreover, we aim to create tool support for scenario derivation. This is getting very important for hierarchical (nested) activity diagrams that are typical for complex systems found in industrial settings. Manual scenario derivation is nearly impossible for such diagrams, because of the amount of activities included in the diagrams.

References

[1] Bachmann, F.; Bass, L.; “Managing Variability in Software Architectures”, *Proceedings of the Symposium on Software Reusability* (SSR`01), 2001.

[2] Briand, L.; Labiche, Y.; “A UML-Based Approach to System Testing”, *2nd Intl. Conference on the Unified Modeling Language (UML 2001),* Toronto, Canada, 2001.

[3] Halmans, G.; Pohl, K.; “Communicating the variability of a software-product family to customers”, *Journal of Software and Systems Modeling*, Springer, 2003.

[4] Kamsties, E.; Pohl, K.; Reuys, A.; Reis, S.; “Use Case- and Architecture-based Derivation of Generic Test Cases for System and Integration Tests for Software Product Families“, *CAFÉ Deliverable SSE-WP4-20020930-01 (Private)*, University of Essen, October 2002.

[5] Kamsties, E.; Pohl, K.; Reuys, A.; “Supporting Test Case Derivation In Domain Engineering“, *accepted for publication at the 7th World Conference on Integrated Design and Process Technology* (IDPT-2003), Austin, USA, December 2003.

[6] Kamsties, E.; Pohl, K.; Reis, S.; Reuys, A.; “An Initial Technique for Deriving Test Cases for System and Integration Testing of Software Product Families“, *CAFÉ Deliverable SSE-WP4-20030630-01 (Private)*, University of Duisburg-Essen, June 2003.

[7] Kim, Y.G.; Hong, H.S.; Cho, S.M.; Bae, D.H.; Cha, S.D.; “Test Cases Generation from UML State Diagrams”*, In IEE Proceedings – Software*, Vol.146, No. 4, pages 187-192, August 1999.

[8] Myers, G.J.; *The art of software testing*, Wiley, 1979.

[9] Paech, B.; “On the Role of Activity Diagrams in UML – A User Task Centered Development Process for UML”, *1st Intl. Workshop on the Unified Modeling Language (UML 98)*, 1998.

[10] Riebisch, M.; Böllert, K.; Streitferdt, D.; Franczyk, B.; “Extending the UML to Model System Families”, *Integrated Design and Process Technology (IDPT-2000)*, 2000.

[11] Winter, M.; *Quality Assurance for Object-oriented Software – Requirements Engineering and Testing w.r.t. Requirements Specification*, Ph.D. Dissertation (in German), University of Hagen, Germany, Sept. 1999.