

# Automated Requirements-based Generation of Test Cases for Product Families

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

*Software product families (PF) are becoming one of the key challenges of software engineering. Despite recent interest in this area, the extent to which the close relationship between PF and requirements engineering is exploited to guide the V&V tasks is still limited. In particular, PF processes generally lack support for generating test cases from requirements. In this paper, we propose a requirements-based approach to functional testing of product lines, based on a formal test generation tool. Here, we outline how product-specific test cases can be automatically generated from PF functional requirements expressed in UML. We study the efficiency of the generated test cases on a case study.*<sup>1</sup>

## 1 Introduction

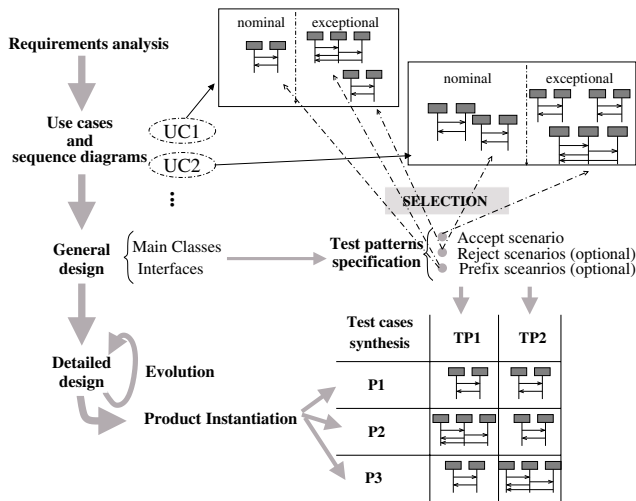
Product families (or product lines) can be defined as “a set of products that share a set of requirements and significant variability”[2], and are currently undergoing a resurgence of interest. PF conception and design brings up a large number of novel issues, among them, PF testing methods. PF requirements play a crucial role in driving the functional testing task. However, there is a real difficulty in ensuring that the requirements of the PF are satisfied on all the products. In other words, *requirements-based testing* of a PF is an arduous task, which is in serious need of automation. Among the numerous test case generation techniques that can be found in the literature, few of them have been adapted to the PF context. The survey of [5] gives a detailed state of the art of PF testing and shows that until now, most test case generation has been done manually, the only test automation being in the generation of test scripts

<sup>1</sup>This work has been partially supported by the CAFE European project. Eureka S! 2023 Program, ITEA project ip 0004.

or test harnesses. Testing a PF is all the more tedious since all the common and the shared variant requirements have to be tested for each product: from a single requirement, a test case has to be written for each specific product. In this paper, we propose a method supported by a toolset, in which test cases for each of the *different* products of a PF are generated from the *same* PF functional requirements. By using our approach, tests designers can take advantage of the presence of a common behavior of products, to reuse tests. We suppose that requirements are documented with high-level sequence diagrams: it is common practice in industries such as the defense industry for the main contractor to provide high-level functional requirements to their sub-contractors. Those sequence diagrams are combined into reusable assets, which we call *behavioral test patterns* (behTP), and used to automatically generate test cases specific to each product. The rest of the paper is organized as follows: Section 2 outlines our method. Section 3 defines and explains the building of our behTPs. Section 4 presents the generation method from those behTPs. Section 5 analyses the results of our method. Section 6 concludes.

## 2 A requirement-based testing method

Our main objective is to demonstrate the feasibility, utility and efficiency of generating test cases for each product from the requirements of the entire PF. Our method is illustrated in Figure 1. In the first part of the method (detailed in Section 3), the test designer constructs particular types of test objectives or test purposes, which we call behavioral test patterns (behTP), from certain of the use-case scenarios. Independently of the test development process, the PF design is developed and the final products are instantiated. Further evolutions can also be carried out without changing the behTPs library, which is considered an asset of the PF (the behTP are defined once, and then reused unchanged throughout the software life-cycle: a behTP is derived from the requirements of the PF, and is thus reusable for all its products). Then, for each instantiated product, a set of test



**Figure 1. The requirements-based methodology for PF testing**

cases is generated from the model of that product together with the set of behTPs. The generated test cases are thus specific to each product, since they stem from specific specifications. The test case synthesis is the second part of the method, and is detailed in Section 4.

BehTPs 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 product-specific model; completing the behTP with these details is therefore a task which can be left to the automatic synthesis. Allowing test designers to work at the behTP 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.

Use of our method and toolset requires two assumptions. First, the requirements of the PF are expressed in the form of use cases documented with generic scenarios (UML sequence diagrams). These sequence diagrams can be generic with respect to the name and type of the instances involved, as well as w.r.t. the method names, and the names and types of the parameters in the method calls. Second, we assume that we have (at the end of the design process) a UML model available for each product, containing: (1) a class diagram (2) state machines for the main classes (3) an object or deployment diagram representing the initial state of the system. These product-specific UML-models could be either built by hand, or by using automatic derivation techniques from a model of the PF.

### 3 From requirements to behavioral test patterns

In this Section, we define the test objectives we deal with, called behavioral Test patterns. Then we explain how to build them from the requirements of the system under test (SUT).

The use-case diagram and the sequence diagrams attached to them represent the requirements of the system. The behavioral test patterns are defined based on these requirements. The sequence diagrams are of either of two types: those describing nominal scenarios, expressing the standard use-case occurrence, and those describing exceptional ones.

A *behavioral test pattern* (behTP) is here defined as a set of generic scenario structures representing a high-level view of some scenarios which the system under test may engage in, according to its specification, and which we want to test. A behTP comprises three parts, each part being specified using sequence diagrams:

- an accept scenario: 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;
- reject scenarios: 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;
- a prefix: 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 behTP, 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 (optional) prefix scenario that must precede the accept scenario. Since this task does not involve writing test code or reading SUT code, the tester can concentrate on building relevant behTP.

The behTP are incomplete and generic in the sense that (1) they do not specify objects or types involved and (2) they specify only a small part of the messages involved. They are thus easy to build since they comprise only test objectives and not exact test sequences. The behTP are built during the analysis phase of the PF development, and added as test assets in the PF. When the testers are asked to test the PF, they test the products one by one, with sets of test cases automatically generated from the test assets (i.e. sets of behTPs) and the UML model of each product. Each behTP will produce a number of different test cases depending on the product for which the tests are generated, as described

in the next section.

## 4 From reusable behavioral test patterns to test cases

In this section, we give an overview of the test synthesis mechanism. The test synthesis tool TGV and its formal basis are presented in [4], the Umlaut UML tool in [3] and the joint use of these tools for conformance testing of UML models in [6]. In this paper, we demonstrate the interest of applying these powerful techniques to the generation of product-specific test cases from PF requirements. We adapt the techniques to the PF context by extending the test objectives of [6] to behTPs built from use cases and by defining the pre-processing necessary to adequately treat genericity. More detailed information (for the case study, the test scenario language and the synthesis process) can be found at [1].

**Pre-processing the behTP.** The requirements, and consequently the behTP are generic in the sense that, thanks to the use of wildcards and variables, neither the identity of the objects involved in the scenario nor the exact parameters values in the method invocations between these objects need to be fully specified. The exact objects and parameters involved must be deduced from the specification of the SUT, in our case, the particular product of the PF to be tested. This refinement of the test pattern may involve increasing the number of behTPs during the pre-processing as explained below. The pre-processing resolves only part of the genericity involved in the behTP: it solves the genericity on the objects instances of the scenarios for the accept and prefix scenarios, the genericity on non-primitives types, and the genericity using variables. The remaining genericity is resolved later by the test synthesis, since TGV tool can handle regular expressions in test objectives. During the pre-processing, the actual objects to be used must be found in the object diagram describing the initial state of the system (for dynamically created objects, naming conventions are used). A simplified version of the pre-processing algorithm is given in Algorithm 1. Its main principle is that an accept scenario or a prefix is replaced by its instantiation using the first object of the right type found, whereas a reject scenario is replaced by all of its possible instantiations. A key part of giving the behTP their product-independent status is to deal with variation modeled using inheritance. Indeed, a behTP scenario may involve objects of a given class C (such as Meeting), while the refined behTP (and thus the resulting test cases) may involve objects of descendant classes of C, exactly which descendant depending on the product under test. Since the wildcards used in the exchanges of messages represent primitive types, they are therefore left unchanged by the pre-processing.

**Processing the behTP and test case synthesis.** TGV test

---

### Algorithm 1 behTP pre-processing

---

```
boolean done=false
// treatment of reject scenarios
for each bTP in ens_bTP until done
find a declaration w=*:T with a wildcard or variable in s ∈btp.reject
for each subtype T' of T {
  for each object o:T' in OD {
    btp'=clone(btp)
    btp'.replace(w,o:T')
    ens_bTP.add(btp')}
done=not(ens_btp owns a btp |reject.own(*:T))
done=false
// treatment of the accept and prefix scenarios
for each bTP in ens_bTP until done
find a declaration w with a wildcard or variable in s ∈btp.prefixUbtpt.accept
case w=*. * {irrelevant}
case (w=*:T){
  for each subtype T' of T {
    choose one object o:T' in OD
    btp'=clone(btp)
    btp'.replace(w,o.name:o.type)
    ens_bTP.add(btp')}
case (w=obj:*){
  find o in OD |o.name=obj
  btp'=clone(btp)
  btp'.replace(w,o.name:o.type)
  ens_bTP.add(btp')}
done=not(ens_btp owns a btp | (prefix or accept).own(*:T or x:*))
}
```

---

objectives are then derived from the pre-processed behTPs. A semantics is given to each behTP in terms of a labelled transition system (LTS) with distinguished transitions *accept* and *refuse*. Then, test synthesis is performed using this LTS as a selection criterion on the LTS obtained from the UML model of each of the products. The synthesis mechanism is described in detail in [4, 6]. Just note that the tool uses on-the-fly model checking algorithms, and that the use of a prefix and reject scenarios helps to reduce the size of the state space to be explored, as does the existence of the pre-processing phase.

## 5 Experiments and discussion

Our case study is a virtual meeting server PF offering simplified web conference services. The whole system contains more than 80 classes and about 2000 executable statements. In this paper, we deal with a simplified version of the PF, and with 3 products of the PF: a demonstration, a personal, and an enterprise edition, respectively named DE, PE and EE. The aim of this case study is two-fold: determining how different the test cases are from one product to another, and estimating the efficiency of such a high-level test generation technique in terms of code coverage on a given product. The number of test cases generated per product for 3 given behTPs is given in Table 1. Since the case study is simplified, the number of generated test cases remains low: it increases in function of the number of vari-

	behTP1	behTP2	behTP3
Demonstration	3	1	0
Personal	10	3	0
Enterprise	7	3	3
Total number of different TC	10	3	3
Av number of TC occurrences	2	2.3	1

**Table 1. number of test cases generated for each product**

ants and products. Columns represent the behTPs and, for each product (row), the number of generated test cases is given. To give a synthetic view of how different the test cases are from one product to another, we give the average number of a test case occurrences (ANO), defined as:  $\frac{\sum_{i=1}^n \text{number of test cases for } P_i}{\text{number of different test cases}}$ , where the  $P_i$  represent the products. If we focus on the PE, 10 test cases are generated from behTP1 and 3 test cases are generated from behTP2. The generation process thus computes several test cases scenarios: they differ in the protocol followed to be able to speak (test cases combine the restricted access list for private meetings, and the other possible actions: asking to speak, being given the floor, ...). We remark that no test case is synthesized for behTP3: this is due to the fact that behTP3 is only applicable to the enterprise edition. If we consider the global results: the ANO values vary from 1 to 2.3. They are low: this is due to the fact that our PF does not contain many variants, and consequently the products are very similar from a behavioral point of view. In a real-world PF, this value would be much greater. The case of a ANO equal to 1 is presented in our example, for a behTP dedicated to a specific variant, present in a single product. The lower value of 2.3% corresponds to the behTP2 for which test cases are re-used across different products.

**Test efficiency.** We focus on one product, the DE. We first analyzed our implementation code: around 9% of the code is dead code (pertinent but unused accessors). Functional system testing cannot deal with this code: it has to be tested during unit testing. Around 26% of the code is robustness code: robustness w.r.t. the specification, which deals with non-nominal treatments, and robustness w.r.t. the environment which asserts that the inputs coming from the environment are correct.

We built a set of 18 behTP, 12 concerning nominal cases and 6 exceptional ones (robustness behTP). We synthesized the corresponding test cases, and measured the percentage of code they cover. About 98% of the reachable nominal code is covered, whereas only 50% of the robustness code w.r.t the specification is covered. Building complementary robustness behTP would help in having a better coverage of the robustness code. The robustness code w.r.t. environ-

ment is not covered; it concerns syntactic verification of the inputs, treatment of network exceptions, etc. These aspects are specific to the distributed platform.

In conclusion, a few generic behTPs can efficiently cover most of the nominal code and a significant part of the robustness code. However, the integration (for robustness w.r.t. environment) and unit (for “dead code” coverage) testing stages remain necessary. The method thus makes the PF testing task easier: the core of the method is to concentrate on building particular test objectives for the whole PF, which is certainly an easier task than directly writing test cases, and then let the TGV tool find the corresponding test case, depending of the operational semantics of each product, computed from the UML model by the Umlaut tool[3]. The test cases obtained are efficient in terms of code coverage, as shown by our experiments.

## 6 Conclusions

In this paper, we have presented a method for functional testing of PF. This method is a first proposal to bridge the gap between the PF requirements and functional test cases dedicated to each product. It is based on building behavioral test patterns (which is a relatively easy task and is certainly easier than writing test cases) which are used to drive the automated generation of product-specific test cases (thanks to prototype tools). Experiments show that test cases synthesized from functional requirements may cover most of the nominal and part of the robustness code of the final product.

## References

- [1] Technical and experimental material. <http://www.irisa.fr/triskell/results/ASE03/>.
- [2] M. Griss. Implementing product-line features with component reuse. In Springer-Verlag, editor, *Proc. of the 6th international conference of Software Reuse*, pages 137–152, 2000.
- [3] W. Ho, J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac’h. UMLAUT: An extendible UML transformation framework. In *Proc. of the 14th IEEE International Conference on Automated Software Engineering (ASE’99)*, page 275.
- [4] C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *Proc. of the 6th world conference on integrated design and process technology*, 2002.
- [5] J. McGregor. Testing a software product line. Technical report, CMU/SEI, 2001.
- [6] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel, and A. Le Guennec. System test synthesis from UML models of distributed software. In *Proc. of the 22nd Conference on Formal Techniques for Networked and Distributed Systems (FORTE’02)*, Houston, Texas, 2002.