Managing Execution Environment Variability during Software Testing: an industrial experience

Aymeric Hervieu[1,2], Benoit Baudry[2], Arnaud Gotlieb[3]

[1] KEREVAL, Thorigné Fouillard, France
[2] INRIA Rennes Bretagne Atlantique, Rennes, France
{Aymeric.Hervieu, Benoit.Baudry}@inria.fr
[3] Certus Software V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway
{arnaud}@simula.no

Abstract. Nowadays, telecom software applications are expected to run on a tremendous variety of execution environments. For example, network connection software must deliver the same functionalities on distinct physical platforms, which themselves run several distinct operating systems, with various applications and physical devices. Testing those applications is challenging as it is simply impossible to consider every possible environment configuration. This paper reports on an industrial case study called BIEW (Business and Internet EveryWhere) where the combinatorial explosion of environment configurations has been tackled with a dedicated and original methodology devised by KEREVAL, a french SME focusing on software testing services. The proposed solution samples a subset of configurations to be tested, based on environment modelling, requirement analysis and systematic traceability. From the experience on this case study, we outline the challenges to develop means to select relevant environment configurations from variability modelling and requirement analysis in the testing processes of telecom software.

1 Introduction

Business and Internet EveryWhere TM (BIEW) is an Internet connection software developed by Orange, a worldwide Telecom Company. BIEW has been designed to fulfil professional needs in mobility. It aims to provide user the ability to connect to the internet through different means, from everywhere. BIEW is able provide an internet connection using 3G, Wifi or Ethernet protocols. Today the application is used by more than 1.5 millions users from all around the world. For the end user the application overrides the connection manager of the operating system, and gathers in the same application different ways to connect to Internet. Fig. 1 presents a screenshot of the application where the grey panel represents state of the connection: mean, time, connexion quality, and amount of data transferred. At the bottom, a set of icons permits users to access to the various functionalities of the application. As it has to provide users the ability to connect to Internet through different protocols, the application has to handle a large number of physical devices. The application has been specified to behave correctly on more than 2,000,000 different environments, composed of operating systems, 3G, Wifi dongles, browsers and mail clients. A the end of the project, specifications of the application contained 1493 requirements. As Internet connection is business critical for a company like Orange, the acceptance testing phase of the BIEW software is a major step in the life cycle of the software. A major dysfunction of the application would have serious financial and reputation consequences for the company. In the previous testing rounds of the application most of the testing activities were based on the craft and experience of software testers. There was no formalized acceptance test process and test teams had no vision of the general efficiency of their test activities. As the project grew, the number of requirements increased and tests activities only based on manual craft and experience appeared to be insufficient. The testers did not have the necessary expertise in requirements management and systematic environment modelling to handle the growing number of different conditions under which the test cases had to be executed. As this process was highly challenging, KEREVAL, a french SME focusing on software testing services, has been solicited to develop a dedicated testing methodology based on variability modelling, requirement analysis and systematic traceability between requirements and test cases. To validate this application, we faced to 3 challenges.

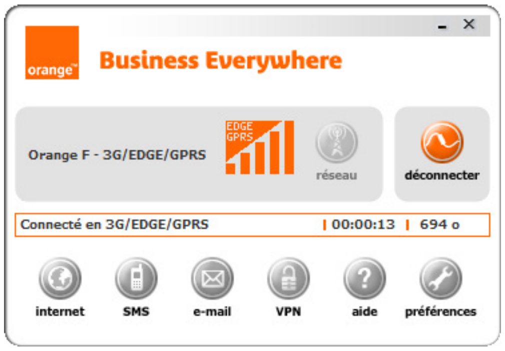


Fig. 1. Screenshot of the application

1. The first challenge is the explosion of environment configurations, due to the heterogeneity of devices available to the end-user. We were asked to find a systematic selection strategy to reduce the number of configurations under which the system has to be tested ;
2. The second challenge is the reduction of the effort to deploy a configuration to run a set of test cases. To deploy an environment configuration, the testers have to install an Operating System (OS), to set up drivers and plug in devices, and finally to configure the application. These tasks are time-consuming and we were asked to find ways to reuse environment configurations as much as possible ;
3. The last challenge is to keep track of the relations between requirements, environment configurations and test cases. Any change in the requirements or the environment configurations may affect the testing strategy and then needs to facilitated by means of a better traceability.

Unfortunately, the literature contains few industrial reports explaining how these challenges can be efficiently handled. Olsen et al. recently presented in [1] an approach for testing professional printers, that has been deployed by a Big Company. The authors considered a controller having a large number of input parameters and chose to model the environment of the controller (i.e., logical relations between parameters) with propositional logic formula. Based on these formula, they generated test cases covering the pairwise combinatorial testing criterion, and executed them on the system. In the BIEW project, a model of the environment is insufficient to generate test cases and test cases need to be generated from test requirements. More over, the owner of BIEW (i.e., Orange) considered worth reaching the coverage of test requirements rather than any other testing criterion.

This paper reports on the methodology we designed and deployed at KEREVAL, to validate the 1493 BIEW test requirements over the different configuration environments. The overall project was intended to last for 5 years. The testing effort for validating each new version of BIEW was estimated in between 100 and 400 Person-Days, with a mean of 300 Person-Days, meaning that it represents an important part of the overall cost of the development. This paper details our methodology to select and run tests cases, to manage the variability associated to the various configuration environments, to deal with the traceability issue between requirement, test cases and environments. The contributions of the paper are two-fold: it introduces an original methodology to manage the complexity of the combinatorial explosion of configuration environments and it describes the benefits and limitations of our implementation of this methodology ; it identifies the challenges that still have to be handled to improve the testing of the BIEW software and more generally telecom software applications.

The rest of this article contains three sections. Sec.2 describes the testing methodology through its main components, i.e., inputs processing, environment variability modelling, test requirements management, test case generation and traceability management. Sec.3 reports on the implementation of this methodology, its industrial adoption and discusses of its benefits and limitations. This section also introduces new research perspectives by identifying several key scientific challenges. Finally, Sec.4 concludes the paper.

2 A methodology to manage test requirements and test cases on a large number of configuration environments

The French SME Kereval, specialized in testing services, designed a methodology to manage test requirements and test cases on a large number of configuration environments. The complete methodology is depicted in Fig.2. It takes both environment specifications and requirements as inputs, and produces concrete test cases and several variability matrix that capture test case execution verdicts. These variability matrix specify the test cases that are executed in selected configuration environments. The process includes 5 steps showed with diamonds shapes in Fig.2, namely *environment analysis*, *requirement analysis*, *test case selection*, *variability matrix design* and *test case execution*:

1. In the *environment analysis* step, the validation engineer converts the specification document into a set of environmental features (e.g., OS, browser, etc.) ;
2. In the *requirement analysis* step, the validation engineer splits the set of customer-oriented test requirements into environment-dependent requirements and environment-independent requirements. The former ones are tagged with the set of environmental features on which they depend ;
3. In the *test cases selection* step, the validation engineer extracts test cases from the requirement analysis phase ;
4. The *variability matrix design* step produces the variability matrix that associate the set of environmental configurations to each test case. The matrix also store the test verdict for each test case with its associated set of configurations, when it becomes available ;
5. Finally, the *test case execution* is a process where the validation engineer distributes individual tasks to the engineer in charge of the settings of a test environment and the execution of the tests cases ;

The rest of the section is devoted to the detailed presentation of these steps, which composes the methodology introduced in this paper.

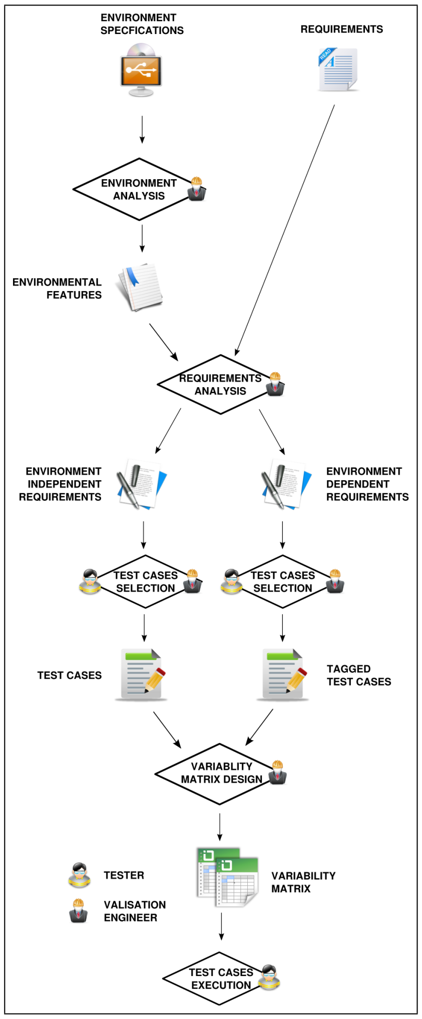


Fig. 2. Test design process

2.1 Environment Analysis

Provided by Orange, a document specification describes an unstructured list of environment items (e.g., OS, browsers, ...). A very first step of our process is to analyse this document and to extract a structured view of the environment under the form of possible configurations. By gathering items corresponding to physical devices or software artefacts, called *environmental features* our process leads to identify possible distinct configurations under the form of *environment configurations*. For the BIEW software, we identified 8 distinct *environmental features*:

- **OS** (5): Win. 2000, Win. XP 32 bits, Win. XP 64 bits, Win. Vista 32 bits, Win. Vista 64 bits
- **Mobile** (25): Novatel Xu870, GT Max GX0301, Lucent Merlin U530, Huawei E870...
- **Wifi internal** (5): intel centrino 2100, 2200, 2915, 3945,
- **Wifi external**(8): Sagem 706 A, Sagem 703...
- **Modem** (8):Sagem F@st 800 USB, Thomson ST330, Siemens A100, ZTE ZXDSL 852...
- **VPN** (4): Safenet, Cisco, Avasy
- **Mail Client** (4): Outlook, Outlook Express, Windows Live Mail, empty
- **Browser** (4): Firefox 2.0, Firefox 1.5, Internet Explorer 5.5, empty

Each *environmental feature*, except **OS**, is optional, that's why each has a common value : empty. We distinguished Wifi Internal and Wifi External features into to 2 *environmental features*, for 2 reasons. Wifi External devices can be plug freely on any configuration while internal devices cannot. Wifi Internal devices are already recognized by the Operating System, drivers are embedded in the OS, on the contrary, Wifi External may sometime require external driver provided by BIEW software.

We also specify an additional *environmental feature*, which is not associated to physical devices or software artefacts. This *environmental feature* contains the kind of telecommunication channel:

- **Bearer** (4): Mobile, Modem, Wifi, Undef

It permits validation engineer to identify certain telecommunication channel independently of any physical devices. Section 2.2 illustrates how this modelling choice will assist the validation engineer .

A configuration, called an *environment configuration*, is a tuple of 9 values: (OS, Mobile device, Internal wifi device, Wifi USB device, Mail client, VPN, Browser,Bearer). Note that *environment configurations* do not necessarily represent actual configuration as, for example, nothing forbids 2 web browsers to be installed within the same machine. Note also that some configurations are not necessarily a *valid environment configurations* because some combinations are forbidden. For example, Firefox 1.5 cannot be installed on Windows Vista 64 bits. According to our definitions, the number of possible *environment configurations* (valid and invalid) is exactly 2560000 [4].

The *environmental feature* **Bearer** will be set up at Undef (undefined) value if a tester selects an *environment configuration* containing more than 1 mean of connection. i.e. in the case of an environment configuration containing a **mobile device** Novatel Xu870 and **Wifi External** device Sagem 706. The main weakness of this step of the process is that invalid configurations are not exclude. Informal knowledge of the tester is not captured.

[4] $\sharp OS * \sharp Mobile * \sharp WiFiInternal * \sharp WiFiExternal * \sharp Modem * \sharp VPN * \sharp Mail * \sharp Browser$

2.2 Requirements Analysis

Requirements have been produced by Orange, and used by developers to write the specification of the BIEW application. The requirements are gathered by *functional domains*, which correspond to the major functionalities of BIEW : e.g. `Power Management` , `POP Locator`, `Startup Preferences` ... In this project, 43 functional domains corresponding to 1493 requirements, are identified. Fig.3 shows a requirement extracted from the functional domain `Wifi Management`. Requirements are composed of a header, including a unique ID, a version number a small summary of the requirement goal, and a detailed explanation of the expected application behaviour on a given situation. Based on the identified *environmental features*, validation engineers decide whether a functional domain is dependent on the environment or not. Classification of a functional domain as dependent or independent of the environment is made after discussions with project managers, software engineers and software testers. For the BIEW application, 33 functional domains (including 841 requirements) are classified as environment-dependent while 10 of them (including the remaining 652 requirements) are environment-independent. Environment-independent functional domains contain requirements that are not dependent of the environment. Each requirement of an environment-dependent functional domain can be tagged with up to 2 tags. *Tags* values correspond to *environmental features.* When a tag is assigned to a requirement, it means that the requirement should be tested in every possible values of the tag. For example, if a requirement r is tagged with OS, then r should be tested over Windows XP 32 bits, Windows XP 64 bits, ... The requirement shown in Fig.3 belongs to an environment-dependent functional domain. As this requirement describes a situation where BIEW depends on the devices wired to the machine, it includes a tag `[WIFI]`. Thus, for this requirement, BIEW should be tested with all the possible Wifi settings. A requirement tagged `[BEARER]` means that the requirement does not directly depend of physical device, but depends on the nature of the telecommunication channel. The

```
[RQ 02000 _V8.0.1_ Select Wifi device in Settings][WIFI]
The user can also select the Wifi device in the settings. He can choose
in a listof all devices authorized by the customisation and detected by the
Client Software Suite on the PC.
```

Fig. 3. Environment-dependent requirement

identification of environment dependency is a complex task as it requires a deep understanding of the application domain, thus requiring extensive discussions among the project members: software developers and testers. We estimated that the time spent to identify requirements dependencies was about 15 Person-days.

2.3 Test Cases Selection

During this step, test cases are selected for each requirement. An example of such a test case is given in Fig.4. Each test case is composed of:

- A unique identification number, which links the test case with the requirement it originates, for traceability;
- A *tag* (optional), which allows validation engineers to identify the environment dependency;
- Pre-requisites that describe the necessary conditions for the test case to be executable;
- A test objective, that is the goal of the test case;
- A test procedure, that gives the detailed plan for executing the test case.

Test case is written in Quality Center (QC), and associated to its requirement. QC permits to maintain the traceability between requirements and test cases. When a requirement evolves, is modified, added, or suppress,the impacted test cases are distinguished. During a test campaign, validation engineer executes the test case, and reports verdict in Quality Center. For BIEW, 3102 test cases are selected from the requirements, among which 1231 are associated to environment-dependent functional domain.

[353-RQ01980][WIFI]
Objective: `Check the prompt display for one descriptor and one security key WPA2.`
Pre Requise:
```
Business EveryWhere Kit installed.
Acces Point AP1 selected
Wireless lan seted up with WPA 2 security
```
Test Procedure:
```
- Launch the BIEW application
- Click on the button connect, on the main screen
- Select the access Point AP1
```

Fig. 4. An example of a Test case

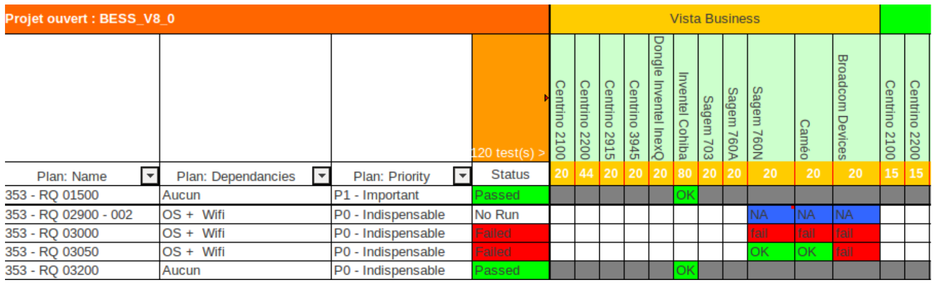
2.4 Variability matrix design

The tag is used to reduce the number of test execution to perform for a given test case. This is basically the approach we adopted to reduce the combinatorial explosion:

- if a test case has no tag, then only a single environment configuration is selected for test case execution ;

– if a test case has a single dependency, all the environment configurations related to the environmental feature are used for test case execution. For example, if the feature is `WIFI`, then all the `WIFI`-environment configurations are selected;
– if the test case has two tags, then all the combinations of environment configurations will be selected for test case execution.

A *Variability matrix* is designed for each of 43 functional domain. When the domain is environment-independent, then the test cases have to be executed on a single environment configuration. On the contrary, test cases originating from environment-dependent functional domain have to be executed on several environment configurations. A variability matrix (as the one shown in Fig.4), captures the dependency in this latter case. In this matrix, each row corresponds to a test case, while columns represent environment configuration and environment configurations combination. As the environment dependency has been limited to 2 *environmental features*, only two levels of combination are possible. In the matrix of Fig.5, the first columns include informations such as the test case name, its priority, its environmental features and its status. Of course, status are only available once test execution has been started. A status can be either `Passed`, `Failed` or `Not completed`, corresponding to the state of the test execution process. A color is associated to each element of the matrix: grey means that the test case, within the considered configuration, is not required to be executed, white means that the test case has to be executed, green means `Passed`, red means `Failed`, while N/A means `Not applicable`. This latter case holds for test case that cannot be executed on a given *environment configuration*. When a test cases has been run on all the *environment configurations* then its statuts is turned `Passed` or `Failed`, depending on the results over all the environment configurations. If the test case fail for at least one *environment configuration* then, its status is turned `Failed`. Variability matrix are then associated to their



Projet ouvert : BESS_V8_0				Vista Business												
			120 test(s) >	Centrino 2100	Centrino 2200	Centrino 2915	Centrino 3945	Dongle Inventel InexQ	Inventel Cohiba	Sagem 703	Sagem 760A	Sagem 760N	Caméo	Broadcom Devices	Centrino 2100	Centrino 2200
Plan: Name	Plan: Dependancies	Plan: Priority	Status	20	44	20	20	20	80	20	20	20	20	20	15	15
353 - RQ 01500	Aucun	P1 - Important	Passed	▨	▨	▨	▨	▨	OK	▨	▨	▨	▨	▨	▨	▨
353 - RQ 02900 - 002	OS + Wifi	P0 - Indispensable	No Run									NA	NA	NA		
353 - RQ 03000	OS + Wifi	P0 - Indispensable	Failed									fail	fail	fail		
353 - RQ 03050	OS + Wifi	P0 - Indispensable	Failed									OK	OK	fail		
353 - RQ 03200	Aucun	P0 - Indispensable	Passed	▨	▨	▨	▨	▨	OK	▨	▨	▨	▨	▨	▨	▨

Fig. 5. Variability matrix

functional domain in QC. Thanks to these rules, only 10603 test case executions were run instead of 33685 + 1871, while the overall environment diversity of the test cases was preserved. The quality of the test suite was evaluated by

2 distinct entities. Developers of the application where executed test cases during development steps put in excerpt a several defects. This test step permits developers to fix quickly the application. The second is an independent entity the Orange development team.This entity is a branch of Orange group which valid the release of a major version. The second entity identified few defects. Now, a part the validation process of the branch, for BIEW 9, relies on our test platform. We were not able to extract relevant metrics to illustrate the quality of the test suite. We obtained those information thought discussions other the different stakeholder.

2.5 Test Case Management and Execution

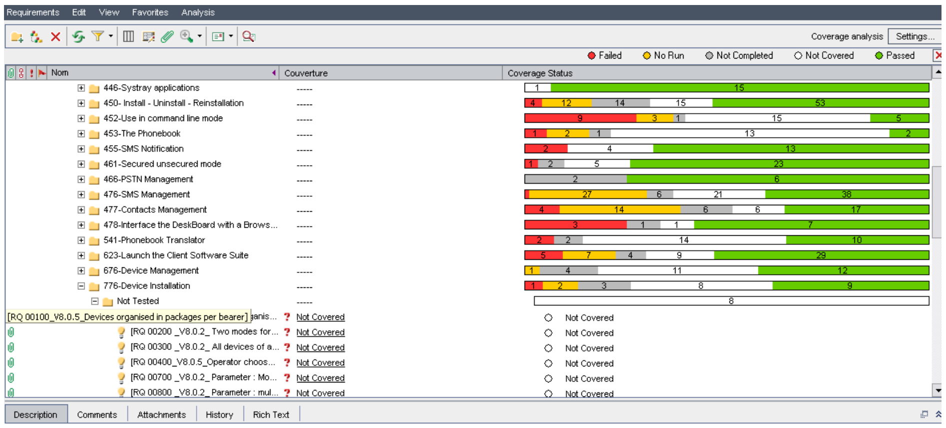
In the BIEW project, we gathered 1493 requirements and classified them in terms of functional domains. For each functional domain, identified as environment dependent, we associated a specific variability matrix. As a result, test cases were formally associated to the requirements for faciliting traceability.

In order to monitor test activities, we associated a status to each requirement in our methodology. The status of a requirement depends on the state of execution of its associated test cases. There are five possible values for the status of a req:

- Not Covered: if there is no test case associated to the req. ;
- Failed: if at least one of the test cases failed ;
- Passed: if all the test cases successfully passed, in all the environments specified by the variability matrix ;
- Not Completed: at least one of the test cases associated to the req. has not yet been executed ;
- Not Runned: none of the test cases associated to the req. has been executed ;

Of course, the main relevant metric used during the acceptance testing phase is the number of covered requirements (and their status). Using this metric, the validation engineer can follow quite easily the evolution of the project and can provide informations related to the state and quality of the deliverables to the development team. Fig. 6 shows a screenshot of the Quality Center (QC) tool that has been used in the BIEW project. QC centralizes and reports on all the test activities of the project. On the left, all the requirements classified by functional domain are shown. For each functional domain, the validation engineer follows test activities through the diagrams shown on the right.

During test execution, the validation engineer assigns to each tester a subset of existing variability matrix. The testers are then responsible of the execution of test cases, as they are specified in the distinct matrix. A test case assembles 3 elements: an *environment configuration*, a test input and a test verdict. Note however, that a matrix does not specify the order on which the test cases have to be executed. Testers have to select a *environment configuration* according to the availability of material (e.g., USB sticks, SIM card) and to prioritize the



🔴 Failed 🟡 No Run ● Not Completed ○ Not Covered 🟢 Passed

Nom	Couverture	Coverage Status
446-Systray applications	-----	⬜ 1 🟩 15
450- Install - Uninstall - Reinstallation	-----	🟥 4 🟨 12 ▨ 14 ⬜ 15 🟩 53
452-Use in command line mode	-----	🟥 9 🟨 3 ▨ 1 ⬜ 15 🟩 5
453-The Phonebook	-----	🟥 1 🟨 2 ▨ 1 ⬜ 13 🟩 2
455-SMS Notification	-----	🟥 2 ⬜ 4 🟩 13
461-Secured unsecured mode	-----	🟥 1 ▨ 2 ⬜ 5 🟩 23
466-PSTN Management	-----	▨ 2 🟩 6
476-SMS Management	-----	🟥 🟨 27 ▨ 6 ⬜ 21 🟩 38
477-Contacts Management	-----	🟥 4 🟨 14 ▨ 6 ⬜ 6 🟩 17
478-Interface the DeskBoard with a Brows...	-----	🟥 3 ▨ 1 ⬜ 1 🟩 7
541-Phonebook Translator	-----	🟥 2 ▨ 2 ⬜ 14 🟩 10
623-Launch the Client Software Suite	-----	🟥 5 🟨 7 ▨ 4 ⬜ 9 🟩 29
676-Device Management	-----	🟨 1 ▨ 4 ⬜ 11 🟩 12
776-Device Installation	-----	🟥 1 🟨 2 ▨ 3 ⬜ 8 🟩 9
Not Tested	-----	⬜ 8
[RQ 00100_V8.0.5_Devices organised in packages per bearer] ganis...	Not Covered	○ Not Covered
[RQ 00200 _V8.0.2_ Two modes for...	Not Covered	○ Not Covered
[RQ 00300 _V8.0.2_ All devices of a...	Not Covered	○ Not Covered
[RQ 00400_V8.0.5_Operator choos...	Not Covered	○ Not Covered
[RQ 00700 _V8.0.2_ Parameter : Mo...	Not Covered	○ Not Covered
[RQ 00800 _V8.0.2_ Parameter : mul...	Not Covered	○ Not Covered

Fig. 6. Screenshot of the test project in Quality Center

execution of test cases based on their knowledge of the fault-proness of the configuration. When *environment configurations* contain items tagged as `[Bearer]`, it means that the test case execution does not require a particular physical device to be set up. Then, the tester can select an environment configuration with any item to perform test case execution.

3 Benefits, limitations and possible innovations

Kereval has developed a cost-effective systematic methodology that relies on two key-components: the formalization of execution environment as a set of *environmental features* and the analysis of customer-oriented requirements to clarify their dependency on *functional domains*. Since the methodology keeps track of the link between requirements and test cases, much testing effort is saved by limiting the number of test cases executed on each environment. The testing effort is also reduced during test cases design, as their dependency to environment has been explicited. Still, the methodology is perfectible. Let us review the limitations we considered, the identification of which could serve as a basis to improve the methodology.

– In the methodology, *environmental features* are totally independent from each other, while in fact, there are many dependencies among them ;
– Several configuration environments, having distinct sets of *environmental features*, are in fact redundant, meaning that test efforts could be saved if we could capture redundant environments ;
– In the methodology, we considered that a maximum of 2 tags could be associated to each requirement for facilitating the representation of dependencies of requirement to the environment configurations ;

– Requirements and environments often change from one version to another of the BIEW software. In our methodology, we did not consider the benefice that could be brought by an impact analysis of these changes ;

In this section, we review each of these limitations by identifying their roots, and, by studying existing research results, we propose and discuss potential improvements of the methodology.

3.1 Explicit representation of the variability within environment configurations

For BIEW, Orange provided us an informal description of the environment configurations, from which, we extracted a list of environmental features. However, in this process, we ignored that some environment configurations may be invalids and some others may appear in the near future. For example, nothing prevented us to consider an *environment configuration* running **Firefox 1.5** on a **64-bits OS** platform, even if Firefox 1.5 cannot run on 64-bits computer architecture. This kind of informations is never explicited in the specification documents, and typically belongs to the background knowledge of the validation engineers. In the case of BIEW, invalid configurations are not selected because their number is still limited. However, on other projects where the number of discrepancies between environment artefacts is larger, such an informal approach is no more acceptable. Another related limitation of our methodology is the limited notion of *environment configuration* which disallows the validation engineer to consider environments with several distinct browsers or client mails. For us, the root of these limitations is the absence of a *formal model* able to capture the variability within environment configurations.

In the literature, feature modelling, introduced by Kang in [2], enables complex and inter-dependant environment variability representation. Feature modelling introduces a tree-based graphical representation of the variability within a set of components of a system, or a set of options within a product line, or a set of features of an environment representation . Looking at the so-called *Feature Model*, which basically captures a set of propositional logic formula representing distinct environments, we modelled the dependencies within environment configurations of BIEW. Fig. 7 is an excerpt of this Feature Model, where the discrepancy between Firefox 1.5 and 64-bits architecture can be explicited using a special operator, called **Mutex** (i.e., exclusive disjunction). In this model, an operator OR can be used to represent configurations with several browsers, enabling the selection of environments with multiple features. The overall Feature Model we built for BIEW is composed of 66 features and implicitly represents $8,243,200$ distinct configurations. Using a Feature Model, a number of manual activities for the testing of BIEW could be automated. Benavides et al. [3] surveyed the automated analysis of Feature Models and identified key analyses, such as the so-called *valid product* and *valid partial product* operations that could be useful in our case:

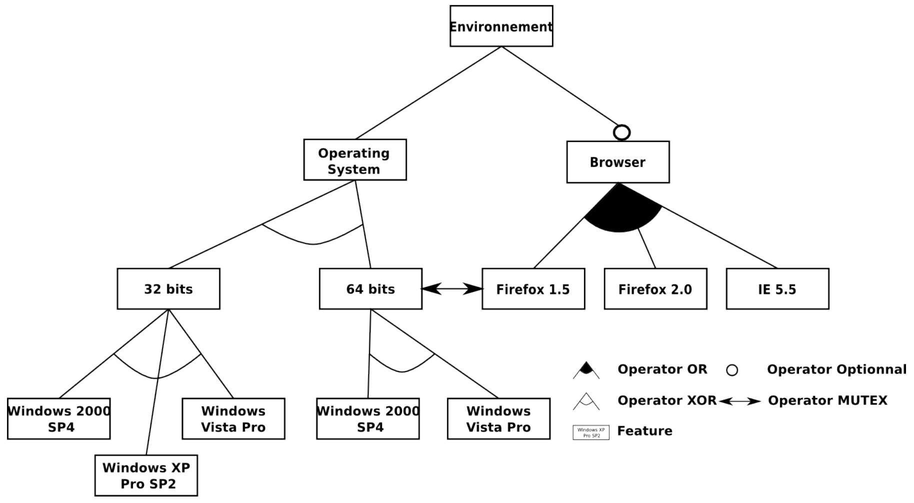


Fig. 7. Excerpt of the feature model representing the execution environment of the BIEW application

- *Valid product* verifies that a given environment configuration respects all the constraints of a Feature Model. For example, a configuration with both Firefox 1.5 and any 64-bits architecture will be automatically rejected. Implementations of this operation relies on the usage of SAT-solving or Constraint Programming techniques.
- *Valid partial product* is a similar operation over only a subset of features, and enables in addition the automatic completion of a partial environment configuration.

To sum up, we think that capturing the distinct environment configurations with a formal model of the variability will be useful to improve the test management and execution of the BIEW software.

3.2 Elimination of redundant configuration environments

In our methodology, the validation engineer identifies the dependency among *environmental features*, through a careful analysis of the specification documents and the customer-oriented test requirements. These dependencies are captured within *variability matrix*. However, a detailed analysis of the *variability matrix* shows that several test environments are redundants. In fact, requirements from distinct functional domains can be tagged with the same features, leading to the creation of distinct matrix, although they represent similar *environment configurations*. We identified 149 such duplicated environment configurations over the 390 configurations used during the overall test project.

3.3 Improving the internal representation

Our methodology involves the tagging of requirements with *environmental features*, to identify their environmental dependencies. Each requirement is tagged with 1 or 2 environmental features because we used a simple two-dimensional representation (i.e., *variability matrix*) for specifying the link between the environment configurations and the requirements.Then tagged requirements are tested under all the combinations of the items of the values.

The current modelling does not permit the validation engineer to define precisely test environments. For example, validation engineer cannot specify that a requirement has to be tested over 1 operating system 32 bits, and 1 operating system 64 bits, under a WLAN and a mobile connexion, with Internet Explorer, and Firefox. To design a test environment according to the depicted process validation engineer needs to 3 new environmental features ({ 1 OS 32 Bits, 1 OS 64 Bits }, { WLAN, Mobile }, { IE,FireFox }), and remove the tag limitation. Creating new environmental features for each specific needs is not a satisfying solution.

The best way to handle this limitation is certainly to increase the declarativity of our approach by allowing the validation engineer to specify at a finest coarse the test environments. Domain specific languages would permit the validation engineer to define precisely its test environments.

3.4 Impact analysis for requirements evolution

In case of evolution of requirements or environmental features, our methodology does not provide tools to help us quickly identify elements to modify and adapt to properly manage these changes. Even if our test management tool, Quality Center, can rapidly detect impacted test cases, nothing is proposed to adapt requirements and tags, that are store in variability matrix. In practice, when a new value for an environmental feature is introduced, test engineers have to re-examine all the variability matrix and identify the impacted requirements.

The literature contains many propositions to handle efficiently these evolutions. For example, Hartman et al. in [4] proposes to use Feature modelling with dependencies to manage context evolutions. Metzeger et al. [5] introduced *xlink* to link two distinct variability models, while Than Tun et al [6] used xlinks to formally establish the relation between a set of requirements to a set of features. Then, using these xlinks, their approach permits the validation engineers to select configurations that cover a selected subset of requirements. We think that this approach is valuable and could be implemented in our specific case for handling the evolution of requirements or environmental features.

3.5 Test criteria over the environment dependency

As exhaustive testing of every requirement on every possible environment configuration is impossible, test criteria have to be introduced in any methodology aiming at testing telecom software applications. In the case of BIEW, we implicitely considered every pair of values for environmental features, meaning that we tested the dependency to the environment with pairwise testing, a Combinatorial Interaction Testing (CIT) criterion [7].

A limitation of original approaches of CIT is however that it did not consider constraints among the variables [7, 8]. In the case of BIEW there are constraints among the environment features that capture the restrictions in configuration environments (see Sec.3.1). Recently, several authors proposed means to generate test configurations from feature models with constraints [9–12]. The authors of this article also contributed to this domain with a similar approach based on Constraint Programming [13]. Generating pairwise-covering configurations has also been studied for other representations of variability and constraints, e.g., [14]. Another intersting extension of CIT approaches is their ability to handle other testing criteria than pairwise. For example, it is possible to consider 3-wise or even N-wise combinations between the variables. We think that qualifying our test methodology with respect to these criteria will be helpful to improve our understanding of the achieved level of quality. This would be helpful in the discussions with our customer to adjust precisely the methodology with the expected level of quality.

4 Conclusion and perspectives

This paper presents a methodology we designed at Kereval to validate a telecom software application on a large number of distinct environment configurations. The main challenge we dealt with consisted to handle the potential combinatorial explosion of the number of possible configurations. In the proposed methodology, we adopted an approach that links the requirements to the environment through the usage of *environmental features*, and dedicated *variability matrix*. We performed a systematic identification of the dependencies between requirements and environmental features and thus were able to construct. We also kept the traceability between test cases, environments and requirements by using these elements. Thanks to this testing methodology, we showed that 70% of the test definition/execution effort could be saved over an exhaustive testing approach. However, we also identified several limitations in our methodology and the paper shows that is a large room for improvements. Among them, the absence of a formal representation of the variability (e.g., Feature Model) is the main limitation to address the problem of the combinatorial explosion of the number of environments to consider. We can also mention the need for impact analysis of requirements and environment change.

Our future plan includes a better formalization of the methodology, through the usage of variability models. Recent works on feature modelling enable automated analysis and then could be highly beneficial in the context of BIEW [6, 5, 4, 3]. We also plan to reason over variability models in order to generate test configurations that respect Combinatorial Interaction Testing criteria [13]. Following an approach inspired by xlink introduced by Hartmann and Than Tun

[6, 4], we will also exploit these variability models to facilitate impact analysis of change in requirements and environmental features. On another side, we would like to evaluate the potential of our methodology on other projects. In some sense, BIEW was a first industrial application that allowed us to identify the limitations of the approach. KEREVAL is involved in several distinct telecom application testing projects (e.g., the testing of mobile phone applications on 15 distinct platforms) and BIEW was the only project on which such a variability management approach was deployed. We are convinced that a fine-tuned methodology for managing distinct environment configurations is essential to save effort and cost in these kind of projects.

5 Acknowledgements

We thank Ludovic Rocher and Mokrane Kessaci for their precise answers to our multiple questions related to the BIEW project. We also thank Orange company which allowed us to use the result of the BIEW testing project in this paper.

References

1. P. Olsen, J. Foederer, and J. Tretmans, “Model-based testing of industrial transformational systems,” in *ICTSS*, 2011, pp. 131–145.
2. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.
3. D. Benavides, S. Segura, and A. Ruiz-Cortés, “Automated analysis of feature models: A detailed literature review,” *Information Systems*, no. 35, pp. 615–636, 2010.
4. H. Hartmann and T. Trew, “Using feature diagrams with context variability to model multiple product lines for software supply chains,” in *Proceedings of the 2008 12th International Software Product Line Conference*, 2008.
5. *Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis*, 2007.
6. T. Than Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans, “Relating requirements and feature configurations: a systematic approach,” in *Proceedings of the 13th International Software Product Line ConferenceTha*, 2009.
7. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The aetg system: An approach to testing based on combinatorial design,” *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437 – 444, 1997.
8. Y. Lei and K.-C. Tai, “In-parameter-order: A test generation strategy for pairwise testing,” in *HASE’98*, 1998, pp. 254–261.
9. S. Oster, F. Markert, and P. Ritter, “Automated incremental pairwise testing of software product lines,” in *Proceedings of SPLC’10*, 2010.
10. G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, “Automated and scalable t-wise test case generation strategies for software product lines,” in *ICST’10*, Paris, France, 2010.
11. B. P. Lamancha and M. P. Usaola, “Testing product generation in software product lines using pairwise for features coverage,” in *ICTSS*, 2010, pp. 111–125.

12. M. F. Johansen, Ø. Haugen, and F. Fleurey, "Properties of realistic feature models make combinatorial testing of product lines feasible," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds., vol. 6981. Springer, 2011, pp. 638–652.
13. A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *Proc. of the 22nd IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'11)*, Hiroshima, Japan, Nov. 2011.
14. B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering*, ser. SSBSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 13–22. [Online]. Available: http://dx.doi.org/10.1109/SSBSE.2009.25