A Requirement-based Approach to Test Product Families

Clémentine Nebut, Franck Fleurey, Yves Le Traon and Jean-Marc Jézéquel

IRISA,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, France
{Clementine.Nebut, Franck.Fleurey, Yves.Le_Traon, Jean-Marc.Jezequel}@irisa.fr

Abstract. Use-cases have been identified as good inputs to generate test cases and oracles at requirement level. To have an automated generation, information is missing from use cases, such as the exact inputs of the system, and the sequential constraints between the use cases. The contribution of this paper is then two-fold. First we propose a contract language for PF functional requirements expressed as parameterized use cases; this language supports the specification of variant parts in the requirements. Then we provide a method, a formal model and a prototype tool to automatically generate both functional and robustness test cases specific to a product from the PF requirements. We study the efficiency of the generated test cases on a case study.[1]

1 Introduction

Product families (PF) are currently undergoing a resurgence of interest, some factors being the relentless pace of hardware development, the growing complexity of software systems, and the need to respond promptly to more rapidly-changing consumers habits. PF conception and design brings up a large number of novel issues, among them, PF testing methods. As underlined in [8], PF requirements assist design evaluation, and consequently play a crucial role in driving the functional testing task. However, there is a real difficulty in ensuring that the requirements of a PF are satisfied on all the products. In other words, *requirements-based testing* of a product family is an arduous task, which is in serious need of automation. In this paper, we propose a method implemented in a prototype tool, which allow product-specific test objectives to be generated from the PF requirements (expressed in the UML with use cases). Our approach aims at using the sequential dependencies existing between the use cases to generate relevant test objectives. We propose to associate contracts to the use cases of the PF, to express those dependencies. Those contracts are then interpreted, an execution model is built for each product, and thanks to adequate test criteria, test objectives are automatically generated by our prototype.

The rest of the paper is organized as follows. Section 2 presents a contract language for PF functional requirements and illustrates it on a case study. Section 3 details

[1] This work has been partially supported by the Families European project. Eureka $\sum$ 2023 Programme, ITEA project 02009

the generation of functional and robustness test objectives from requirements enhanced with contracts. This generation is based on a use case labeled transition system, and we propose test coverage criteria to extract test objectives from this model. Section 4 is an experimental part aiming at studying on a case study both the relevance of the generated test objectives and their effectiveness. The key question addressed is to determine if test cases generated at very high level of a PF are still relevant at product-specific code-level. Section 5 and 6 respectively briefly states the related work and gives our conclusions and future work.

2 A contract language for functional PF requirements

In this section, we present a way to express the sequential constraints existing between the use cases of a PF, remaining within the UML. In general, the partial order existing between the uses cases is given in their textual description, or just left implicit. We here propose to make it explicit in a declarative way, associating contracts (i.e. pre and post conditions) to the use cases. Using contracts is a good way to express easily and quickly the mutual obligations and benefits among functional requirements, expressed with use-cases. Pre and post conditions are attached as UML notes to each use case, and are first-order logical expressions. In the following, we present our case study and then use it to illustrate the contracts.

2.1 A case study: a virtual meeting server PF

Our case study is a virtual meeting server PF offering simplified web conference services. It is used in the advanced courses of the Univ. of Rennes. The whole system contains more than 80 classes but a simplified version is presented here with few variants for the sake of readability (only functional variants appear since we address functional testing). Our case study is thus a simplistic example of PF, but is sufficient to illustrate our method.

The virtual meeting server PF (VMPF) permits several different kinds of work meetings to be organized on a distributed platform. When connected to the server, a user can enter or exit a meeting, speak, or plan new meetings. Each meeting has a manager. The manager is the participant who has planned the meeting and set its main parameters (such as its name, its agenda, ...). Each meeting may also have a moderator, designated by the meeting manager. The moderator gives the floor to a participant who has asked to speak. Before opening a meeting it, he or she may decide that it is to be recorded in a log file. The log file will be sent to the moderator after the closing of the meeting. The corresponding use case diagram is given on Figure 1.

Three types of meetings exist:

– *standard meetings* where the current speaker is designated by a moderator (nominated by the organizer of the meeting). In order to speak, a participant has to ask for the floor, then be designated as the current speaker by the moderator. The speaker can speak as long as he or she wants; he or she can decide to stop speaking by sending a particular message, on reception of which the moderator can designate another speaker.

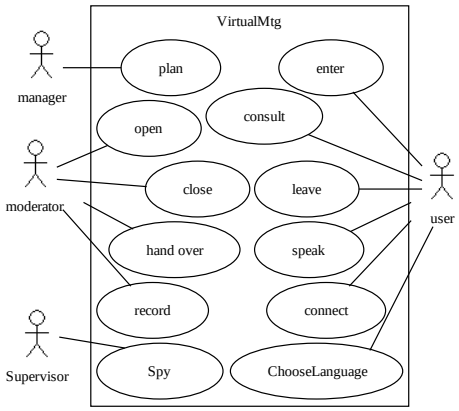

Fig. 1. Use case diagram

- *democratic meetings* which are like standard meetings except that the moderator is a FIFO robot (the first client to ask for permission to speak is the first to speak)
- *private meetings* which are standard meetings with access limited to a certain set of users.

We define our PF describing the variation points and products (the commonalities corresponding to the basic functionalities of a virtual meeting server, as described above).

Variation points In this article, for the sake of simplicity, we only present 5 variation points in our VMPF:

- the limitation or lack of it, of the number of participants to 3.
- the type of meetings available; possible instantiations correspond to a selection of 1, 2, or all of the 3 types of possible meetings.
- the presence or absence of a facility enabling the moderator to ask for the meeting to be recorded.
- the languages supported by the server (possible languages being English, Spanish, French).
- the presence or absence of a supervisor of the whole system, able to spy and log it.

The other variation points which are not described here concern the presence of a translator, OS on which the software must run, various interfaces - from textual to graphical, network interface etc... Testing all the possible products independently is inescapable. In our case, this would mean testing (2*7*2*7*2*3*3)=2352 products (considering 3 OS and 2 GUIs). In order to simplify the presentation, in this paper we only consider 3 products (a demonstration edition, a personal edition, and an enterprise edition). However, this does not in any way reflect a restriction on the method. The characteristics of the 3 products are given in table 1.

edition	demonstration	personal	enterprise
meeting limitation	true	true	false
meeting types	{std}	{std, democ, priv}	{std, democ, priv}
recording	false	false	true
language	{En}	{En}	{En, Fr, Sp}
supervisor	false	false	true

Table 1. Variation points and products

The *demonstration edition* manages standard meetings only, limited to 3 participants. Only English is supported, and no supervisor is present.

The *personal edition* provides the three kinds of meetings but still limits the number of participants. Only English is supported, and no supervisor is present.

The *enterprise edition* limits neither the type of meeting nor the number of participants. It is possible for the moderator to record the meeting. English, Spanish and French re supported: each participant chooses his or her preferred language, the default being English. A supervisor is present.

2.2 Use cases parameters and contracts

We here present our contract language for requirements. The declarative definition of such contracts expressions is simple to achieve and forces the requirement analyst to be precise and rigorous in the semantics given to each use case, being in the same time flexible and easy to maintain and to modify: writing contracts is quite an easy task as soon as the use cases are well defined.

Use cases parameters We consider parameterized use cases ; parameters allow us to determine the inputs of the use case (denoted UC in the following). Actors involved in the use case are particular parameters. For example, the *enter* use case is parameterized by the entering participant, and the entered meeting. It is expressed as follows:

```
UC enter (u:participant, m:meeting).
```

Parameters can be either actors (like the participant *u* in the UC enter) or main concepts of the application (like the meeting *m* in our example). Those main concepts will probably be reified in the design process, and are pointed out as business concepts in the requirements analysis. All types are enumerated types, they are only needed when the use case orderings are deduced (from the execution model presented below). For example, each participant and each meeting are declared by a specific label.

Contracts: logical expressions on predicates The UC contracts are first-order logical expressions on predicates, that are declared as follows:

```
UC plan (u:participant, m:meeting).
pre logical-expression
post logical-expression
```

A predicate has a name, and a set of typed formal parameters potentially empty (those parameters are a subset of the use cases parameters). The predicates are used to describe facts (on actors state, on main concepts states, or on roles) in the system. In this way, the predicate's names may generally be either semantical derivatives of a use case names (as *opened*), or role names (as *moderator*) or a combination of both. The predicates names are semantically rich: in this way, the predicates are easy to write and to understand. In order for the contracts to be fully understandable, the semantics of each predicate has to be made explicit, the most precisely possible so as to avoid any ambiguity in the predicate's sense. As an illustration, here are two examples of predicates, with their semantics:

- *created(m)* is a predicate which is true when the meeting *m* is created and false otherwise;
- *manager(u,m)* is a predicate which is true when the participant *u* is the manager of the meeting *m* and false otherwise.

Since classical boolean logic is used, a predicate is either true or false, but never undefined.

The precondition expression is the guard of the use case execution, and the postcondition expresses the new values of the predicates after the execution of the use case. The operators are the classical ones of boolean logic: the conjunction (*and*), the disjunction (*or*) and the negation (*not*). The implication (*implies*) is used to condition a new assertion with an expression. It allows the specification of results depending on the preconditions of a use case. Quantifiers (*forall* and *exists*) are also used in order to increase the expressive power of the contracts.

Contracts in the PF context In a PF, some requirements are common to all the products and some of them are only present in certain products, depending on variation points. That is why our contract language allows to specify which parts of the requirements depend on a certain variant. We thus propose to add tags (in fact UML tagged values) on contracts or on use cases, specifying which variants they depend on. If a tag is attached to an element e, then e is valid only for the product selected by this tag, i.e. the product owning one of the variant specified in the tag. By default, an element e with no tag is valid for all the products. The format of those tags is: *VP{variant_list}*, where *VP* is a variation point name, and *variant_list* is a list of instantiations of the variation point. For example, in our VMPF, the tag *recording{true}* selects the product owning a recording facility, i.e. the enterprise edition, and the tag *language{En}* selects the products handling the English language, i.e. all the products. Several contracts of the same type can thus be added to the same element, if they are tagged differently. When several preconditions (resp. postconditions) are selected for a same product, they are conjuncted.

An example of contracts is given on figure 2: the use case *enter* requires the entering participant u to be connected, and the entered meeting m to be opened. For a private meeting, u must be authorized in m, and for limited meetings, there must be strictly less than 3 participants already entered in m.

UC enter(u:participant; m:mtg)
pre connected(u) **and** opened(m)
pre priv(m)}**implies** authorized(u,m) *{VPMeetingType(priv)}*
pre not exists (u,v,w:participant) {entered(u,m) **and** entered(v,m) **and** entered(w,m)
and u/=v **and** v/=w **and** w/=u} *{VPLimitation(true)}*
post entered(u,m)

Fig. 2. Contracts of the use case *enter*

From a set of use cases with contracts of a product family, and using the characteristics of each product given in terms of variants, a set of use cases with contracts can be automatically built for each product, following Algorithm 1.

Algorithm 1 Algorithm to extract a product requirements from the product family requirements

```
algorithm extractRequirementsForAProduct
param p: the product
result : requirements R(p) for p
for each use case uc in the PF requirements
  if no tag is present or p.satisfies(tag)
  then
    add uc to R(p)
  end
end
for each use case uc in R(p)
  for each precondition in uc
    if a tag is present and not p.satisfies(tag)
    then
      remove precondition
    end
  end
  for each postcondition in uc
    if a tag is present and not p.satisfies(tag)
    then
      remove postcondition
    end
  end
end
return R(p)
```

3 Automatic test generation from Use cases enhanced with contracts

In this section, we explain how we exploit the enhanced use cases to generate test objectives. We first build a use case transition system per product and then exploit it with several criteria to generate relevant test objectives.

3.1 The Use Case Transition System

From a requirement analysis leading to a set of use cases enhanced with pre and post conditions, we propose to build a representation of the valid sequences of use cases. Since pre and post conditions contain parameters, this representation also deals with those parameters. In fact, the idea is to "instantiate" the use cases with a set of values replacing its parameters. As an example, in the virtual meeting, we want to obtain the ordering of use cases with 2 participants *p1* and *p2*, and a meeting named *m1*. The instantiated use cases of *plan(p:participant,m:meeting)* are *plan(p1,m1)* and *plan(p2,m1)*. In the following, we call *instantiated use cases* (resp. *predicates*) the set of use cases (resp. predicates) obtained by replacing its set of parameters by all the possible combinations of their possible specific values.

A transition system to represent valid sequences of use cases The valid sequences of use cases are represented by a transition system M defined by $M = (Q, q_0, A, \hookrightarrow)$ where:

- Q is a finite non-empty set of states, each state being defined as a set of instantiated predicates,
- q_0 is the initial state,
- A is the alphabet of actions, an action being an instantiated use case,
- $\hookrightarrow \subseteq Q \times A \times Q$ is the transition function.

We call such a transition system a Use Case Transition System (UCTS). States of the UCTS represent the state of the system (in terms of value of predicates) at different stages of execution. Transitions, labeled with an instantiated use case, represent the execution of an instantiated use case. A path in the UCTS is thus a valid sequence of use cases. A partial UCTS obtained for the demonstration edition is given on figure 3.

Due to its finite set of states (itself due to the finite number of combinations of predicates), the UCTS is itself finite. Its maximal size in the worst case is $2^{n_p \prod_{i=1}^{n_v} v_i}$ states, where n_v is the number of types used, v_i is the number of possible values for each type *i*, and n_p is the number of predicates. This maximal size is reached when all the predicates use all the types, and when all the possible states are reachable from the initial state. In practice, this maximal size is never reached. For the demonstration edition with 3 participants and one meeting, the UCTS has 11600 states. The size of the UCTS grows up with the number of use cases, and thus with the complexity of the system to test. Thus for large system, building the UCTS is very expensive in memory and in time. To tackle this problem, work in underway to use an abstraction on certain parameters, in order to reduce the UCTS size.

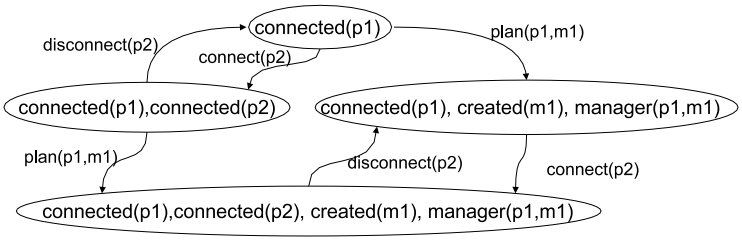

Fig. 3. UCTS extract

Building algorithm The first step to build a UCTS for each specific product is to extract the requirements for each product from the PF requirements. This is simply done parsing the variation notes in the requirements, and using a table such as Table 1. Then, for each product-specific requirements, Algorithm 2 is applied to build the UCTS. Upon initialization, the initial state is deduced from the initial true predicates. Then the algorithm tries successively to apply each use case with all the combinations of effective parameters, as a puzzle game. Applying a use case is possible when its precondition is true w.r.t the set of true predicates contained in the current state's label, and leads to create an edge from the current state to the state representing the system after the postcondition is applied. The algorithm stops when all the reachable states are explored.

To obtain instantiated use cases from the formal description of use cases, the instances of each type are given, it corresponds to the enumeration of all the types of the system. In practice, we give to the building algorithm all the instances it has to deal with, under the form of a declaration. As an example, to deal with 3 participants and 1 meeting, we declared:

```
user1, user2, user3:PARTICIPANT
m1: MEETING
```

3.2 Test case generation w.r.t coverage criteria

A UCTS is a representation of all the possible orderings of use cases. Thus from a UCTS, we aim at generating test objectives w.r.t a given UCTS covering criterion. We define in this sub-section one **structural criterion** to cover a UCTS and one **semantical criterion**, which is not based on the UCTS itself but on the contracts system.

Test objective: A test objective (TO) is defined here as a sequence of instantiated use cases. Note that generally, a test objective cannot be directly used on an implementation: a test case generator is needed to obtain test cases from TOs.

Test objectives set consistency with an UCTS: A test objectives set is said to be consistent with an UCTS iff each TO exercises a path of the UCTS. A path in the UCTS is here defined as the classical notion of path in a graph, the first vertex corresponding to the initial state.

Algorithm 2 Algorithm producing the UCTS

```
algorithm buildUCTS
param initState: STATE ; useCases : SET[ACTION]
var
  result : UCTS
   to_visit : STACK[STATE]
  currentState : STATE
  newState : STATE
init
  result.initialState←initState
  to_visit.push(initState)
body
  while (to_visit≠∅)
  do
    currentState←to_visit.pop
    ∀ uc ∈useCases | currentState ⇒uc.pre
      do
        newState ← apply(currentState, uc)
      if newState ∉result
      then
          result.Q ←result.Q ∪ {newState}
          to_visit.push(newState)
      fi
      result.↪ ←result.↪ ∪ {(currentState,uc,newState)}
    done
  done
end
```

All Instantiated Use Cases criterion: (AIUC) A test objective set *TOs* satisfies the *all instantiated use cases* coverage criterion for a use case transition system *ucts* iff each instantiated use case of the system is exercised by at least one TO from *TOs*.

All Precondition Terms criterion: (APT) A test objective set *TOs* satisfies the *All Precondition terms* criterion for a contracts system iff each use case is exercised in as many different ways as there are predicates combinations to make its precondition true.

We defined other criteria such as covering all vertices or all edges of the UCTS but they were leading to inefficient tests (*all vertices*) or to a too large number of tests (*all edges*). Intuitively, the criterion *All Precondition Terms* (APT) guarantees that all the possible ways to apply a use case are exercised: a use case can be applied when its precondition is true ; this precondition being a logical expression on predicates, there are several valuations of the predicates which makes it true (as an example, if a precondition is $a\ or\ b$, 3 valuations makes it true: $(a, b);\ (a, not\, b);\ (not\, a, b)$). The criterion *APT* will find sequences of use cases such that each use case is applied with all the possible valuations of the expression : $(precondition = true)$. To implement this criterion, all those valuations are computed, and then paths in the UCTS are found to reach states that verify those constraints. The two criteria are implemented with a breadth first search of

the UCTS from the initial state. Such a technique ensures that the obtained TOs sets are consistent with the considered UCTS. The choice of a breadth-first visit is made in order to obtain smaller TOs: small tests are more meaningful and humanely understandable than larger ones.

3.3 Robustness testing

The tests generated as described above ensure that the application under test fits the requirements, but do not verify that violating contracts causes errors. To generate robustness tests from enhanced UCs, the contracts must be detailed enough so that all the unspecified behaviors are incorrect. If so, the UCTS built from the enhanced use cases will be used as an oracle for the robustness. To improve the contracts, we propose to use a requirement simulator, which allows the requirement analyst to see step by step which use cases are applicable, and then to determine if his or her requirements are sufficient. This simulator interactively computes valid sequences of instantiated use cases: all the choices are made by the simulator's user, by selecting an instantiated use case in a list of all the applicable instantiated use cases.

As soon as the requirements are precise enough, the generated UCTS can be used as an oracle for robustness tests. The principle is to generate paths that lead to an invalid application of a use case. The idea is thus to exercise correctly the system and then make a non specified action. The execution of such a robustness test must lead to a failure (in our example, the receipt of an error message). If not, a robustness weakness has been detected. The goal is thus to test the robustness/defensive code of the system. The difficulty is to propose an adequate criterion and the UCTS plus the contracts provide all the information we need for that purpose. The criterion we use to generate robustness paths with the UCTS is quite similar to the *All Precondition Terms* one: for each use case, it looks for all the shortest paths leading to each of the possible valuations that violate its precondition.

Robustness criterion : A test objective set *TOs* satisfies the robustness criterion for a contracts system iff each use case is exercised in as many different ways as there are predicates combinations to make its precondition false.

The robustness tests test the defensive code of the application, which is not tested with the functional tests previously generated. Joining the two sets of tests, not only will we test that the application does what it should (according to the requirements) but also that it does not what it should not.

4 Experimental validation: test cases generation and efficiency measures

This section offers an experimental validation of the proposed approach. As explained in Section 3, our test generation from requirements provides the tester with test objectives, i.e. sequences of instantiated use cases. To produce concrete test cases from TOs, a mapping has to be performed. For our study, we used an ad hoc test case generator, since a UC exactly corresponds to a command of the system in our example.

We obtained the test cases using a template associated to each use case giving the syntactic requirements of the implementation. In this section, we give an overview of the test objectives synthesized for the 3 products, then we study the efficiency of the tests generated for the demonstration edition.

4.1 Test generated for the 3 products

From the PF use cases enhanced with contracts, we derived one specific UCTS per product, and then we generated the test cases (TC). Statistics are given in table 2 (demonstration, personal and enterprise edition are respectively denoted DE, PE, and EE). A study of those TCs reveals that common tests have been generated (corresponding to commonalities of the PF), and specific tests have been generated for each product, due to the different combinations of variants in the products.

Edition	DE	PE	EE
# generated TC with AIUC	50	65	78
# generated TC with APT	15	18	21
# generated TC for robustness	65	110	128
average size of the tests	5	4	4

Table 2. Statistics on generated tests

4.2 Study of the generated test efficiency for demonstration edition

For the experimental validation, we used a Java implementation of the virtual meeting. Around 9% of the code is dead code. Nevertheless, this code is relevant: it consists of pertinent but unused accessors, which could be used in future evolutions of the system. Functional testing cannot deal with this code: it has to be tested during the unit test step. For the following study, we removed those 9% of dead code to focus on the efficiency of our tests on reachable code. Around 26% of the code is robustness code: robustness w.r.t. the specification which asserts that only the required functions are present, and robustness w.r.t. the environment which asserts that the inputs coming from the environment are correct.

The results of the code coverage measures are given in figure 4. The APT (resp. AIUC) criterion covers 71% (resp. 60%) of the functional code. Note that since the AIUC criterion generates much more TC than the APT one, the APT criterion is more efficient in terms of covered statement per TC. Since our robustness tests stem from functional requirements, they cannot cover all the robustness code but they cover 100% of the robustness code w.r.t. requirements. The uncovered code concerns syntactic verification of the inputs treatment of network exceptions, these aspects are specific to

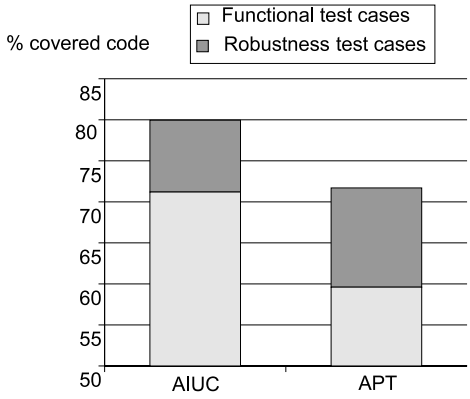

Fig. 4. Code coverage of the tests

the distributed platform. Globally, the robustness tests add a 10% code coverage to the functional tests.

This study shows that tests generated from the requirements expressed at the PF level are relevant at the product code level, with the use of adequate criteria.

5 Related work

While OO testing is becoming an important research domain [3,9] and while the PF approach is increasingly being used in industry [4,5], among the numerous test case generation techniques that can be found in the literature, few of them have been adapted to the PF context. The survey of [9] shows that until now, most test case generation has been done manually, the only test automation being in the generation of test scripts or test harnesses. It insists on the difficulty and the cost of writing test-case generators for PFs. If some PL testing process are proposed in the literature, few of them are automated: for example, in [1], a three-phase, methodological, and code-based testing approach is proposed and in [2], a testing process is detailed to perform functional, structural and state-based testing of a PL.

Independently from the PL context, the main contribution for system testing from use cases can be found in [6], where the authors propose to express the sequential constraints of the use cases with an extended activity diagram. If activity diagrams can seem suitable to this purpose, we soon discovered that they quickly become far too complicated and unreadable. On the opposite, contracts on the use cases remain a simple way to express the dependencies.

6 Conclusion and future work

This paper presents a requirement-based testing technique for PF, that utilizes the PF use-cases to generate product-specific functional test objectives. This technique is as-

sociated with a light formalism to express at the PF level, in a declarative way, but unambiguously, the mutual dependencies between requirements, in terms of pre/post conditions (kind of "contracts"). From these enhanced requirements, a labeled transition system per product is built that captures all possible valid use-case sequences from an initial configuration, and coverage criteria are used to extract relevant correct sequences which cover a large part of the product code. Though our approach certainly not replace integration or unit testing stages, that allow specific aspects (that are not described at very high level) to be covered, for the strict coverage of functional and robustness code, results are good since all the code that could be covered has been covered. All our method is supported by a prototype tool, which is integrated in the Objecteering CASE tool.

Future work concerns the building of a generic mapper from test objectives to test cases, using scenarios, as well as a validation of our approach with another case study. We will to adapt it to requirements classification methods such as the one of [7].

References

1. Mark Ardis, Nidel Daley, Daniel Hoffman, and David Weiss. Software product lines: a case study. *Software - Practice and Experience*, 30(7), 2000.
2. Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barabara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based product line engineering with UML*. 2002.
3. Robert V. Binder. *Testing object-oriented systems*, chapter 8. Addison-Wesley, 2000.
4. Jan Bosch. Product-line architectures in industry: a case sudy. In *Proc. of the 1999 International Conference on Software Engineering (ICSE 99*, pages 544–554, New York (USA), 1999.
5. L.G. Bratthal, R. Van Der Geest, H. Hofmann, E. Jellum, Z. Korendo, R. Martinez, M. Orkisz, C. Zeidler, and J.S. Andersson. Integrating hundred's of products through one architecture - the industrial IT architecture. In *Proc. of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 604–614, New York (USA), 2001.
6. L. Briand and Y. Labiche. A uml-based approach to system testing. *Journal of Software and Systems Modeling*, pages 10–42, 2002.
7. J. Kuusela and J. Savolainen. requirements engineering for product families. In *Proc. of the 2000 International Conference on Software Engineering*, pages 61–69.
8. R.R. Lutz. Extending the product family approach to support safe reuse. *journal of systems ans software*, 53:207–217, 2000.
9. J.D. McGregor and D.A. Sykes. *A practical Guide to Testing Object-Oriented Software*. Addison Wesley, 2001.