System Testing of Product Lines: From Requirements to Test Cases

Clémentine Nebut, Yves Le Traon, Jean-Marc Jézéquel

To cite this version:

Clémentine Nebut, Yves Le Traon, Jean-Marc Jézéquel. System Testing of Product Lines: From Requirements to Test Cases. K. Pohl. Software Product Lines, Springer Verlag, pp.447-478, 2006, 978-3-540-33252-7. inria-00512533

HAL Id: inria-00512533

https://inria.hal.science/inria-00512533

Submitted on 30 Aug 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

12 System Testing of Product Lines: From Requirements to Test Cases

C. Nebut, Y. Le Traon, and J.-M. Jezequel

Abstract

Product line processes still lack support for testing end-product functions by taking advantage of the specific features of a product line (commonality and variabilities). Indeed, classical testing approaches cannot be directly applied on each product since, due to the potentially huge number of products, the testing task would be far too long and expensive. There is thus a need for testing methods, adapted to the product line context, that allow reducing the testing cost. The approach we present is based on the automation of the generation of application system tests, for any chosen product, from the system requirements of a product line. These PL requirements are modeled using enhanced UML use cases which are the basis for the test generation. Product-specific test objectives, test scenarios, and test cases are successively generated through an automated process. The key idea of the approach is to describe functional variation points at requirement level to automatically generate the behaviors specific to any chosen product. With such a strategy, the designer may apply any method to produce the domain models of the product line and then instantiate a given product: the test cases derived from product-specific behaviors are executed against the chosen end product to check that the expected functionalities have been correctly implemented. The approach is adaptive and provides automated test generation for a new product as well as guided test generation support to validate the evolution of a given product.

12.1 Introduction

Product lines elaboration and design brings up a large number of novel issues, testing methods being one among them [2,14,19,20,22,23,26]. While the elicitation of product line requirements is known as crucial task for the elaborated design, product requirements are seldom used for driving the functional testing task. However, the end product is expected to satisfy its requirements: testing is the classical way to obtain confidence in a given product with respect to its requirements. There is thus a need of adapted techniques to assist this test generation from requirement in a PL context. Like any kind of software, product lines obviously require several types of software tests. In particular, unit testing has to be performed independently on each asset, integration testing techniques can be used to assemble the assets to obtain a product, and system testing ensures that the end product has the required features. We here focus on system and functional testing. One of the specific issues related to PL testing concerns the way a testing technique deals with the creation of new

products and the evolution of existing products. In this chapter, we present the automation as a relevant way for dealing with these issues.

Testing a PL is all the more tedious since the common and the shared variant requirements have to be tested for each instantiated product. Indeed, the same piece of functional test code, derived from a requirement, cannot be reused exactly: for instance, in an object-oriented product line, the objects addressed to realize a given functionality may be different from one product to another, due to the crossing of different variation points. For example, the initialization sequence leading to the testing of a particular point may be totally different from one product to another. So, for testing a given function common to all products, specific test cases may have to be written for each specific product. As a result, manually writing the tests cases for all the products is not conceivable, since it is far too expensive. Automating the test generation appears as a possible way to deal with these cost and time-to-market issues.

Many approaches already exist to automatically generate tests from the requirements of a "classical" software (e.g., [4,9,24]), but they have to be adapted in order to deal with the variability expressed in product line requirements. To benefit from the product line approach, there is a need for specifying the requirements of a product line and then deriving automatically the test cases. That means to solve several problems (1) How to express the product line requirements (and in particular the variability)? (2) How to generate tests from them? (3) Is it possible to generate test cases that can directly be applied by a test driver?

Our approach is a proposal to answer those questions. Our idea is to express the requirements using enhanced Unified Modeling Language (UML) [28] use cases or to transform the requirements into enhanced UML use cases. The UML use cases are enhanced in order to express commonality and variability, and enhanced with parameters and with contracts. Those use cases are also supposed to be documented by scenarios. Use cases and scenarios are combined to generate test objectives that are refined into test scenarios (a test scenario is a potentially abstract and incomplete representation of a test case). Then product-specific test synthesis is achieved to obtain test cases.

Two main approaches already exist to test PL from the use cases (see Chaps. 11 and 13 and [2,14]), that are complementary more than in opposition to ours. Though they have the same purpose of automating the testing task, the approach proposed here differs from the one proposed in Chap. 11 in the sense that this latter approach is data driven since it is an adaptation of the category-partition method, while ours is behavior driven. Our approach is also complementary to the ScenTED approach (Chap. 13) which is a systematic approach to derive test scenarios for product lines. It tackles in particular the issue of the test artifacts reuse. In Sect. 12.7.4, we explain how our approach could be coupled with those two approaches.

The rest of this chapter is organized as follows. Section 12.2 proposes an overview of the approach and presents an illustrative example. Section 12.3 details our requirement model, i.e., an enhanced use case model. Section 12.4 presents the simulation mechanism of the use cases. The simulation is used to generate test objectives, as detailed in Sect. 12.5. Section 12.6 explains how test cases can be derived from the test objectives, using test scenarios and behavioral test patterns to guide the test synthesis tools. Section 12.7 provides experiments and discusses our approach, in particular with respect to related work. Section 12.8 concludes.

12.2 Overview of the Approach

This section gives an overview of the proposed approach, which is summarized in Fig. 12.1. Each step of this approach will be detailed in a particular section in the following of the chapter. This section ends with the presentation of an illustrative example that is used all over this chapter.

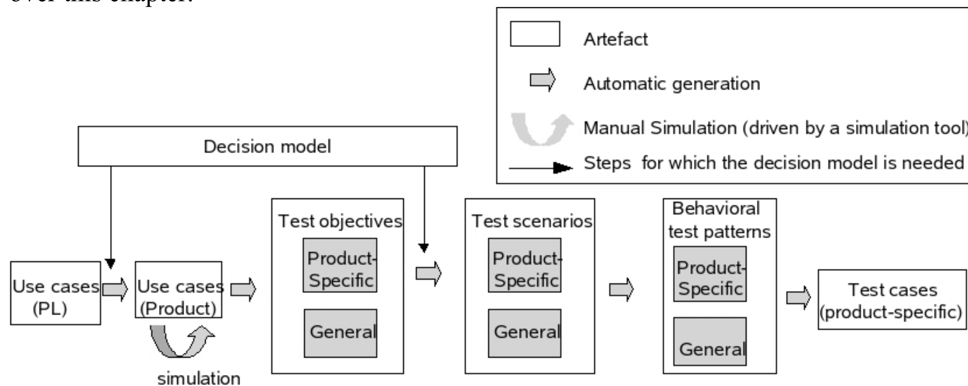

Fig. 12.1. Overview of the test case generation

12.2.1 From the Product Line Requirements to Product-Specific Requirements

Use cases are an easy and natural way to express system functional requirements. They are widely used in industry, probably since the underlying approach is simple and just consists in producing a structured document in natural language, for example following the Cockburn schema [7]. Thus we have based our approach on requirements written in the form of UML use cases. To be used as first input of an automated test generation process, UML use cases need to be formalized and specialized for the product line context. The formalization we propose first consists in making explicit the conceptual objects at business level that are implicit in the use cases; this leads to dealing with parameterized use cases. Use cases formalization implies the expression of the constraints linking them: use cases usually depend upon one another. The constraints are expressed locally on each use case using contracts, i.e., preconditions and postconditions, written in a dedicated use case constraint language (based on first-order logic).

The specification of the variation points in the use cases is also supported, allowing describing which parts are common to the product line, and which depend on a variation point.

From the knowledge of a decision model and the use cases describing the requirements of a product line, the product-specific requirements for each product can be automatically deduced. So, the strategy proposed is to go from the requirements expressed for the product line to the specific requirements that apply to a product; and then for every product the test case derivation method is (re)applied to its specific requirements. The test technique is thus functional/black box and does not include any specific tactic to deal with

PL typical variability and commonalities at design level. So, the force of the approach is to describe functional variation points at requirement level to automatically generate, based on the decision model, behaviors specific to any chosen product. Then, the designer may apply any method to produce the domain models of the product line and then instantiate a given product: the test cases derived from product-specific behaviors are executed against the end product to check whether the expected functionalities have been correctly implemented. In this approach, we do not take into account the design activities carried out to go from requirements down to domain applications, except for traceability purposes. However, we suggest bridging the gap between requirement level behaviors and the final design in two steps:

- Deriving test objectives into test scenarios by exploiting the sequence diagrams associated to use cases. Test scenarios may be combined in behavioral test patterns. A behavioral test pattern describes the expected and rejected behaviors of one execution of the product, but in an incomplete way, since the very specific design details are not known.
- By applying a test synthesis tool to generate the final test cases from each behavioral test pattern. The idea is to use a detailed description of the final design behaviors (typically expressed by statecharts associated to each active class) to extract from the end-product design the exact expected/rejected inputs/outputs of the end-product corresponding to a test scenario.

So, the “pattern,” i.e., the skeleton of the product-specific test scenario is expressed using only requirement and analysis views while the final test cases are extracted from the final detailed design, without care of the intermediate refinement steps. As is explained in the following, the use of a test synthesis tool is not currently possible, mainly because of traceability and design incompleteness issues and also due to tools limitations. However, if a model-driven approach is adopted, these limitations are to be overcome.

Taking an opposite solution – but for testing purposes – to the general tendency in PL engineering, where a topmost important feature is reuse and factorization, every time a variation is introduced, all test cases for a newly instantiated product are automatically derived again. We believe this is the most efficient way to update dynamically the test cases for a product. An improvement of the approach, which is beyond the scope of this chapter, would be to identify among those derived, the test cases which are affected by the newly introduced variations.

12.2.2 Simulating Product-Specific Requirements

Once the product-specific requirements have been expressed using an enhanced use case definition, they can be simulated. The simulation process allows the requirement analyst to check whether the requirements are correct, which is of prime importance. Indeed, it is necessary to get trust in the requirements correctness before building the derived artifacts (such as system tests and analysis documents). Simulation is also the basis for the test objectives generation.

12.2.3 Generation of the Test Objectives

The simulation model is based on a transition system deduced from the use case description, and especially from the contracts. Not only does simulation allow the requirements analyst to ensure that her requirements are correct (from her point of view), but it also makes the test generation possible. We define a set of test criteria based on the simulation model to generate interesting paths of the simulation model, called test objectives. Such test objectives are obtained in the form of sequences of use cases with actual parameters.

12.2.4 Generation of the Test Scenarios

The generated test objectives are high-level tests that have to be progressively refined into test cases. A first step of this refinement consists in transforming the test objectives into test scenarios, using the sequence diagrams attached to each use case. A test scenario is a sequence diagram representing a test case, but in which there can be missing messages.

12.2.5 Behavioral Test Patterns and Synthesis of Test Cases

To transform the test scenarios into test cases, we propose to use synthesis tools. Test synthesis tools are originally used for testing telecommunication and distributed software. From a final design which describes precisely the expected product behaviors (using statecharts), a test synthesis tool automatically extracts the exact test cases which are the refinements of a test scenario. In our approach, the test synthesis tools are guided with particular test purposes called behavioral test patterns, and derived from the test scenarios.

12.2.6 An Illustrative Example of Product Line

The illustrative example that will be used all over the presentation of the method is a virtual meeting system offering simplified web conference services. The same system has been implemented in Java, Eiffel, and C# languages. It is used in the advanced courses of the University 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). The case study is complete enough to illustrate our method. The virtual meeting server PL (VMPL) 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, etc.). Each meeting may also have a moderator, designated by the meeting manager. The moderator gives the floor to a participant who has been asked to speak. Before opening a meeting, 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. 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.
- Democratic meetings which are like standard meeting 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 PL describing the variation points and products (the commonalities corresponding to the basic functionalities of a virtual meeting server, as described above). For the sake of simplicity, we only present 5 variation points in our product line:

- The limitation or lack thereof upon the number of participants to three.
- The type of available meetings; 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, the operating system (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*2 = 2352 products (considering 3 OS and 2 GUIs), since the meetings can be limited or not (2 combinations), there can be 1, 2 or 3 types of meeting available among 3 types (7 combinations), the meetings can be recorded or not (2 combinations), there can be up to 3 languages supported (7 combinations), the system can be spied or not, there are 3 kinds of OS (3 combinations) and 2 GUIs (2 combinations). In order to simplify the presentation, in this chapter 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 the following Tab. 12.1.

Table 12.1. Variation points and products

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

12.3 An Enhanced Use Case Model for Product

Use cases are good entry points for test generation [3,4,9,24], and several proposals exist to adapt use cases to the product line context [1,5,10,11,13]. We detail in this section the use case model that is the foundation of our test generation process.

12.3.1 Enhancing Use Cases with Parameters and Contracts

Use case parameters. We consider parameterized use cases; parameters allow to determine the inputs of the use case (denoted UC in the following). For example, the use case *enter* 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 entities in the requirements analysis. All types are enumerated types, they are only needed for the simulation.

Use case contracts. Use cases are also enhanced with contracts that can be statically evaluated. This approach is inspired by Meyer's Design-By-Contract method [21]. The declarative definition of such contracts expressions forces the requirement analyst to be precise and rigorous in the semantics given to each use case, being at 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.

To write contracts that can be evaluated, we propose a Use case Constraints Language (UCL), based on first-order logic. The constraint language recommended by the UML is the OCL [27]; nevertheless, we believe that the OCL is not suitable for requirements phases. Indeed, the OCL has a syntax difficult to understand and requires a specific learning. We have thus defined the UCL, however it can be seen as a subset of the OCL, with syntactic sugar in order to have an easy-to-handle language. The UCL provides a rigorous model as a response to proposals such as the Catalysis approach [8], which suggests enhancing use cases with pre and post conditions, like any other action.

The UC contracts are first-order logical expressions on predicates. A predicate has a name, and a (potentially empty) set of typed formal parameters (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. 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, so as to avoid any ambiguity in the predicate's meaning. 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 specifying conditional contracts. Quantifiers (forall and exists) are also used in order to increase the expressive power of the contracts.
We also defined enumerated properties, for example, *meetingType* can be defined as an enumerated property. For the simulation, the various possible values of *meetingType* will be required (for example: *standard*, *democratic*, and *private*).

An example of such contracts is given below, for the use cases *open* and *close*.

Examples of Enhanced Use Cases

```
UC open(u:participant;m:meeting)
pre created(m) and moderator(u,m) and not closed(m) and not
opened(m) and connected(u)
post opened(m)

UC close(u:participant; m:meeting)
pre opened(m) and moderator(u,m)
post not opened(m) and closed(m)
and forall(v:participant)not entered(v,m) and not asked(v,m)
and not speaker(v,m)
```

12.3.2 Expressing Variability at the Use Case Level

The objective is to provide ways to specify which parts of the requirements depend on a particular variant, i.e., to document variability in use case models. The coarsest granularity level to define variability is the use case itself. A use case can be specific to the presence of certain variants, as for example the use case *Record*, which is only present in the products owning the recording facility.

Variability can also occur at the parameters level. In our example, for some products the use case *Open* owns a parameter representing the moderator of the meeting, and in the others, for which only democratic meetings can be planned, the use case *Open* does not own such a parameter.

The contracts may also depend on some variants. For example, in the case of limited meetings, the use case *enter* will have a precondition checking that the meeting is not already full.

Thus, to specify the variability, we have defined tags (in fact UML tagged values) for the following model elements: contracts, parameters, and use cases. Those tags are a way to specify which variants the model elements depend on. If a tag is attached to a given model element *e*, then *e* is taken into account only for the product selected by this tag, i.e., the product owning one of the variants specified in the tag. By default, a model element *e* with no tag is taken into account 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 virtual meeting product line, the tagged value *recording{true}* selects the product owning a recording facility, i.e., the enterprise edition, and the tagged value *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 differently tagged. When several preconditions (resp. post-conditions) are selected for a same product, they are conjuncted.

An example of contracts is given below: 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*.

Example of Variability in an Enhanced Use Case

```
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)
```

From a set of use cases with contracts for a product line, and using the decision model (i.e., characteristics of each product given in terms of variants), a set of use cases with contracts can be automatically built for each product, following the Algorithm 1.

```
algorithm extractRequirementsForAProduct
param p: the product
result : requirements R(p) for p

for each use case uc in the PL requirements
  if no tag t is present for uc or p.satisfies(t)
  then
    add uc to R(p)
  end
end
for each use case uc in R(p)
  for each precondition pre in uc
    if a tag t is present for pre and not p.satisfies(t)
    then
      remove pre
    end
  end
  for each postcondition post in uc
    if a tag t is present for post and not p.satisfies(t)
    then
      remove post
    end
  end
  for each parameter param in uc
    if a tag t is present for param and not p.satisfies(t)
    then
      remove param
```

```
    end
  end
end
return R(p)
```

Algorithm 1. Algorithm to extract the requirements of a product from the product line requirements

12.4 Simulating the Use Cases

In this section, we explain how the enhanced use cases can be simulated for a chosen product, the simulation being the basis of our test generation process.

12.4.1 The Simulation Model

The simulation model is made of:

- Use cases enhanced with parameters and contracts
- The enumeration of all the instances of objects present in the system
- An initial state

Declaring the objects of the system allows to instantiate the use cases: an instantiated use case is a use case whose formal parameters have been replaced by actual parameters. As an example, in the virtual meeting, suppose that we declared 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 their sets of formal parameters by all the possible combinations of their possible specific values.

To begin the simulation, we need an initial state and a simulation state. The simulation state is the current valuation of all the instantiated predicates of the system. In our implementation, the state is represented by a set of true instantiated predicates. The initial state is thus given in terms of instantiated predicates that are valuated to true at the beginning of the simulation. An instantiated use case can be executed or not executed from a given simulation state, depending on its precondition: it can be executed if its precondition is implied by the current state of the simulator. To determine the effects of the execution of an instantiated use case, we use its postcondition: to obtain the new current state, we modify the current state so that the postcondition becomes true. The simulator allows the requirement analyst to visualize at each step of the simulation which actions are valid, i.e., which use cases can be applied with which parameters. The requirement analyst can thus choose one of those actions, which will be simulated, leading the simulation system in a new state.

The benefits of such a simulation are obvious: the requirement analyst can check that the specified product has globally the same behavior like the one he or she had on mind. The simulator also permits to verify properties on the system. For example, one can check that it is not possible to be in a meeting if not connected to the server.

12.4.2 Exhaustive Simulation and Building of a Behavioral Graph

The exhaustive simulation leads to build a behavioral graph. We defined such a graph as a particular labeled transition system called UCTS (Use case Transition System). A UCTS is defined by the quadruple $M = (Q; q_0, \mathrm{A}, \rightarrow)$, where:

- Q is a finite nonempty 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
- $\rightarrow \subseteq Q \times \mathrm{A} \times Q$ is the transition function

A state of the UCTS represents the state of the system (in terms of value of predicates) at different stages of execution. A transition, labeled with an instantiated use case, represents the execution of an instantiated use case. A path in the UCTS is thus a valid sequence of instantiated use cases. A partial UCTS obtained for the demonstration edition is given in Fig. 12.2. 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, where n is the number of instantiated predicates present in the system. In practice, this maximal size is never reached, since all the potential states are not reachable. However, in case of combinatorial explosion of the number of states, the graph is not built exhaustively, but only partially using on-the-fly generation.

For example, in the virtual meeting, if the instantiated predicate *Entered(p1,m1)* is true (meaning that the participant *p1* has entered the meeting *m1*), then necessarily the instantiated predicate *opened(m1)* is also true (meaning that the meeting *m1* is opened). As a consequence, all the potential states for which *entered(p1,m1)* is true and *opened(m1)* is false are not reachable, and thus the actual size of the UCTS is smaller than the maximal size. For the demonstration edition with 3 participants and one meeting, there are 21 instantiated predicates (in fact 9 predicates were used to describe the requirements, which are instantiated into 21 instantiated predicates) and the UCTS has 1616 states whereas its theoretical maximal size is $2^{21} = 2\ 097\ 152$ states.

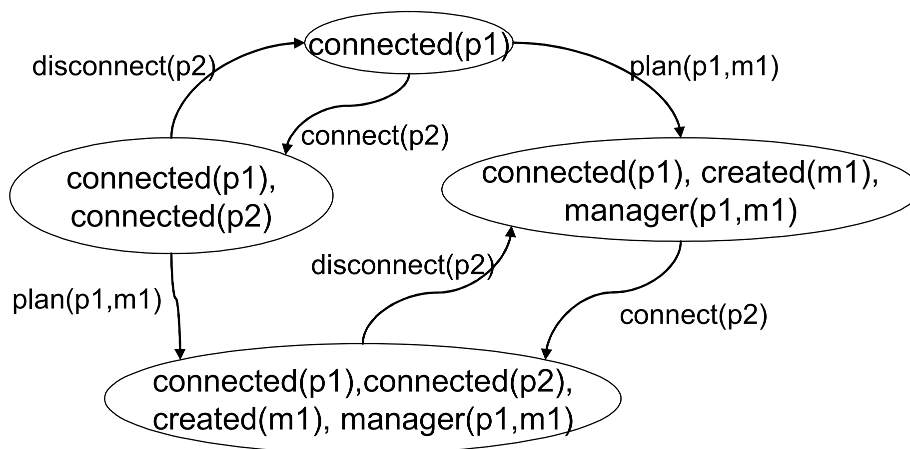

Fig. 12.2. An example of a partial UCTS

12.4.3 Simulating Each Product

The first step to build a UCTS for each specific product is to extract the requirements for each product from the PL requirements. This is simply done by parsing the variation notes in the requirements, and using a decision model, following algorithm 1. Then, for each product-specific requirement, algorithm 2 is applied to build the UCTS. Upon initialization, the initial state is deduced from the initial true predicates. Then the algorithm successively tries to apply each instantiated use case. Applying a use case is possible when its precondition is true with respect to 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.

The simulation of the use case model for each product is the basis of the test generation process. The first step of this test generation process is detailed in the next section.

12.5 Test Objectives

In this section, we explain how test objectives can be generated using the simulation model. Test objectives can be seen as application test specification. We first formalize the notion of valid sequence of instantiated use cases and then we define test objectives from the notion of 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`"0)
  do
    currentState←to_visit.pop
    ∀ uc ∈useCases | currentState ⇒ uc.pre
    do
      newState← apply(currentState, uc)
      if newState ∉result.Q
      then
         result.Q result.Q ∪ {newState}
         to_visit.push(newState)
```

```
      fi
      result.→ ← result.→ ∪{(currentState,uc,newState)}
    done
  done
end
```

Algorithm 2. Algorithm producing the UCTS

Valid sequence of instantiated use cases. A sequence S of instantiated use cases is said to be valid with respect to a system of enhanced use cases UCS if and only if there exists, in the UCTS corresponding to UCS, a path whose sequence of labels is identical to S. A path in the UCTS is here defined as the classical notion of path in a graph.

Test objective. A test objective (TO) is defined here as a valid sequence of instantiated use cases beginning with the root of the UCTS (i.e., the initial state).

Test objectives set consistency with an UCTS. A set of test objectives is said to be consistent with an UCTS iff each TO exercises a path of the UCTS.
When extracting test objectives, we aim at minimizing cost by generating:

- A small number of test objectives. Since a test objective has to be treated (either manually or automatically) to obtain a test case, too many test objectives would lead to having a large test cost.
- Small test objectives, since we believe that they are more understandable than larger ones (the size of a test objective being given in terms of the number of instantiated predicates composing it). For example, when built with a breadth-first algorithm, the height of the UCTS for demonstration edition is 10. We thus believe that the size of the test objectives should be smaller than 10.

In other words, we want to obtain a small number of efficient test objectives, instead of a large number of redundant test objectives. A test objective is redundant with respect to a set S of test objectives if it does not improve the global efficiency of S. The efficiency of the tests is measured here in terms of code coverage.

The two constraints (cost minimization and test efficiency) seem contradictory, but the experimental studies showed that the two criteria defined in the following satisfy these constraints [24]. The efficiency of a test objective can be measured using various criteria (code covered by the corresponding test case, coverage of control graphs, etc.). In [24] and in Sect. 12.7, we have used the code coverage.

All Instantiated Use Cases criterion. (AIUC) A test objective set TOS satisfies the all instantiated use cases coverage criterion for a given use case transition system iff each instantiated use case of the system is exercised by at least one TO from TOS. An instantiated use case is said to exercise a test objective TO iff it is included in it.

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. 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: (true, true), (true, false), and (false, true). The criterion APT will select sequences of use cases so that each use case is applied with all the possible valuations of the expression precondition = true.

These two criteria are not related by a theoretical subsume relationship. To illustrate the APT criterion, suppose that a use case *U(x:X,y:Y)* has the precondition: *p(x) or q(x,y).* Then the APT criterion selects 3 states in the UCTS (*x1* being an instance of type X and *y1* being an instance of type Y):

- One for which the instantiated predicate *p*(x1) is true and the instantiated predicate *q(x1,y1)* is false
- One for which the instantiated predicate *q(x1,y1)* is true and the instantiated predicate *p(x1)* is false
- One for which both instantiated predicates *p(x1)* and *q(x1,y1)* are true

Then a path will be chosen to reach each state, from the initial state of the UCTS. Those 3 paths satisfy the APT criterion.

Other criteria can be used, such as covering all the vertices or all the edges of the UCTS, but they lead either to inefficient tests (all the vertices) or to a very large number of tests (all the transitions) [24].

The two criteria are implemented with a breadth-first search of the UCTS. Such a technique ensures that the obtained TOs 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 understandable than larger ones.

As an example, let us consider again the UCTS of Fig. 11.2, for which we assume that *{connected(p1)}* is the initial state. When applying the AIUC criterion, we will try to exercise the instantiated use case *disconnect(p2).* For that, if we adopt a deep-first search algorithm, we obtain the path *[connect(p2), plan(p1,m1), disconnect(p2)]* (the size is 3). If we apply a breadth-first search, we will first visit all the successors of the initial node (i.e., *{connected(p1), connected(p2)}* and *{connected(p1), created(m1)*, *manager(p1,m1)}*) then explore the successors of those 2 nodes. The path that will then be found is: *[connect(p2), disconnect(p2)*] (the size is 2).

Robustness Testing

The tests generated as described above exercise the application into a nominal way since only expected behaviors are produced from requirements. The system robustness may also be tested since the application should detect the execution of nonexpected use cases in a given test sequence. To generate such robustness tests from enhanced UCs, the contracts must be detailed enough so that all the unspecified behaviors are considered incorrect: 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 nonspecified action. The execution of such a robustness test must lead to a specific treatment (e.g., emitting an error message, raising an exception). If not, a robustness weakness has been detected.

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. This criterion is illustrated in Fig. 12.3.

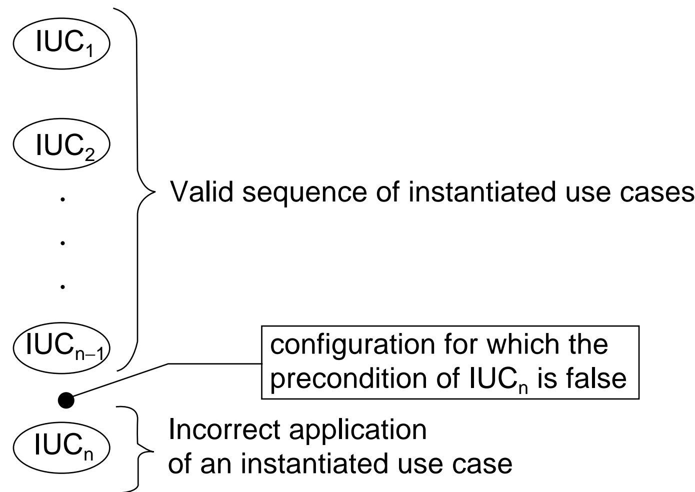

Fig. 12.3. Robustness test objectives

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. By 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 do what it should not.

Specific and General Test Objectives

At this stage, when the sets of test objectives have been generated for each product, the various test objectives are parsed, in order to detect which test objectives are common to the product line, and which test objectives are specific to a given product.

Test Objectives versus Test Cases

In general, the test objectives generated as described above are not executable test cases. Indeed, they are sequence of instantiated use cases and have no links with the implementation of the system. In particular, they do not take into account the interface that the system uses to offer the described services. The following section proposes a method to generate application test cases from test objectives.

12.6 Test Case Generation

Generating test cases that can directly be launched by a test driver requires more information than only the use cases and their contracts. Other modeling elements are needed to make precise the exact interface of the system, i.e., the protocol between the users and the system under test to realize a given use case. In this section, we propose to use particular scenarios to bridge the gap between test objectives (that are at the requirement level) and test cases (that are at the implementation level).We first generate application test scenarios, that are scenarios the tester wants to exhibit. Then we propose to complete those test scenarios in order to obtain test cases, using test synthesis tools. This is done using an intermediate test purpose format named *Behavioral test pattern*.

12.6.1 Generating Test Scenarios

Test scenarios are derived from test objectives using the scenarios attached to each use case: we assume that each use case is documented by its contracts and by system scenarios. We assume that those scenarios are expressed with UML sequence diagrams. Examples of sequence diagrams are given in Figs. 12.4 and 12.5.

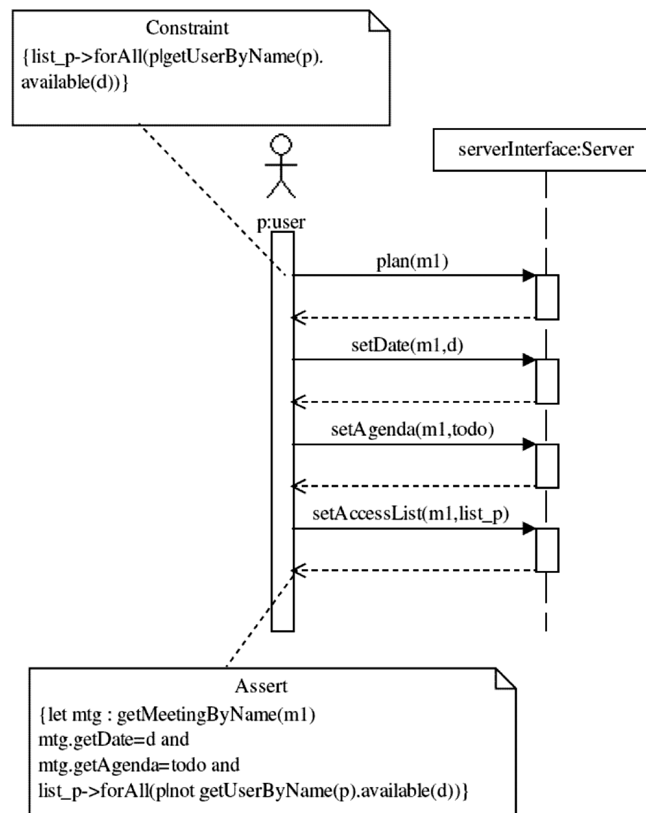

Fig. 12.4. A nominal sequence diagram for the use case plan

Using sequence diagrams. The use of sequence diagrams is interesting for three main reasons:

1. First, sequence diagrams are a way to improve the verdict preciseness. The test objectives built with the contracts method do not embed a precise oracle. The oracle embedded is just the expectation:
   - Of a noninterrupted execution for the functional test objectives
   - Either of an error or of a warning for the robustness test objectives

   Such verdicts are limited since they check neither the system outputs consistency nor any property of the system state. Sequence diagrams are of a lower level of abstraction than the use cases, thus they can embed more precise oracles.
2. Second, sequence diagrams allow to obtain test scenarios from which a code generator can generate the test cases. The test objectives generated are far from the messages exchanged during the test, since they just consist of sequences of parameterized cases. The communication protocols are unknown at this stage. The sequence diagrams attached to the use cases allow us to bridge part of the gap between the test objectives and the test cases, since they describe the expected exchanges of messages between the actors and the system.
3. Third, the scenarios and sequence diagrams are increasingly being used in industry in the early phases of requirements. The conclusion of the survey of industrial software projects [34] published in 1998 insists on the industrial need to base system tests on use cases and scenarios, and explains that most projects lack a systematic approach to define test cases based on scenarios. In [31] published in 2000, the authors still remark that in practice, scenarios from the analysis phase are seldom used to create concrete system test cases. The method presented here makes easier the use of scenarios in the validation phase.

Each of the sequence diagrams we deal with is attached to a use case and represents one of its nominal or exceptional scenarios. Nominal scenarios represent the basic ways to successfully exercise a use case. Exceptional scenarios represent ways to exercise a use case leading to a failure, the raise of an exception, or an error message: exceptional scenarios make the use case fail. The sequence diagrams are system level, in the sense that they only involve the system itself and the actors.

Those sequence diagrams may involve parameters: since they are attached to parameterized cases, it is quite natural to find in the sequence diagrams at least the same parameters as in its owner use case. The sequence diagrams contain more information than the use case, and thus they may own more detailed pre- and postconditions than the use case contracts. As a result, each of those sequence diagrams may own OCL constraints describing on which condition they can be exercised, and what are the consequences on the system.

One can wonder why the OCL is used instead of the UCL. The reason is that, at the use-case level, the contracts are high-level ones, and independent from the static models (class diagrams for example) that will be designed later in the development process. Thus for the use cases, the OCL is not well suited, that is why we defined the UCL. On the contrary, at the sequence diagrams level, we want to design contracts relying on the rest of the

model (on static models for example).We thus need a language to navigate into a UML model, and the OCL is perfectly suited for that. In our context, the nominal scenarios will be used for functional testing and the exceptional ones will be used for robustness testing.

To sum up, the sequence diagrams we deal with are system level, they may involve parameters and they may own additional OCL contracts.

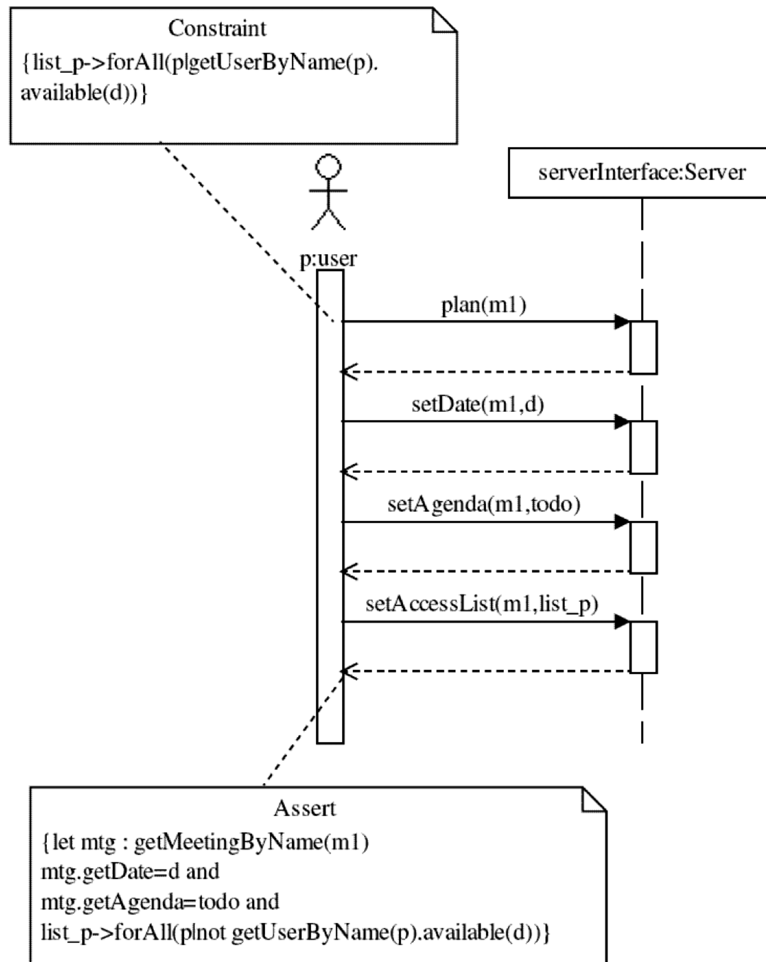

Fig. 12.5. An exceptional sequence diagram for the use case plan

Figures 12.4 and 12.5 provide a nominal and an exceptional scenario for the use case *plan* of the Virtual Meeting system. In the two scenarios, *d* and *list_p* are scenario parameters, which designate the date and the list of the invited participants of the meeting being planned, respectively. The nominal precondition is an OCL precondition that checks whether the invited participants are available at the meeting date. The nominal postcondition checks that the meeting has been planned with the correct parameters. The exceptional scenario checks

that the participants are not available at the meeting date in its precondition, and that the meeting is not planned in its postcondition.

In the product line context, a given sequence diagram can be either common to the product line, or only to a given set of products, depending on the presence of a particular variant. We thus use the same notation as for the use cases to express the variability at the sequence diagram level (see Sect. 12.3.2). Future work will consist in using sequence diagrams directly modeling the variability, such as the sequence diagrams proposed in [35].

Building test scenarios. We propose to replace the instantiated use cases with instantiated scenarios in the test objectives. Sequences of scenarios are thus obtained, and scenario composition is applied on them to obtain a global system test scenario (strong sequential composition is used: strong sequential composition imposes that all the events of a scenario are executed before an event of the next scenario can be executed).

When an instantiated use case is replaced by a scenario, the scenario is partially instantiated using the effective parameters of the instantiated use case. As we already mentioned it, the scenario may also own other parameters; those parameters are not instantiated at this stage. A partially instantiated scenario is thus defined as a scenario whose formal parameters corresponding to the use case parameters are replaced by effective parameters. In the following, this instantiation is supposed to be achieved by the *inst* method.

To define precisely how test scenarios are built, we first introduce the following notations:

- We note $\{scn_{i,j}\}_{j\in 1..n}$ the set of n nominal scenarios attached to the use case uc_i, and $\{sce_{i,j}\}_{j\in 1..m}$ the set of m exceptional scenarios attached to the use case uc_i
- The strong sequential composition of scenarios is denoted by the symbol °.
- The Cartesian product on sets is denoted $\times$.

With those conventions, a test scenario is defined from a tuple of scenarios $(sc_1, \ldots, sc_n)$ as: $sc_1 \circ \ldots \circ sc_n$ (the strong sequential composition the tuple elements). The set of tuples defining a set of test scenarios $TS = \{ts_1, \ldots, ts_u\}$ obtained from a test objective $TO = [iuc_1 \ldots iuc_t]$ is denoted TS_{tuple}. The set TS_{tuple} is obtained applying a Cartesian product on sets of partially instantiated scenarios, as explained in the following definitions.

Functional nominal test scenarios. A nominal test objective $TO = [iuc_1 \ldots iuc_t]$ is transformed into the set of tuples TS_{tuple} defined by:

$$\begin{aligned} TS_{tuple} &= \prod_{i=1}^{t} \{scn_{i,j}.inst(iuc_i)\}_{j\in 1,\ldots,n} \\ &= \{scn_{1,j}.inst(iuc_1)\}_{j\in 1,\ldots,n} \times \ldots \times \{scn_{t,j}.inst(iuc_t)\}_{j\in 1,\ldots,n} \end{aligned}$$

Building the functional test scenarios can be seen as replacing one after the other each of the instantiated use cases of TO by each of its nominal scenarios. Once all the instantiated use cases have been replaced, then a tuple of sequence diagrams is obtained, and strong sequential composition is achieved to obtain a test scenario.

Functional robustness test scenarios. A robustness test objective $TO = [iuc_1 \ldots iuc_t]$ is transformed into the set of tuples TS_{tuple} defined by:

$$TS_{tuple} = \prod_{i=1}^{t-1} \{scn_{i,j}.inst(iuc_i)\}_{j\in 1,\dots,n} \times \{sce_{t,j}\}_{j\in 1,\dots,m}$$
$$= \{scn_{1,j}.inst(iuc_1)\}_{j\in 1,\dots,n} \times \dots \times \{scn_{t-1,j}.inst(iuc_{t-1})\}_{j\in 1,\dots,n}$$
$$\times \{sce_{t,j}.inst(iuc_t)\}_{j\in 1,\dots,m}$$

Building the robustness test scenarios can also be seen as replacing the instantiated use cases by its scenarios. The process to replace the *t–1* first instantiated use cases is the same as for functional test scenarios. The last instantiated use case is each time replaced by one of its exceptional scenarios.

Some test objectives are general for the whole product line, and others are specific to products. During the replacement of the use cases by sequence diagrams, this is taken into account: for product-specific test objectives, only the sequence diagrams corresponding to the particular product have to be taken into account, thus producing specific test scenarios; while for general test objectives, all the scenarios have to be taken into account, thus producing either general test scenarios (when only general sequence diagrams have been used) or specific test scenarios.

The cartesian product of scenarios may lead to a very large number of tests if there are a large number of scenarios per use case. If the test launching is automatic, this is not a problem. If the number of tests has to be reduced, then another strategy has to be applied. Techniques such as the ones proposed in the tobias tool [15] can be used: in the tobias tool, _lters are proposed to reduce the combinatorial explosion of the number of tests generated by combining different test schemas. Filters applied at runtime allow not to run tests with a prefix that have already failed. Such a technique could be used with our approach.

Examples. To illustrate how the test scenarios are built, suppose that we want to generate the test scenarios corresponding to the functional test objective *[connect(p1), plan(p1,m1)]*. We suppose that the use case *connect* is documented by 2 nominal sequence diagrams:

– *SNconnect1* describing a participant asking to connect and then giving her address requested by the system
– *SNconnect2* describing a participant asking to connect giving her address

and that the use case *plan* is documented by the 2 nominal sequence diagrams:

– *SNplan1* describing the planning of a meeting with a name, a date, and an agenda
– *SNplan2* describing the planning of a meeting with just a name and a date

Four functional test scenarios will then be generated: *(SNconnect1, SNplan1)*, *(SNconnect1, SNplan2)*, *(SNconnect2, SNplan1)*, and *(SNconnect2, SNplan2)*. All the combinations of scenarios are thus tested, for example, the one of Fig. 12.6 composing *SNconnect1* with *SNplan2*. In a general case, when the system under test is described by many scenarios, testing all possible combinations of scenarios may lead to a combinatorial explosion: another strategy may consist in executing each (nominal and exceptional) scenario at least once.

If we want to generate robustness test scenarios, only the exceptional sequence diagrams of the use case *plan* will be used. Suppose that we have 3 exceptional sequence

diagrams *SEplan1*, *SEplan2*, and *SEplan3*, we will then generate 6 test scenarios composing the 2 nominal sequence diagrams of the use case *connect* with the 3 exceptional sequence diagrams of the use case *plan*.

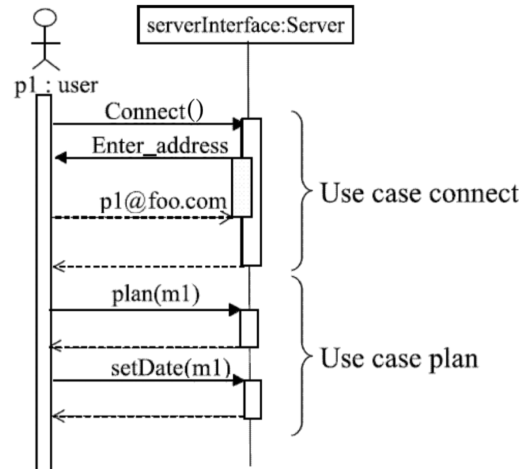

Fig. 12.6. An example of test scenario

Verdicts. The oracle embedded in the test scenarios is built from the OCL pre- and postconditions associated to the sequence diagrams. The test scenarios can emit 3 kinds of verdicts:

- The fail verdict is emitted when a postcondition is violated during the execution. The postconditions ensure that the system is in a correct state after the execution of a sequence diagram. If not, an error is detected.
- The pass verdict is emitted when the test scenario can be executed without error.
- The inconclusive verdict is emitted when a test scenario execution had to be aborted due to a violated precondition. An inconclusive verdict does not mean that an error is detected; it means that the test scenario could not be played. It should be possible to refine each test objective (except for the last use case of a robustness test objective) into a test scenario which satisfies all the preconditions. The fact a test scenario violates a precondition reveals a default in the test objective refinement, when use cases have been replaced by scenarios. An automated approach to generate test cases unhappily may generate such nonrelevant tests. Here we identify them with a distinct verdict, and a manual refinement of the associated test objective into a correct test scenario must be done.

12.6.2 Test Scenarios and Test Cases

The test scenarios may still be incomplete, depending on the sequence diagrams that have been used. The only case when a test scenario can directly be considered as an application test case occurs when the sequence diagrams used exactly contain the messages to exchange

to realize the use case, only using the use case parameters, and without using wildcards (a wildcard is a symbol replacing any expression, the symbol * is often used, see Fig. 12.7).

In a product line, it is very useful to model the sequence diagrams documenting the use cases using parameters and omitting certain parts of the scenario, in order for them to be generic, and to correspond to all the products [25,26]. The other way to proceed is to design specific sequence diagrams for each product, but that leads to several problems: time, maintenance, and so on. For example, in the virtual meeting system, 3 different types of meetings can be planned: democratic, standard and private. However, since the way to plan a meeting is similar for each type of meeting, it can be useful either not to specify the type at all (like in Fig. 12.4) or to replace the type by a wildcard, like in Fig. 12.7.

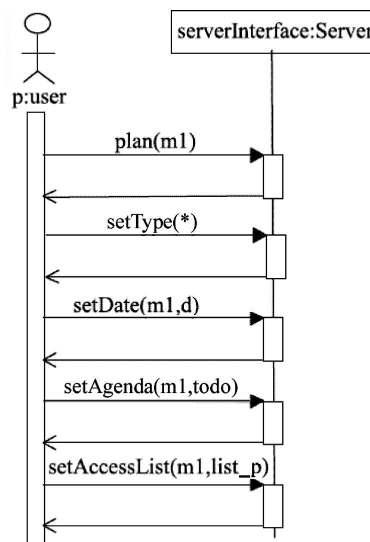

Fig. 12.7. An example of sequence diagram with wildcard

Thus the test scenarios built with the method described above still contain genericity marks: parameters, wildcards, or lack of certain messages or parameters. In the example of Fig. 4, the message setting the meeting type is missing, and in the example of the Fig. 12.7, only the type of the meeting is missing. In order to complete them, we propose to use test synthesis tools.

12.6.3 Test Synthesis Tools

The objective here is not to explain in detail the principles of the test synthesis tools, but to explain why and how they can be used to transform test scenarios into test cases. In short, the principle of the test synthesis tools (such as Agatha [16] or TGV [12]) is to explore the behavioral specification of a system, in order to derive tests from it. We have chosen to use the TGV tool since the exploration is driven by a test purpose: that means that the behavioral specification is parsed until a test case corresponding to the given test purpose is found.

The UMLAUT tool generates a simulation API from the UML model of a system. The way such an API can be built can be found in [29]. This simulation API can then be used by TGV. TGV also needs a test purpose, which has to be given in the form of a labeled transition system (LTS). From the simulation API and the LTS representing the test purpose, TGV builds on the _y the LTS representing the operational semantics of the system and stops this building task when a path in the built operational semantics satisfies the test objective. Such a path is considered to be a test case, and is transformed into a UML sequence diagram.

Ideally, it should be possible to use the test scenarios as test purpose (sequence diagrams can easily be transformed into labeled transition system), in order to obtain test cases using the TGV and UMLAUT tools. The problem with such an approach is the huge size of the LTS representing the operational semantics of each product. In practice, as soon as a real-sized system is studied, if the test purpose is not detailed enough, the part of the LTS that has to be built is far too huge. That is why we propose to use what we call *behavioral test patterns* to guide the test synthesis, instead of just test scenarios. As explained in the following, those behavioral test patterns can be generated from the use cases and the test scenarios.

12.6.4 Using Behavioral Test Patterns

A behavioral test pattern is a test purpose composed of 3 parts, each being given in the form of sequence diagrams:

– The specification of the behavior the test designer wants to test; such a scenario, also called .positive scenario., serves to select the scenarios of the specification which are relevant for the test case.
– The specification of the behaviors the test designer wants to avoid in the test; such scenarios, also called .negative scenarios., serve to eliminate the scenarios of the specification which are irrelevant for the test case.
– The specification of the behavior needed to place the system under test in a state in which the positive scenario can take place; such a scenario, also called .prefix scenario, serves to factorize the part of the positive scenario which may be common to several behavioral test patterns.

The behavioral test patterns are an efficient way to guide the test synthesis. The negative scenarios describe the behaviors which, though correct, are unwanted in the test. Several negative scenarios can be associated to the same behavioral test pattern. They serve to limit the exploration required by the synthesis algorithms in order to find a test case that fits the behavioral test pattern, thereby improving performance. From a pragmatic point of view, if several test executions fit the accept part of the behavioral test pattern, negative scenarios can be used to guide the synthesis tool to produce the most suitable test case. Guiding the tool may be done to help minimize the synthesized test case by excluding calls which are known to be superfluous for the purposes of the test. This reduction of “noise” is particularly useful in testing concurrent applications.

The prefix is a high-level representation of the initialization of the behavior to be tested. It describes the preamble part of the test case, i.e., the behavior previous to that des-

cribed in the positive scenario. The prefix serves to guide the synthesis toward the production of a minimal preamble. Like the negative scenarios, the prefix can be constructed from the other use-case scenarios. Unlike a negative scenario, a prefix may be composed of a sequence of such scenarios. Building the prefix is therefore a process of selecting use-case scenarios and composing them.

To guide the test synthesis, behavioral test patterns are much more efficient than just test scenarios. The behavioral test patterns can automatically be generated from the use cases and the test scenarios, as explained in the following.

Generating behavioral test patterns. A test scenario corresponds to the prefix and the positive scenario of a behavioral test pattern. The test scenario is a composition of various sequence diagrams, the last one representing the positive scenario and the other ones representing the prefix.

The difficulty is thus to generate the negative scenarios. One criterion is to avoid behaviors involving objects which do not interact with the objects involved in the test objective. Suppose that we want to generate the negative scenarios of a behavioral test pattern from a functional test objective [iuc1,...,iucn]. All the instantiated exceptional scenarios of the system will be added as negative scenarios, as well as all the instantiated scenarios handling none of the object handled in the test objective [iuc1,...,iucn].

For example, let us come back to the example of Fig. 12.6. If we want to generate a behavioral test pattern corresponding to this test scenario, we will have as preamble the first part of the test scenario corresponding to the connection, then as positive scenario the second part concerning the planning. Concerning the negative scenarios, we will add all the instantiated scenarios which are not dealing with instances *p1* and *m1* (for example, the planning of *m2*), and all the exceptional scenarios of the other use cases of the system.

To sum up this section, from test objectives, test scenarios are generated using the scenarios attached to each use case. The use case scenarios may include genericity marks (such as parameters and wildcards), thus the test scenarios are still incomplete.

To complete the test scenarios, test synthesis tools can be applied. However, the test synthesis usually fails for large system when the synthesis is not guided by a very detailed test scenario. Thus we propose to guide the test synthesis using particular sets of scenarios called *behavioral test patterns*.

12.7 Results and Discussion

This section offers an experimental validation of the proposed approach: we give an overview of the tests synthesized for the 3 products of our PL example, then we study the efficiency of the tests generated for the demonstration edition. The link from the test scenarios to the test cases (using test synthesis tools and behavioral test patterns) is not yet integrated in our prototype tools, so the experiments we present here are based on the rest of the approach: from use cases to test scenarios.

12.7.1 Test Generated for the 3 Products

From the PL use cases enhanced with contracts, we derived one specific UCTS per product, and then we generated the test scenarios (TS). Statistics are given in Tab. 12.2 (demonstration, personal and enterprise edition are denoted DE, PE, and EE respectively). A study of those test scenarios reveals that common tests have been generated (corresponding to the commonalities of the PL), and specific tests have been generated for each product, due to the different combinations of variants in the products.

Table 12.2. Statistics on generated tests

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

12.7.2 Study of the Generated Test Efficiency for Demonstration Edition

For the experimental validation, we used a Java implementation of the virtual meeting. The virtual meeting example has been built using a common modeling for the whole product line, making use of various well-known design patterns. To perform code coverage studies for the demonstration edition, we performed an ad hoc and manual analysis to distinguish the source code of the product line which was not executed by the demonstration edition, in order to obtain exact coverage figures, which only concerns the code involved in this product. For this given instance of product, around 20% of the code (in terms of executable lines of code) is specific to the product while the remaining is extracted from the common code. This proportion is the same for all of the products.

Moreover, we studied the code of the demonstration edition to evaluate which part of the code is possible to cover with a pure functional and system testing approach. 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 study presented below, 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 with respect to the specification which asserts that only the required functions are present, and robustness with respect to the environment which asserts that the inputs coming from the environment are correct.

The results of the code coverage measures are given in Fig. 12.8. The APT (resp. AIUC) criterion covers 71% (resp. 60%) of the functional code. Note that since the AIUC criterion generates many more TC than the APT one, the APT criterion is more efficient in terms of covered statement per test scenario. Since our robustness tests stem from functional requirements, they cannot cover all the robustness code but they cover 100% of the robustness code with respect to requirements. The uncovered code concerns syntactic verification of the inputs treatment of network exceptions, these aspects are specific to the

distributed platform. Globally, the robustness tests add a 10% code coverage to the functional tests. So, for the parts of code related to functional requirements, half of the robustness code and 98% of the functional one have been covered. The remaining uncovered code is specific to the platform or unused code ("dead code") dedicated to future PL evolution. This result is promising since it reveals that the functional code can be tested from test cases derived from requirement stages. The same kind of approach could be used to generate test cases dedicated to nonfunctional properties, such as security and real time.

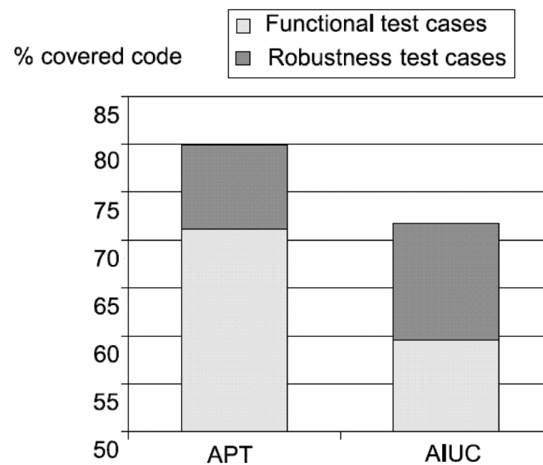

Fig. 12.8. Code coverage of the tests

The ratio between the number of robustness tests and the corresponding coverage is deceptive. Improving the robustness test efficiency would require defining more efficient test generation criteria and more detailed scenarios (however requiring such detailed diagrams from the designers has a heavy cost).

This study shows that the tests generated from the product requirements expressed at the PL level (and extracted for a specific product using a decision model) are relevant at the product code level, with the use of adequate criteria. However, to get higher confidence in these encouraging results, future work will consist in evaluating the approach with other case studies, and other efficiency criteria (code coverage is a weak criterion, better criteria are branch coverage or mutation score for example). Other experiments also showed that classical faults – using mutation analysis – manually injected in the products were detected by our tests. The approach has also been successfully applied on two systems components of last generation combat aircrafts (Mirage 2000-9 and Rafale), of mid-complexity (several thousands C++ KLOC). These real-case studies are not designed in a product-line context but reveal that approximately 80% of the functional requirements could be treated and used for test generation. This experience return shows the relevance of the approach for functional requirements, since the 20% nontreated requirements were related to detailed design features and did not describe services requirements.

12.7.3 Discussion on the Benefits and Limitations of the Approach

As several other approaches to test product lines (and in particular the approach presented in Chap. 11), we assume that a common requirement model is available. In Chap. 11, this requirement model is made of PLUCs (Product Lines Use Cases), while in our approach it is made of use cases with contracts, parameters, and sequence diagrams. Our approach is not dependent on the way UML models are obtained: they can be obtained using model transformations on a common model for the whole PL or manually built. In this sense, our approach fits into the overall process of product line engineering, since it only requires a common requirement engineering phase.

The automation of the approach can be discussed. Globally, to use our approach, a tester has to:

- Take the use cases, sequence diagrams, the decision model, and the UML model of the considered product
- Manually define the instances/objects the tests have to deal with (at the requirement level and at the code level)

The quality of the obtained tests strongly depends on the quality of the inputs, which come from the specification. This is a classical problem for testing, since tests are always generated to validate an implementation with respect to a specification. If the quality of the specification is low, test cases will only test a little part. Improving the quality of the input models is beyond the scope of this chapter. However, robustness test cases may help reveal lack of precision in the specification, since robustness test cases aim at exploring the bounds of the possible behaviors. This analysis is manual but may help identifying defaults in the specification. Concerning the artifacts the tester has to manipulate, if the inputs (mainly the use cases and the sequence diagrams) are detailed enough, the generated tests will be efficient enough, and the manual task of the tester will not be important: it will simply consist in the verdict analysis emitted by the tests. However, if those inputs are not detailed enough, some tests may be missing to satisfy a chosen coverage criteria. Classical unit testing must be done to complete the test. The main advantage of the approach is to get confidence in the end-product implementation with respect to the functional product requirements, even in the case these requirements are not complete enough to cover the whole code. A consequence of the approach is to identify – by measuring the actual test coverage obtained for requirement-based tests – the lack of precision in the requirements and analysis views.

Concerning the adequacy of the approach to the PL context, when new requirements are added, a brutal approach consists in regenerating automatically all the test cases.

However, the test generation tool allows a guided test cases generation. For instance, only tests cases that exercise the new added property, parameter, or use case can be generated. To ensure some regression testing, it is highly recommended to reapply the existing test cases when testing the newly added features. The approach is thus adaptable and allows both to generate again test cases and to generate test cases that exercise a chosen requirement. Thus, the process is either incremental for an underevolution product or allows a full-test generation when a new product is created. As explained in the case study, the approach does not allow nonfunctional test case generation from requirement. We believe

that an analogous approach may be applied for some specific nonfunctional properties, such as execution time and security testing.

12.7.4 Related Work

The PL engineering now appears as a major issue in the field of software engineering; however, PL validation is not yet mature, and in particular PL testing is not studied enough in comparison with the large set of new issues implied by PL testing. However, judging by the test generation approaches briefly presented in the SPLIT workshop [32] (e.g., use of mutation techniques and formal methods), PL testing is undergoing a resurgence of interest.

The PL testing issues and challenges are described in [22]. They are also evoked in [18], which gives an overview of the product line testing. McGregor describes the whole PL testing process, in particular, all the different test artifacts that have to be produced are described, as well as the process from which they are produced and the related PL specificities. The main contribution concerning the testing process comes from references [6] and [33].

Concerning methodological and technical PL testing approaches, from our point of view the two main approaches are [2] and [14,30]. Details on those approaches can be found in Chaps. 11 and 13.

In Chap. 11, the authors have adapted the well-known Category-Partition (CP) method in their PLUTO approach. The CP method is applied at the use-case level, and more precisely at the PLUC level. The PLUCs mechanism to manage variability and ours are quite similar. However, the underlying testing method is different in the sense that the approach proposed in Chap. 11 focuses more on test data. We thus believe that for applications for which the handled data are more complex than the control, the PLUTO approach is better-suited than ours. On the contrary, for applications with complex control, our approach is better suited. As previously explained in this chapter, one of the weaknesses of our approach is that the test data have to be manually managed by the tester. We thus believe that our approach would benefit from a coupling with the PLUTO approach. The PLUTO approach could for example be used to generate adequate test data to feed our approach.

In Chap. 13, the authors propose a testing method relying on different test strategies, depending on the ways variability appears in the use cases. Four strategies are identified: abstraction, parameterization, segmentation, and fragmentation. The most adequate strategies are discussed depending on the type of variability that can appear in the event flow, the pre- and postconditions, the actors, and the relationships. This approach is systematic, but yet not automatic. Our approach would benefit from using parts of the ScenTED approach in several ways. A first obvious point is that we focus on functional system testing whereas ScenTED covers other kinds of testing such as integration testing. Second, ScenTED introduces an enhancement of activity diagrams such that activity diagrams can embed variability information. Such activity diagrams could be used in our approach instead of sequence diagrams, or (better) complementary to sequence diagrams.

12.8 Conclusions and Future Research

The testing task is known to be an important part of software development and usually suffers of time-to-market constraints leading to reduce the time dedicated to the validation of the system. The problem of the testing cost is all the more crucial in the product line context since it is not a single system that has to be validated, but several (and potentially a large number of) systems of the same product line. That is why the automation of the testing task is a challenging issue in the field of product line validation.

We have presented a complete chain for functional test cases derivation from the functional requirements of a product line. Avoiding testing all possible combinations of products (most of them being never instantiated in practice), the approach targets a given product in the product line, extracts its functional requirements using the decision model, and generates test cases from these requirements. Requirements, expressed by used cases, are improved by declarative information under the form of contracts as an anchor for further testability purposes and to express variability and commonality.

At requirement stage, the analyst may check the consistency of each product's requirements using the UCTS as a simulation platform. The test cases are generated in two steps: correct sequences of use cases are deduced from use case contracts and then scenarios are substituted to each use case to produce a test scenario that is finally transformed into a test case thanks to test synthesis tools. One of the principal objectives of this approach is the possibility to use it in an industrial context. For that, instead of pushing formal methods to the industry (one of the motto in this community) we proposed to work the other way round, i.e. starting from established practices and gently pushing them towards formally exploitable models. We concentrated here on widely accepted practices based the use of the UML to support an object-oriented development process. The industrial feasibility of the approach has been validated for a single product in the context of the Carroll project, with the industrial partner Thalès [17] and using academic case studies for the product-lines aspects.

In this context, the approach we presented partially automates the generation of product-specific system test cases from Use Cases, taking into account traceability problems between high-level views and concrete test case execution. Due to the automation, the approach is adaptable to several product-line evolution processes. Indeed, it supports full-test generation when a new product is added to the product line as well as partial generation of dedicated test cases when new features are added to an existing product.

Several future research directions can be explored to improve our approach. The first step consists of studying the different ways for users to enter the models of use case dependencies. As mentioned in the previous section, other approaches propose graphical notations and, in particular, UML activity diagrams to enter such models. It is thus worth studying precisely the exact expressiveness of the two languages (i.e., activity diagrams versus contracts) and detecting in which situations one language is better-suited than the other. Then, compatibility rules between the languages can be detected and transformations from one language to another can be envisioned. The second step is to focus on test

data. Currently, our approach needs to be manually fed with the test data for the test generation and the simulation. Since the existing research work in the field principally aims at generating relevant test data for product line testing, future research can couple our approach with the existing work concerning test data generation and experiment the efficiency of the generated tests. Finally, our approach needs to be validated with real-world case studies.

Acknowledgments

We gratefully acknowledge the extensive reviews of Antonia Bertolino, Erik Kamsties, Timo Käkölä, Andreas Metzger, and Antti Tevanlinna, which significantly improved the quality of this chapter.

References

1. Bertolino, A., Fantechi, A., Gnesi, A., Lami, G., Maccari, A.: Use case description of requirements for product lines. In: Proceedings of the International Workshop on Requirements Engineering for Product Lines (2002) pp 12–19
2. Bertolino, A., Gnesi, S.: Use case-based testing of product lines. In: Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (2003) pp 355–358
3. Binder, R.V.: *Testing Object-Oriented Systems* (Addison-Wesley, Reading, MA 2000) Chapter 8
4. Briand, L., Labiche, Y.: A UML-based approach to system testing. J. Softw. Syst. Model. 10–42 (2002)
5. Bühne, S., Halmans, G., Pohl, K.: Modelling dependencies between variation points in use case diagrams. In: Proceedings of the 9th International Workshop on Requirements Engineering: Foundation For Software Quality – REFSQ'03 (2003)
6. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns* (Addison-Wesley, Reading, MA 2001)
7. Cockburn, A.: Structuring use cases with goals. J. Object-Oriented Program. 35–40 and 56–62 (Sept/Oct and Nov/Dec 1997)
8. D'Souza, D.F., Wills, A.C.: *Objects, Component, and Frameworks with UML: The Catalysis Approach, Chapter Interaction Models: Uses Cases, Actions, and Collaborations* (Addison-Wesley, Reading, MA 1999)
9. Fröhlich, P., Link, J.: Automated test case generation from dynamic models. In: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00) (2000)
10. Gomaa, H., Shin, M.E.: Multiple-view meta-modeling of software product lines. In: Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (2002) 238–246
11. Halmans, G., Pohl, K.: Communicating the variability of a software-product family to customers. Softw. Syst. Model. **2**(1), 15–36 (2003)
12. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: Proceedings of the 6th World Conference on Integrated Design and Process Technology (2002)
13. John, I., Muthig, D.: Product line modeling with generic use cases. In: Proceedings of SPLC2 Workshop on Techniques for Exploiting Commonality Through Variability Management (2002)
14. Kamsties, E., Pohl, K., Reis, S., Reuys, A.: Testing variabilities in use case models. In: Proceedings of the Fifth Workshop on Product Family Engineering. Lecture Notes in Computer Science, vol 3014 (Springer, Berlin Heidelberg New York 2003)
15. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering tobias combinatorial test suites. In: Proceedings of ETAPS/FASE'04 – Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol 2984 (Springer, Berlin Heidelberg New York 2004)
16. Lugato, D. et al: Validation and automatic test generation on UML models: the AGATHA approach. Electron. Notes Theor. Comput. Sci. **66**(2) (2002)
17. Lugato, D., Maraux, F., Le Traon, Y., Normand, V., Gallois, J.P., Dubois, H., Pierron, J.Y., Nebut, C.: Automated functional test case synthesis from Thales industrial requirements. In: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (2004)

18. McGregor, J.D.: Testing a software product line. Technical report, CMU/SEI (2001)
19. McGregor, J.D.: Building reusable test assets for a product line. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools (Springer, Berlin Heidelberg New York 2002) pp 345–346
20. McGregor, J.D., Sykes, D.A.: *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley, Reading, MA 2001)
21. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
22. Muccini, H., van der Hoek, A.: Towards testing product line architectures. In: Proceedings of the ETAPS03 Workshop "Test and Analysis of Component Based Systems" ("TACOS'03"), vol 82 (2003)
23. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.M.: A requirement-based approach to test product families. In: Proceedings of the 5th Workshop on Product Families Engineering (PFE-05). Lecture Notes in Computer Science (Springer, Berlin Heidelberg New York 2003)
24. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.M.: Requirements by contracts allow automated system testing. In: Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'03) (2003)
25. Nebut, C., Pickin, S., Le Traon, Y., Jézéquel, J.M.: Reusable test requirements for UML-modeled product lines. In: Proceedings of the Workshop REPL'02 (Requirements Engineering for Product Lines) (2002)
26. Nebut, C., Pickin, S., Le Traon, Y., Jézéquel, J.M.: Automated requirements-based generation of test cases for product families. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03) (2003)
27. OMG: OCL. http://www.omg.org/docs/ptc/03-08-08.pdf (2003)
28. OMG: Unified modeling language specification, version 2.0. http://www.omg.org/docs/formal/03-03-01.pdf (2004)
29. Pickin, S., Jard, C., Le Traon, Y., Jéron, T., Jézéquel, J.M., Le Guennec, A.: System test synthesis from UML models of distributed software. In: Proceedings of the 22nd Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02), Houston, Texas (2002)
30. Reuys, A., Kamsties, E., Pohl, K., Reis, S.: Model-based system testing of software product families. In: Proceedings of the 17th Conference on Advanced Information Systems Engineering (CaiSE'05) (2005)
31. Ryser, J., Glinz, M.: Scent – a method employing scenarios to systematically derive test cases for system test. Technical report (Institut für Informatik, University of Zurich 2000)
32. Proceedings of the International Workshop on Software Product Line Testing (2004)
33. Tevanlinna, A. et al: Product family testing: a survey. SIGSOFT Softw. Eng. Notes **29**(2) (2004)
34. Weidenhaupt, K. et al: Scenario usage in system development: A report on current practice. IEEE Softw. (1998)
35. Ziadi, T., Hélouet, L., Jézéquel, J.M.: Behaviors generation from product lines requirements. In: Proceedings UML2004 Workshop on Software Architecture Description (2004)