

Reusable Test Requirements for UML-Modeled Product Lines¹

Clémentine Nebut, Simon Pickin, Yves Le Traon and Jean-Marc Jézéquel
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Clementine.Nebut, Simon.Pickin, Yves.Le_Traon, Jean-Marc.Jezequel}@irisa.fr

Abstract

The object paradigm is increasingly being used in the construction of both centralized and distributed systems and is a key aspect of the current trend for model-driven architectures. In this paper, we present an approach to natural expression of test requirements and to formal validation in a UML-based development process which takes advantage of product lines (PL) specificities. We proceed by building behavioral test patterns (i.e. the test requirements) as combinations of use-case scenarios, these scenarios being product-independent and therefore constituting reusable PL assets. We then present a method for automated synthesis of test cases for specific products from these product-independent behavioral test patterns and product-specific design models, remaining entirely within the UML framework. We illustrate our approach using a virtual meeting PL case study.

1 Introduction

In the software field, object-orientation and component models are the main enabling technologies underpinning a product-line driven approach. In the development of a product line [1,2], the initial definition of the individual products depends mainly on marketing considerations. At this stage, functional requirements may be defined, in terms of use-cases and coarse scenarios. These use-cases/scenarios offer the natural basis for specifying test requirements. In our method, use-case scenarios (nominal and exceptional) are used to specify *behavioral test patterns* that then become reusable “test assets” of the product line. Use-case scenarios cannot be used directly for testing because they are generic and incomplete. That is, they are independent of the low-level design, which is specific to a particular product. They specify the general system functionality but not the exact way of exercising this functionality in terms of sequences of calls and responses. The complete sequence of method calls, the exact values of parameters in the method calls and even the exact types of the participating instances may not be known at this stage.

The main contribution of this paper is a method, supported by the Umlaut/TGV tool[3,4], for automatic generation of detailed test cases associated to specific products, from sets of incomplete and generic scenarios associated to a product line, i.e. *behavioral test patterns*. Of prime importance are the generic scenarios representing the test requirements (main expected behavior): the use-case scenarios. In this way, the method relates the principal test cases to the use-cases, contributing to the overall coherence of the development process.

We propose a two-step process, from test requirements to product-specific test cases. In step 1, test requirements are defined independently of the target final-product, in terms of behavioral test patterns. From use cases, we show how scenarios can be structured to produce reusable test patterns common to an entire product line and represented using the UML [5]. In step 2, when the final design is available and a product chosen as test target, we synthesize test cases for that particular product.

In Section 2, the issues of testing product lines are presented, as well as the testing requirements we propose, and our associated method dealing with behavioral test patterns. In Section 3, the test pattern methodology is detailed and exemplified.

2 Testing product lines : issues, requirements, method

We focus on exploiting test requirements, expressed as a combination of use cases scenarios. The product line testing we deal with in this paper is black-box testing: it is well-suited to dealing with variability in PLs. In this

¹ This work has been partially supported by the CAFE European project. Eureka Σ! 2023 Programme, ITEA project ip 0004 and the COTE project of the French RNTL program.

section, we present the issues involved in testing product lines, the test patterns we propose to use as testing requirements, and the underlying test methodology.

The philosophy for building product lines is build for reuse: from the first design of the PL, components are conceived, modeled and implemented to be reusable. This reuse philosophy can then be extended to the testing requirements of PLs. While the OO testing is becoming an important research domain[6, 7], to our knowledge, no study has been conducted dealing with the question of reusable, product-independent test assets in PLs, nor with the question of automatically synthesizing product-specific test cases from these product-independent assets.

As underlined in [8], product family requirements assist design evaluation. The idea we develop here is that the requirements and tests designers can take advantage of the presence of a common core, and consequently of a common behavior of products, to reuse both the tests and the requirements. In fact, we believe that the tests should be defined to be as independent as possible of the variants. At first sight, there is a contradiction in this approach:

- on the one hand, to be reusable, test scenarios must be expressed at a very high level to ensure that they are not dependent on the variants and on specific products,
- on the other hand, generic scenarios are too vague and incomplete to be directly used on a specific product, so that reuse of generic test scenarios does not seem possible.

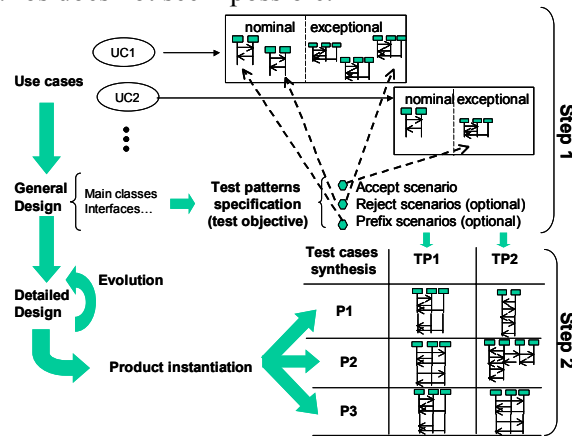


Figure 1. The PL testing methodology

This paper deals with this apparent contradiction by automatically deriving test cases dedicated to a specific product from a (set of) generic scenario(s). The methodology is illustrated in Figure 1. At the analysis stage, use cases are defined and sequence diagrams are attached to each of them. The sequence diagrams are of two types: those describing nominal scenarios, expressing the standard use-case occurrence, and those describing exceptional ones. The testing requirements are defined based on these scenarios in terms of *behavioral test patterns*.

A behavioral test pattern is here defined as a set of generic scenario structures representing a high-level view of some scenarios which the system under test (SUT) may engage in, according to its specification, and which we wish to test. The genericity may involve abstraction on instance names, method names or on method parameters. The scenario structures are elaborated with the aim of using them as selection criteria to derive a test case; they will usually include a subset of the communications of the corresponding test case. We suppose that in the derived test cases, the SUT environment role will be played by the tester. A test case derived from a test pattern specifies how to stimulate the SUT via the test interface, as well as specifying the expected responses at this interface, in such a way as to cause a conformant implementation to execute a scenario which fits the test pattern.

In our method, a test pattern comprises three parts, each part specified using sequence diagrams:

- the specification of the (visible or internal) behavior the test designer wants to test; the *accept* scenario structure serves to select the scenarios of the specification which are relevant for the test case;
- the specification of the (visible or internal) behavior the test designer wants to avoid in the test; the *reject* scenario structures serve to eliminate the scenarios of the specification which are irrelevant for the test case;

- the specification of the behavior needed to place the SUT in a state in which the accept behavior can take place; the *prefix* scenario serves to factorize the part of the accept scenario which may be common to several test patterns.

To specify a test pattern, the tester selects from among the nominal and exceptional scenarios an *accept scenario* describing the behavior that has to be tested, one or several (optional) *reject scenarios* describing the behaviors that are not significant for the test and one or several (optional) *prefix scenarios* that must precede the accept scenario.

Independently of the test development process, the PL design is developed and the final products instantiated. Further evolutions can also be carried out without changing the test patterns library, which is considered an asset of the PL. The most important aspect of our method is that the test patterns are defined once, and then kept unchanged and reused throughout the software life-cycle even if the code or the model evolve : a test pattern is a requirement for a product line, which is reusable for all the products of the PL. Test cases are derived for each specific product, taking into account all the choices and selections made during the product instantiation. For each product and for each test pattern, one or several test cases are synthesized.

The building of test patterns as requirements for testing (step 1) is explained and illustrated in the next section, and an overview is given on the derivation of test cases from test patterns (step 2).

3 Behavioral test patterns as requirements for testing product lines

3.1 The virtual meeting server example

The virtual meeting server PL offers simplified web conference services, that is, it permits several different kinds of work meetings on a distributed platform (a kind of generalized “chat” program).

When connected to the server, a user can enter or exit a meeting, speak, or plan new meetings. The use case diagram is given in Figure 2.

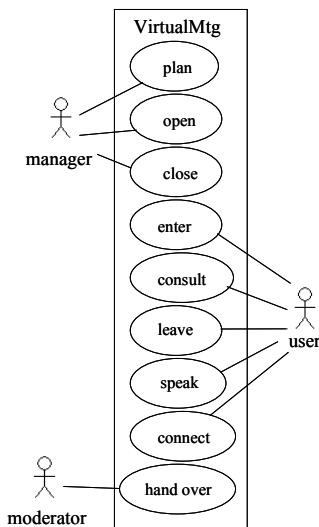


Figure 2 - Use case diagram for the virtual meeting

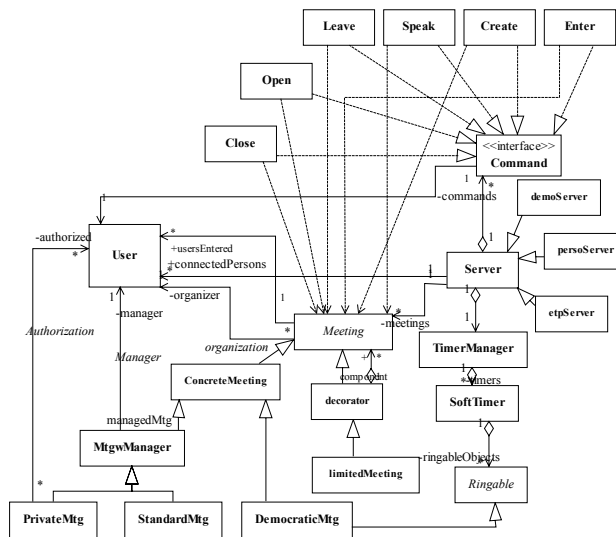


Figure 3 - Class diagram for the virtual meeting PL

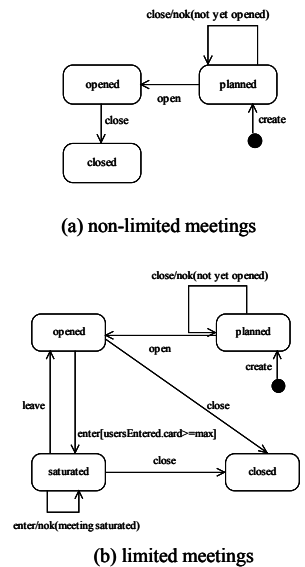


Figure 4 - Virtual meetings state machines

Three types of meetings exist:

- *standard* meetings where the current speaker is designated by a moderator (nominated by the organizer of the meeting)
- *democratic* meetings which are standard meetings where 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.

Marketing considerations mean that three products are to be derived in the virtual meeting PL : a demonstration edition, a personal edition, and an enterprise edition. The demonstration edition manages only standard meetings, limited to a certain number of participants. The personal edition provides the three kinds of meetings but still limits the number of participants of the meetings. The enterprise edition limits neither the type of meeting nor the number of participants in a meeting.

Two main variants are identified in the virtual meeting PL: firstly, the limitation of the number of participants in meetings, and secondly, the type of meetings available.

Figure 3 gives the partial class diagram for the Virtual Meeting PL. All the possible commands are reified and inherit the **COMMAND** interface. Only the main commands appear in Figure 3 to keep the diagram readable. To model behaviors, state diagrams are attached to the main classes that represent the users and the meetings. A state diagram is attached to the root of the concrete meetings hierarchy *concreteMeeting* and to the decorator *limitedMeeting*. Several design patterns are used but they are not described here, for the sake of conciseness.

3.2 Step 1: From use-case scenarios to behavioural test patterns

Building a test pattern consists in selecting – from among the high level scenarios – the behavior that the test designer wants to test, the behaviors that are irrelevant for testing it and, possibly, a prefix. This can be done as soon as the main classes of the product lines are defined: no design detail or state machines are used at this stage. This allow the test patterns to be reusable, because they are independent of the detailed design and the implementation.

The use case scenarios

The use-case scenarios employed to construct the test patterns are represented as generic sequence diagrams. They do not specify the exact parameter values of the methods, nor the exact name and type of the objects. They specify only general behaviors associated to a use case, and are defined very early in the conception (in the analysis phase). Wildcards and scenario parameters are used to achieve the genericity. As an example, in Figure 5 we give the scenarios associated to the *enter* use case. They are quite trivial: a user trying to enter a meeting can either be accepted or rejected.

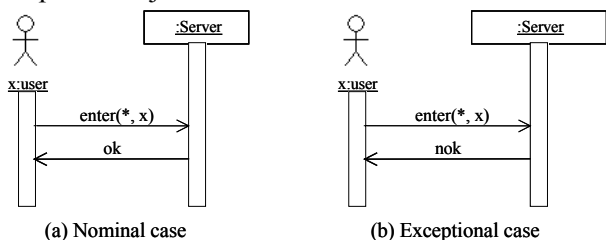


Figure 5 - Enter use case scenarios (x is a scenario parameter)

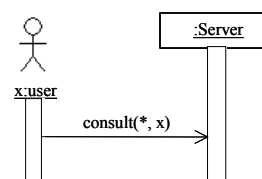


Figure 6 - A reject scenario for the *enter* use case

Reject scenarios

The reject scenarios describe the behaviors which, though correct, are unwanted in the test. Several reject scenarios can be associated to the same test pattern. They serve to limit the exploration required by the synthesis algorithms in order to find a test case that fits the test pattern, thereby improving performance. From a pragmatic point of view, if several test executions fit the accept part of the test pattern, reject 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. It may also be to exclude calls which are known to interfere with the test. We now give examples of these two cases.

To illustrate the case of superfluous calls, we consider deriving a test case from the nominal *enter* use case in our virtual meeting example. Any user can at any moment consult the state of the different meetings managed by the server. Calling the *consult* method is not relevant in testing of the functionality of the *enter* use case. The presence of a call to the *consult* method can be avoided in a synthesized test case by adding the reject scenario given in Figure 6 to the test pattern. This then means that the tester (taking the place of the system environment, ie. the users) will not call the *consult* method, at any moment, with any parameter values, during the test of the *enter* use case.

Note that this reject scenario corresponds to both the nominal and the exceptional scenarios attached to the use case *consult*, illustrating the reuse of scenarios in different test patterns. The use-case scenarios are defined once, and are then used either as reject or accept scenarios depending on which use case is being tested. In other words, once basic scenarios are defined, they are reused as building blocks to build test patterns.

To illustrate the case of interfering calls, we consider deriving a test case for the exceptional *enter* use-case scenario in our virtual meeting example. If the test designer wishes to check that an attempt to enter a meeting which is full will be refused, using the accept scenario as a test pattern is not sufficient. This is since the tool may instead produce a test in which the meeting is closed before the attempt takes place. Using the nominal scenario attached to the *close* use-case as a reject scenario will ensure that this does not occur.

Consider the case where the scenario associated to use case *i* is to be used as the accept scenario. Clearly, in this case, the scenarios associated to use cases which are independent of *i* can be used as reject scenarios. The task of adding reject scenarios can thus be partially automated, as soon as the designer of the PL has defined the dependencies between use cases. The dependencies are represented using a square boolean matrix of dimension the number of use cases, where $Dep[i,j]=true$ if and only if use case *i* depends on use case *j*. When testing a particular use case *i*, all the scenarios attached to use cases *j* such that $Dep[i,j]=false$ are used as reject scenarios. Such a matrix can be built in the early phases of conception.

Prefixes

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 described in the accept scenario. The prefix serves to guide the synthesis towards the production of a minimal preamble. Like the reject scenarios, the prefix can be constructed from the other use-case scenarios. Unlike a reject 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 (by weak sequential composition).

Selection of the different parts of a test pattern

Once use-case scenarios have been built and placed in a test library, building a test pattern is a matter of selecting the accept scenario, the unwanted behaviors, and the prefix scenarios from this library. Figure 7 shows a test pattern for the virtual meeting example.

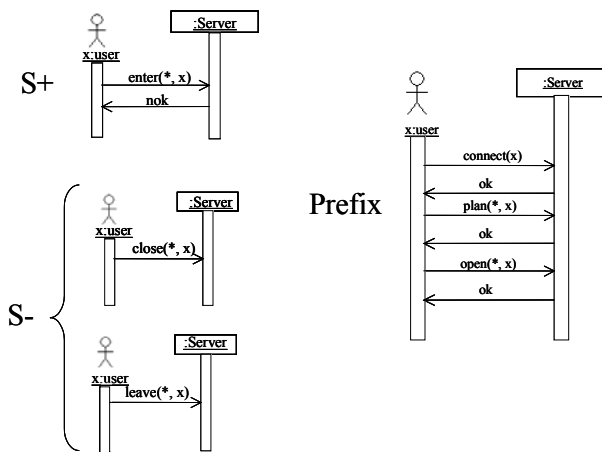


Figure 7 - A test pattern to check that an attempt to enter a meeting which is full is unsuccessful

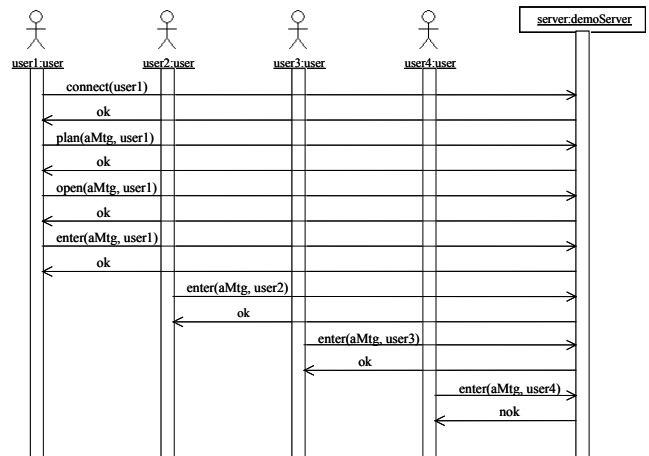


Figure 8 - A test case for the saturated meetings

The behavior to be tested, as described by the accept scenario, is: “an attempt by a user to enter a meeting is rejected”. The unwanted behaviors are any closing or leaving action. The former since we do not want to synthesize a test in which an attempt to enter a meeting is rejected due to the meeting having been closed. The latter since we wish to synthesize a test in which other users may enter but not leave the meeting. Note that the *consult*, *hand over* and *speak* scenarios could also have been (automatically) added as reject scenarios. The prefix describes how a user

connects to the system, plans a meeting and then opens it. Of course, the opening action could be moved from the end of the prefix to the beginning of the accept scenario.

The test synthesis from a test pattern and a UML specification is summarized in the next following.

3.3 Step 2: The test synthesis from behavioral test patterns

Test patterns are independent of low-level design and implementation choices. While defining a high-level test scenario is not difficult when the main classes are identified, refining and adapting it to the final software product is an arduous process. However, the details of the low-level design are contained in the UML specification; completing the test pattern with these details is therefore a task which can be left to the automatic synthesis. Allowing test designers to work at the test pattern level rather than the test case level thus frees them from the need to specify the low-level detail, enabling them to concentrate on the essential aspects of the test.

Our process to generate the concrete test cases from the test requirements is based on the TGV generation tool [4]. The aim is not here to detail the whole process of the test synthesis. In short, the test pattern is first pre-compiled in order to treat the genericity, then compiled into a Labeled Transition System (LTS) [4]. A simulation API is built from the UML specifications, in order to be able to build lazily the LTS representing the operational semantics of the entire system. From those two LTSs and a set of other information such as the definition of the inputs and outputs, TGV builds an Input-Output LTS (IOLTS) representing the test case. This IOLTS is then transformed into a sequence diagram corresponding to the synthesized test case.

Figure 8 is an example of test case synthesized from the test pattern of Figure 7 and the final UML model whose simplified class diagram is given in Figure 3 and two of whose the state diagrams are given in Figure 4. The multi-instance representation of the tester (as several users) requires certain hypotheses concerning the concurrency.

4 Conclusion

In this paper, we have presented a methodology for testing product lines, from a natural way of specifying test requirements to the generation of concrete test cases. We have defined behavioral test patterns as reusable (and incomplete) testing scenarios expressed in UML, and explained how to build them from the use-case scenarios. We have also presented a methodology and tools for the automated production of test cases from these test patterns and a UML specification of the system under test.

The method, using prototype tools like Umlaut and TGV, is not yet completely tool-supported, but further work is underway to fully automate it.

From a methodological point of view, the contribution of this paper is to propose a whole process, from early modeling of requirements to test cases. This process allows concrete test cases to be produced for each specific product from reusable and generic test patterns and UML specifications

5 References

1. Atkinson, C., et al., *Component-based Product Line Engineering with UML*. Component Software Series, ed. C. Szyperski. Addison-Wesley, 2002.
2. Bosch, J., *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.
3. UMLAUT, *Unified Modeling Language All pUrposes Transformer*. 2002.
4. Jard, C. and T.Jéron. *TGV: theory, principles and algorithms*. in *6th world conference on integrated design and process technology*. 2002.
5. OMG, *Unified Modelling Language Specification, version 1.4. OMG Standard*, in *Object Management Group*. 2001.
6. McGregor, J.D. and D.A. Sykes, *A practical guide to testing object-oriented software*. Addison Wesley ed, 2001.
7. McGregor, J.D., *Test patterns: please stand by*. JOOP, 1999. vol. 12 p. 14-19.
8. Lutz, R.R., *Extending the product family approach to support safe reuse*. *Journal of systems and Software*, 2000. vol. 53 p. 207-217.