INTERNATIONAL CONFERENCE ON ENGINEERING DESIGN, ICED'07
28 - 31 AUGUST 2007, CITE DES SCIENCES ET DE L'INDUSTRIE, PARIS, FRANCE

PROPOSAL FOR A HOLISTIC APPROACH TO IMPROVE SOFTWARE VALIDATION PROCESS IN AUTOMOTIVE INDUSTRY

Roy Awédikian[1&2], Bernard Yannou[2], Mounib Mekhilef[2], Line Bouclier[1], Philippe Lebreton[1] and Ludovic Augusto[1]

[1]Johnson Controls Inc., Pontoise, France
[2]Ecole Centrale Paris, Laboratoire Génie Industriel, France

ABSTRACT

Challenging the development of high quality complex software systems, that satisfy the requirements while being reliable, efficient, extendible and reusable, is not a new issue. Since 70's, software engineering tends to overcome this problem of quality in software systems. Although many methods, techniques and tools have been proposed, they are, most of the time, context-dependent and they focus on specific technical issues. In this paper and in order to globally improve the software validation process in automotive industry, we implement a systemic approach decomposed into two stages. In the first stage, we explore the different complementary domains contributing to the improvement of the validation process and in the second stage, we identify and propose to improve three main fields: the coverage of customer requirements, the usage of experience feedback and user profiles in test design. The proposals in these three domains are argued from ground observations and a review of the literature and clear specifications are provided in accordance with the expected benefits for the software industry.

Keywords: Test design, software testing, formal methods, experience feedback, software quality management, software reliability engineering

1 INTRODUCTION

Nowadays, electronics represents more than 30% of the global cost of a car. Car electronic architecture becomes more and more complex and car manufacturers outsource the design of electronic modules to automotive electronic suppliers. The software part is the added value of these modules and they account for more than 80% of the total number of defaults detected on these modules. A software default is called "bug". In software development, a loop-type design process is initiated between the client and the supplier. About ten intermediary client deliveries are carried out.
After each delivery, some "bugs" are detected by the client and forwarded to the supplier who must react quickly and efficiently. This could lead to the conclusion that car manufacturers have more efficient testing approaches than their suppliers. This efficiency can be related to many factors such as:

- The car manufacturers benefit from real electronic platforms where the supplier modules are installed and tested. In fact, the modules are tested in simulated real environments with surrounding modules in a global systemic approach.
- The car manufacturers use their experience feedback of recurrent bugs to test a given module with the knowledge of bug probabilities and even user behaviour profiles to design the most relevant tests.

Once an electronic module is launched on the market (i.e., integrated into a vehicle), an average of one "bug" per year is detected by the end-users, which may becomes dramatic for the electronic supplier in financial terms if the product has to be systematically changed.
In fact, in term of bug's occurrence, two types of contract engage electronics suppliers with car manufacturers:

- *Implicit contract*: during software development process, each customer delivery must be free of bugs.

- *Explicit contract*: on launched electronic module, car manufacturer tolerate a certain number of defective products expressed in terms of PPM (pieces per million). PPM includes all software defects but also electronic, mechanical and production defects.

As the automotive market becomes more and more competing, decreasing the number of "bugs" becomes of major importance for car manufacturers and consequently a major quality indicator for automotive electronics suppliers.

Through our research project, we were asked by the automotive electronic supplier Johnson Controls Inc or "JCI Company" to improve the quality of their software validation process in order to improve the quality of their products and therefore better satisfy the requirements and expectations of their clients.

In this paper, we go through this problem with a systemic approach in order to identify domains from which we might be able to improve the global performance of the software validation process. The added value of such an approach is the resolution of the problem with a global viewpoint. In other word, code and customer requirements reviews (by coverage approaches) but also experience feedback management and user profiles (here drivers') definition. Consequently we first characterize the software design environment of JCI Company and point out issues and related solutions proposed in the literature. Despite these solutions may be satisfactory under given isolated software testing issues, they concurrently do not appear to be adapted to practical industrial constraints in a competitive environment. Therefore, in the second section, we focus our research on three main questions and we propose founded recommendations and specifications for a new consistent test design approach based on the customer requirements coverage, the usage of experience feedback and of user profiles in test design. Our main goal is to design efficient test cases and thus reduce the number of bugs detected by customers and users. Finally, we point up the relevance of a "Six Sigma" approach on test design process which can illustrate and monitor the benefits of such a systemic approach in test design.

2 JCI COMPANY CONTEXT AND LITERATURE REVIEW

2.1 General organization

The automotive electronic division of JCI Company develops electronic products (body controllers, clusters, displays …) for car manufacturers (Renault, Nissan, Samsung, PSA, GM, Ford, Daimler Chrysler, BMW …).

In 2006, the automotive electronic division was restructured for a customer-oriented logic which replaces the previous product-based organization. It was motivated by the need of more communications between different product lines of a same customer. Two kinds of JCI automotive electronic design centers must be considered:

- "Core" centers, in France and Germany, where activities like management, customer relations and innovation are conducted
- "Development/Validation" center, in Bulgaria, where software development and validation activities are performed. Their main target is to check the compliance of the software product regarding the customer requirements.

2.2 The software project or design process organization

At JCI Company, the lifecycle of a software product is divided into 5 global stages (Figure 1):

1. RFQ: Request For Quotation
2. Design
3. Design Validation
4. Production Validation
5. Production

In each stage, engineering activities are performed:

- according to the standard V-cycle of the software industry (Figure 2)
- and in an iterative way in order to take customer constraints and requirements priorities into account

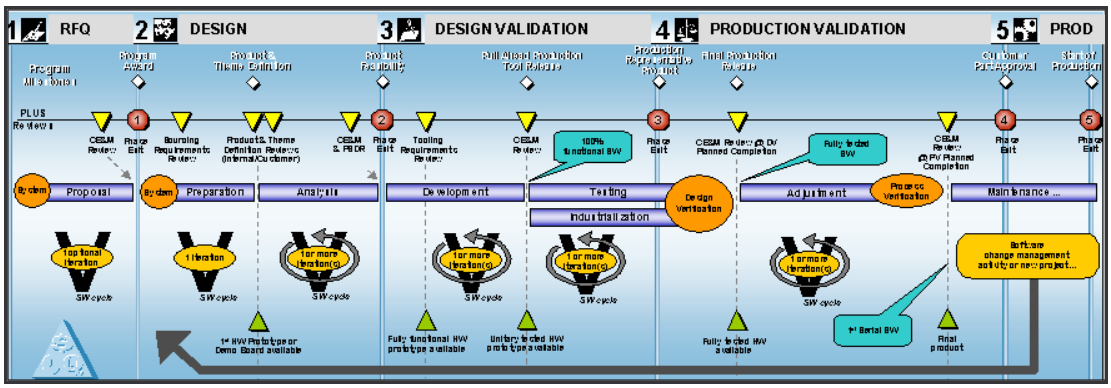

Figure 1 Overall software product lifecycle

The main engineering activities (Figure 2) are:

- Requirements specification and management
- Global design
- Component development
- Integration
- Validation

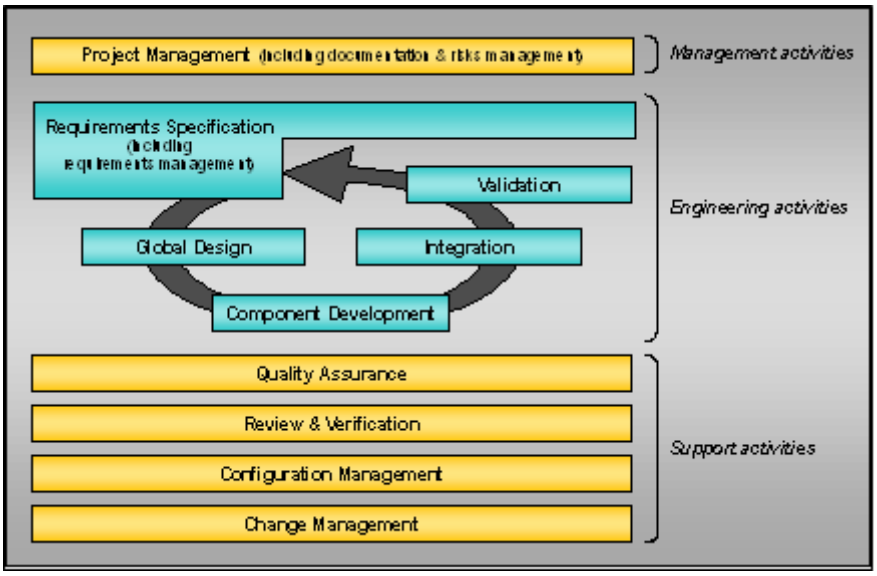

Figure 2 Elementary V-cycle for any of the global stages of the software lifecycle

Besides the project leader and the coordination team, one can identify two technical teams in a software project (Figure 3):

- One in charge of the development of the software product
- And the other in charge of its validation.

The coordination team is located in the so called “Core” centers, close to the customer (France and Germany). Development and Validation teams are generally located in Bulgaria.

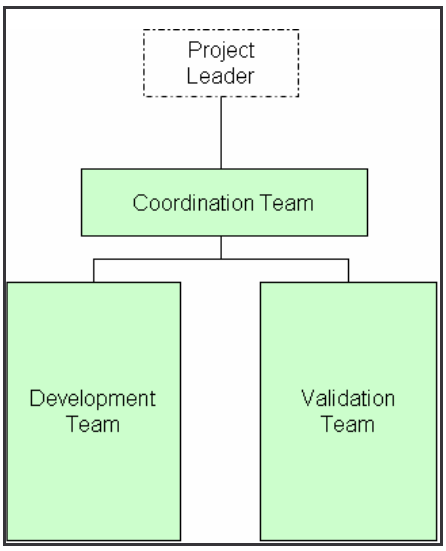

Figure 3 Functional organization of one software project at JCI Company

2.3 Customer requirements

At the beginning of a project, JCI Company officially receives the customer requirements. We identify three main characteristics of these requirements:

1. JCI customers consider different standards to express the requirements of a given electronic module. Even, the requirements of two different products (body controller and cluster) for the same customer are expressed in different formalisms. Some customers use semi formal methods, such as Statechart or UML illustrated respectively in [1] and [2], to express some of their requirements; others use natural language and therefore introduce ambiguity on the understanding of the requirements.
2. Requirements continuously evolve during development stages and also during series life of the product. The reason for this evolution is to filter out inconsistencies, change or improve the product.
3. Often requirements are expressed in many documents, emails, and even some phone calls.

In electronic domain, several standards organizations (including the Institute of Electrical and Electronics Engineers - IEEE) have identified in [3] four categories of requirements:

- *Functional requirements* are the main customer requirements. They point to the product functionalities. By example, in a body controller product, an automotive electronic module in charge of managing all electrical currents of a car, one of the functional requirements is the specification of "Front Wiper" user functionality.
- *Non functional requirements* stand the interface requirements between functionalities and software performance in term of CPU load and memory capacity. An example of non functional requirements can be the communication protocols.
- *GUI (Graphical User Interface) requirements* are the customer requirements related to user interface. This category of requirements is frequent in electronic display product.
- *Non technical requirements* include all organizational customer requirements. Confidentiality, return of experience, past defects reviews capitalization are examples of these requirements.

2.4 The software validation process

At the other site of physical products, software products are not physical entities but they are signal transfer functions, i.e. they transform input signals into output signals. The quality of a software product improves each time a software defect is detected and corrected. A software defect (a "bug") is an unexpected behaviour of a software product regarding the requirements. Validation techniques are divided into two families [4]: *static* and *dynamic* methods. Dynamic methods involve the execution of the tested program, whereas static methods do not.

Static methods include proofs and static analysis:

- *Proving* consists of stating the correctness of the program by establishing that its code satisfies

theorems deduced from the specification.

- *Static analysis* consists of analyzing the code of the program to verify that it satisfies implicit and explicit properties required by the specification. Static analysis can be either manual (code review) or automated (type checking, complexity measures).

Dynamic methods include symbolic execution and testing:

- *Symbolic* execution is performed by executing tested programs with symbolic input values instead of numeric ones, and yields as results symbolic expressions corresponding to the outputs of the program.
- *Testing* is performed by submitting a set of possible (numeric) inputs to the tested program and comparing the computed result (outputs) to the expected one. We define a *test case* as a sequence of input assignments (test actions) and output verifications (expected results).

Proof and symbolic execution methods are known as formal techniques that can successfully enhance the quality of software although they are very hard and expensive to deploy into a software development process. However, they are currently used on safety critical systems such as aircraft avionics, nuclear power plant control and patient monitoring. They require to be highly reliable, since failures in this kind of systems may lead to catastrophic consequences.

Testing approach is widely studied in academic research and deployed in software industry. At JCI Company, software validation process is primarily based on a testing approach. Beizer has illustrated well this approach in his book [5]. In this paper, we focused our research on testing domain.

Generally, testing methods are divided into two families [6], according to the source from which test cases are selected: *program-based testing* and *specification-based testing*.

- In *program-based testing*, also known as structural or white-box testing, test cases are derived from the program code. Tests cases are selected in order to fulfil a given coverage criterion (all instructions, all executable paths, all conditions … of the program). While this approach gives good results, it remains insufficient. For instance, examining the code of the tested program is unlikely to detect that the program does not perform one of the customer requirements.
- In *specification-based testing*, also known as functional or black-box testing, test cases are derived from the specification of the tested application as expressed by the customer. The goal is to select test cases that cover each property described by the specification. Specification-based testing is not sufficient to detect when the software performs undesirable behaviours that are not included in the specification.

Program-based testing and specification-based testing are complementary techniques; the bugs revealed with one technique are not necessarily easily detected with the other.

According to [7], both program-based testing and specification-based testing can be classified into two groups, according to the way test cases are selected: *deterministic* testing and *probabilistic* testing.

- Testing is *deterministic*[1] when test cases are determined only according to testers experience and intuition.
- Testing is *probabilistic* when test cases are selected randomly, either on a uniform profile test space or according to a probabilistic distribution (statistical testing).

Random test selection is easy, inexpensive and can give good results. However, this method is generally considered as weak, it does not give a good coverage of the test space. However, statistical testing is able to address desired (critical) zone of the test space.

Presently, during the JCI development process, a code review and a deterministic white box testing is performed on each software component, i.e. a part of the global software which provides a specific service. In addition, after mixing these components, the software validation process must check if the global software is compliant to customer requirements. This process includes an automated static analysis approach (using tools such Polyspace) but is mainly based on a deterministic black box testing approach. Software validators must design software test cases in order to detect the maximal number of bugs (and the most frequent and/or dangerous ones) so as to be able to ensure the compliance to the customer requirements. Since the exhaustive set of test cases is usually huge, its size must be reduced while retaining its pertinence. The goal is to select the smallest number of test cases that will detect the greatest number of bugs in the tested program.

For instance, let us consider a functionality from an automotive electronics product.

[1] In software engineering, deterministic is a dedicated term used when we design test cases according to know how. Once test cases are determined, their execution can be repeated.

An excerpt from the way how car manufacturers state their functional requirements is given in Figure 4.

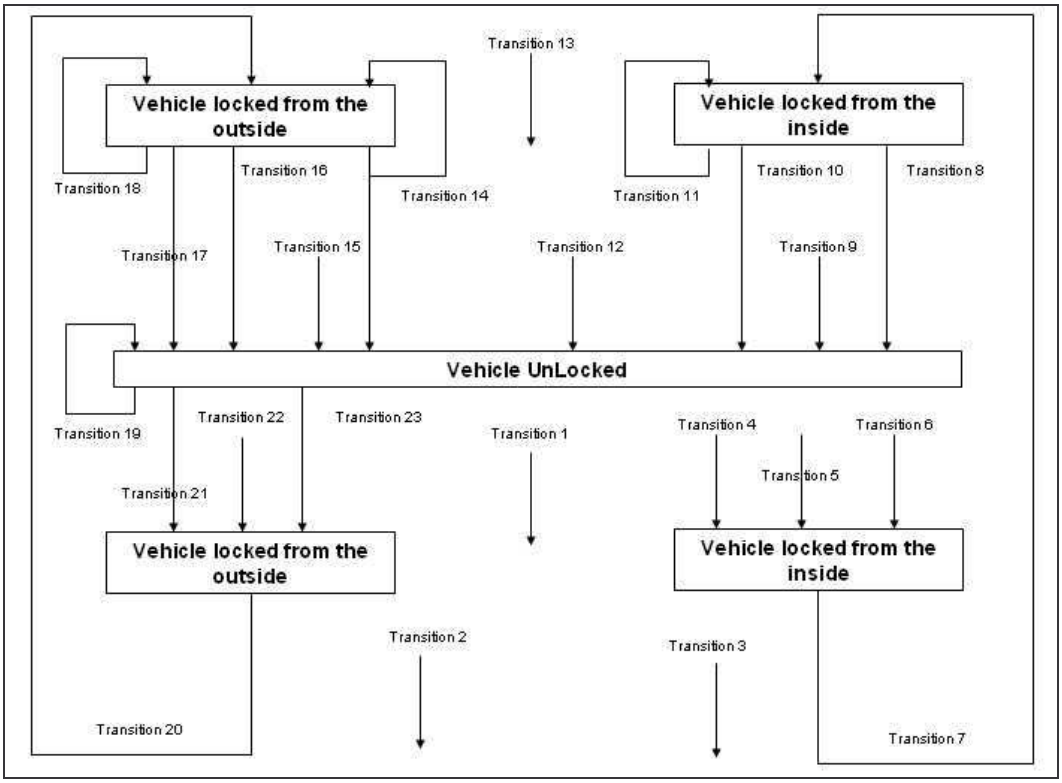

Figure 4 Excerpt from the expected behaviour (functional requirements) of a functionality as stated by car manufacturers

An excerpt from a test case designed for the validation of a functionality is given in Figure 5:

1. In test step 96, validators wait for 500 ms without carrying out any actions on the product and check that the outputs of the product didn't change
2. In test step 97, validators activate the front wiper switch, wait for 200 ms and check that the concerned outputs was activated according to the expected behaviour

Test Step No	Test Actions	Expected Results
…	…	…
96	Test #96 Wait 500 ms	FrontWipingRequest = 0 WipingCommand = 0 WashingCommand = 0
97	Test #97 $FRONT_WIPER_HIGH_SW = 1 Wait 200 ms	FrontWipingRequest = 7 WipingCommand = 3 WashingCommand = 0
…	…	…

Figure 5 Excerpt from a test case as designed by JCI validators

2.4.1 Existing techniques to reuse previous bugs

At JCI Company, two strategies were developed in order to use experience feedback in testing new software product:

- Usually, software defects are saved in a database in order to follow up detected bugs (traceability), extract quality indicators (project management) and capitalize potential reused bugs (reusability). Once a bug is detected on a project, the project leader decides if this bug must be verified on other projects or not. The decision process is not formal and is mainly based

on the experience of the decision maker. In case of a reused bug, this bug is transferred to software lead engineers who confirm or not the possible re-use of this bug. Finally, each reused bug or group of bugs is summarized in a lesson learnt (a textual sentence) to be consulted before beginning a validation stage. The way of describing a bug in the bug database has a major impact on this process. The free fields (cause, problem and solution of the bug) are filled without complying with a standard format and therefore identifying common or similar bugs remains a difficult task.

- Standard test cases (Figure 6) were developed and classified by functionality of product. In fact, potential bugs start to be systematically identified for recurrent product functionalities and documented in standard test case patterns which are systematically consulted (for the given product type) before beginning a validation stage. This is a conventional RETEX (RETurn of EXperience) strategy, but which remains to be completed for any product line. The main difficulty of such an approach is to describe standard test cases with a suitable language level for practitioners.

VPR ID	Type of Test	Date Modified
VPR.SPEED.0001.01	Functional	22.03.2006
Goal	**Initialization of the pointer**	
Applicable if	– The device displays the vehicle speed with pointer	
Description of test	– The Ignition is switched ON. – Set the signal concerning the Speed to value > 0 km/h. – The ignition is switched Off – Set the signal concerning the Speed to value = 0 km/h. – The Ignition is switched ON.	
Expected behaviour	– At the second ignition ON the pointer should be on its stop position.	
Additional Comments	– If the project contains 2 or more product lines (ex. Low line, High line) repeat the tests on both lines.	
TCR reference (base and defect ID)		

Figure 6 Example of a standard test case as developed at JCI Company

2.4.2 Criteria to stop testing

As mentioned before, testing software exhaustively remains a very hard problem. Therefore, testing techniques are often based on specific hypotheses and objectives which help practitioners and managers to decide when to stop testing protocol.

Currently, during the white box testing of software components at JCI Company, practitioners use mainly code coverage (A survey on code coverage based testing tools is done in [8]) as a criterion to stop testing. Code coverage (statements, decision, branches … coverage) is a way to measure how thoroughly a program is covered by a set of test cases. This criterion seems to be relevant since the goal of the testing activity is to check the correctness of the software module. But, this criterion does not directly assess the compliance of the software module to the customer requirements (validation activity); this is a biased indicator. At JCI Company, customer requirements are referenced and managed using professional tools (Doors, Reqtify) and therefore criteria to stop software validation are primarily based on the *coverage rate* of these requirements. Note that time and budget constraints are strongly present in automotive industry. In addition, managers are provided with information on:

- The ratio of bugs detected by JCI Company comparatively to those detected by the client after the product delivery. But, the efficiency is related to a specific customer and gives no idea on the software product quality itself. Indeed, the worst the customer validation process is the best the efficiency of the JCI validation process will be.

Through our literature review, we identified some relevant criteria that can be used to decide if the software is ready to be released or not. These criteria are mainly based on three factors "Cost, quality and time". Among these criteria, let us mention:

- A stop testing criteria based on covering customer requirements in various ways is proposed in [9]. This approach can only be applied on formal or semi formal customer requirements.
- A stopping criterion based on estimated reliability and confidence is presented in [10]. This criterion relies on reliability and cost target and needs an appropriate data collection on detected bugs.
- A Cost-benefit stopping criteria based on estimates of the faults remaining in the system, the cost to repair them both before and after release, and the customer dissatisfaction is presented in [11]. This stopping rule sounds interesting, although the prediction of the number of bugs still non relevant in software industry.

3 TOWARD A NEW APPROACH TO DESIGN EFFICIENT TEST CASES

In spite of the different activities of software testing and bugs capitalization performed at JCI, after each customer intermediate delivery and even on launched product, some bugs are detected by customers and users. In accordance with JCI context and needs, our goal is to develop a new approach able to efficiently design test cases and monitor the software validation process. The main expected advantages of this approach are:

- Ensure the coverage of functional customer requirements (and not only a coverage of code lines, which is more a constraint on means)
- Use at best the experience feedback (capitalized bugs in database) on probable frequent bugs on similar products so as to avoid the creation of these bugs during the software design stage and to facilitate their detection during the validation stage (by the design of appropriate tests). Once a bug is detected, the retrieval of a similar bug in the bug database may also accelerate the bug correction in providing adequate solutions.
- Test the software product with the knowledge of user profiles so as to maximize the probability to detect a real life bug that would be detected by the client and to be able to associate a degree of gravity related to the client (driver or passenger) safety.

Beyond, a particular software design and validation project, we should also be able to deduce quality indicators for a project so as, in the medium and long terms, to be able to monitor the activity and continuously improve its quality (Design for Six Sigma methodology is envisaged).
Consequently, we focused our research on three elementary issues: functional customer requirements modeling and coverage, software defect classification and analysis and user profile definition.

3.1 Functional customer requirements: Modeling and coverage

As mentioned above, JCI customers express their requirements differently. Some customers use semi formal methods to express part of their requirements; others use natural language. In [12], three ways of modeling functional customer requirements have been identified:

- *Informal modeling*: using a language with neither syntax nor a semantic. For example, the natural language.
- *Semi formal modeling*: using a language with syntax but a poor semantic. For example, "transition systems" (state/transition diagram, statechart, Markov chain …). Design test cases from semi formal requirements are called model based testing.
- *Formal modeling*: using a language with defined syntax and semantic. For example, Z language, B language, VDM approach, SDL approach. Design test cases from formal requirements are called formal testing.

In our context, customer requirements are usually informal or semi formal. Formal modeling is specific to critical systems when human life is concerned (avionics, railways, aerospace …).
In fact, modeling functional customer requirements with a generic semi formal model (based on transition systems and decision tables) helps us to:

- Check the consistency of requirements (Model checking)
- Standardize the requirements representation

- Automate the test cases design
- Measure the coverage of the requirements

Moreover, since each user's functionality in an electronic product (for instance, "front wiper" functionality in a body controller module) has a number of inputs, outputs and internal variables, we can assume that the functional behaviour of such functionality can be modelled using transition systems and decision tables. The inputs and outputs of the functionality can be associated respectively to the transitions and states of the transition systems.

In addition, we performed a study on a large number of functional requirements provided by several JCI customers and we came out to the conclusion that "transition systems" and "decision tables" (semi formal method) are more and more used in specifying functional requirements (Figure 7). Therefore, this type of modeling appears to be the most adapted one to our context.

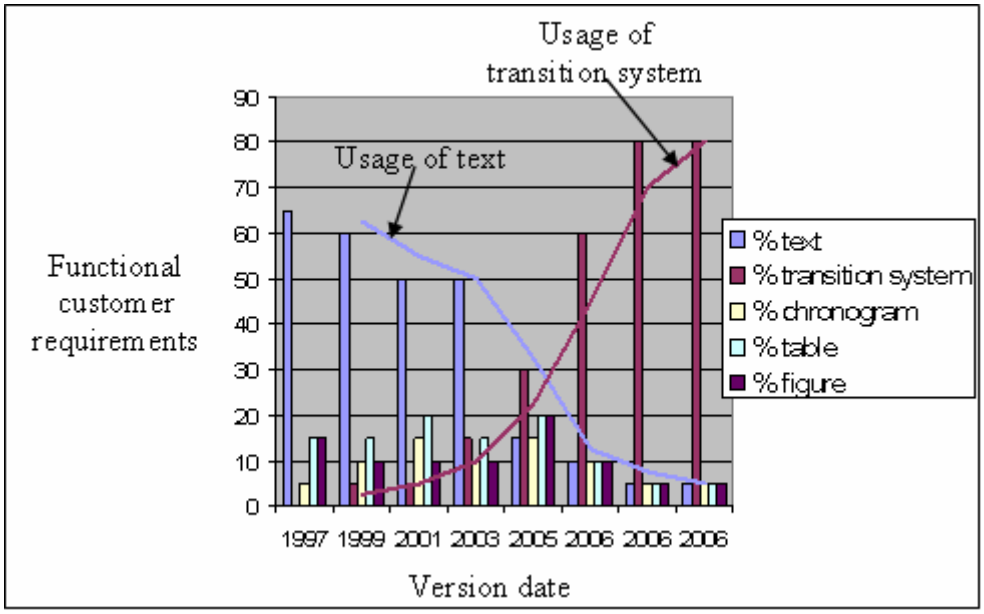

Figure 7 Growing part of a more formal expression of functional customer requirements

3.2 Experience feedback: Software defect classification and analysis

Many bug's characteristics might be relevant for analysis. Bugs are introduced due to a particular reason into a particular piece of software at a particular time. The bugs are detected at a specific time and context by noting some sort of symptom and they are corrected by appropriate means. Each of these aspects could be relevant for a measurement and analysis purpose.

In order to extract knowledge from the JCI database of bugs, three necessary stages have been developed in [13]:

1. *Data selection and pre-processing*: In JCI bug's database, each bug is described by 121 free and predefined fields. 75% of these fields are filled by practitioners and 25% of filled fields are free fields. Fundamental for successful data mining activities, data has to be selected and cleaned regarding to a classification objective. In fact, some data can be unnecessary for our study, other interesting data can be inexistent (Figure 8). We select from existing bug attributes those which contribute to improve the quality of the software validation process: Product, Project, Version, Detection date, Problem resume, Problem description, Cause resume, Cause description, Origin phase, Detection phase, Detection mean, Nature, Gravity, detection frequency. Other types of information that could be useful would be worthy to be documented. This fact is not so dramatic since we have analysed that these lacking fields may be reconstructed from the present data via expert rules. This is the case of the four following data:
    - the customer functionalities,
    - the software components impacted by the bug,
    - the causes and
    - the problems from software user and developer viewpoints.

	Bug attributes		Utility	Comments
	Already exist	To create		
1	Origin phase		Yes	Development phase where the bug was integrated into the software
2	Gravity		Yes	Customer and user impact
3	Cause resume		Yes	Free field to describe the cause of the bug
4	Correction date		No	Correction date of the bug – not in our scope
5		Cause type	Yes	Identify recurrent causes
…	…	…	…	…

Figure 8 Bug attributes selection and pre-processing

2. *Data mining*: This stage involves the data mining algorithms and techniques commonly used to produce patterns and extract interesting information from selected data. Among these techniques, let us mention: classification trees, association discovery, clustering, and Bayesian networks.
3. *Interpretation and evaluation*: Extracted information from previous stages has to be assimilated by experts in order to be transformed into useful knowledge. This knowledge contributes to:

- Improve the design of test cases by identifying risky zone (in the test space) which can be illustrated by a list of validation recommendations or rules (check-lists, standard test cases, highlighting of functional requirements)
- Improve the bugs database by developing a typology of cause and problem as it was done in [14] on a database of cost saving ideas in aeronautic domain.

3.3 User profiles: Definition and implementation

The reliability of a software system depends on how we will use it. Making a good reliability assessment or prediction depends on testing the product as if it were in the field. The operational profile, a quantitative characterization of how the software will be used, is therefore essential.
A profile is a set of independent possibilities called elements, and their associated probability of occurrence. If operation A occurs 60 percent of the time, B occurs 30 percent, and C occurs 10 percent, the profile is [A, 0.6...B, 0.3...C, 0.1]. The operational profile is the set of independent operations that a software system performs, their succession probabilities and their probability distributions of time inter-operation. In order to evaluate the feasibility of operational profiles on automotive electronic products, we consider the example of a body controller product. In fact and according to Musa [15], developing an operational profile for a system involves one or more of the following five levels:

1. *Client type list*: A client is the individual, group or organization that is purchasing the software system. In response to a car manufacturer (customer) request, JCI develops an electronic product and sells it only to the customer who asked for it. Therefore, in our context, we consider that software system is dedicated to only one customer (car manufacturer) and a client type list is not necessary.
2. *User type list*: A system user may be different from the client of a software product. A user is a person, group, or institution that operates, as opposed to acquires, the system. According to JCI experts, the users of body controller products can be classified regarding to criteria such job, climate, sexe, age, culture …
3. *System modes*: A system mode is a way that a system can operate. Most software systems have more than one mode of operation. A body controller product has 3 operating modes: normal mode, factory mode and diagnostic mode. Note that the occurrence probability of the normal mode is about 99% and consequently the other modes can be ignored in defining operational profiles.
4. *Functional profile*: The next step is to break system modes down into the functions it needs, to

create a function list in determining each function's occurrence probability. A list of 31 customer functionalities were identified for the normal mode of a body controller. The best source of data to determine occurrence probabilities is usage measurements, i.e. frequency measurements of the users operations, taken on the last release or on similar system. In our context, we don't have this type of information (since it is considered as confidential by the clients) and therefore the process of predicting the use (for example, a survey or a qualitative assessment) for each couple of user type/functionality is extremely important.

5. *Operational profile*: Finally each functionality is mapped onto operations. It can be noted that operations may be involved in performing different functions. Experts must identify for each customer functionality a list of operations and therefore define their succession probabilities and their probability distributions of time inter-operation.

The development of operational profiles for a product will contribute to:

- Improve the design of test cases by identifying potential operations of users on the product and therefore develop a list of validation recommendations or rules (check-lists, standard test cases, highlighting of functional requirements).
- Assess the reliability of the software product by providing information on bugs detection under a specific user profile. This assessment can help managers to decide if the software is ready to be released or not.

4 CONCLUSIONS AND PERSPECTIVES

Through our global approach, we were able to identify three improvement fields in the automotive software validation process: customer requirement reviews, experience feedback management and user profiles definition. Solutions proposed in the literature may be satisfactory under given isolated software testing issues, but appear to be not adapted to practical industrial constraints. Therefore, we have proposed founded recommendations and specifications for a new consistent test design approach able to efficiently design test cases and monitor the software validation process.

In a long term perspective, we aim at fixing a set quality indicators for the software validation process. In that manner, it could be envisaged to better monitor the activity of validation along the design projects. The "Six Sigma" approach appears to be relevant since customer-oriented statistical information can be raised on process milestones (customer deliveries). Within the "Six Sigma" framework, the Define, Measure, Analyze, Improve and Control (DMAIC) methodology is undoubtedly an appropriate approach. DMAIC, which refers to a data-driven quality strategy for improving processes, would be the embedding system for managing the test design quality.

REFERENCES

[1] Harel D. Statecharts: a Visual Formalism for Complex Systems. *Journal of Science of Computer Programming*, 8, 1987, pp. 231-274.

[2] Object Management Group (OMG). Unified Modeling Language : Superstructure, version 2.0, 2005.

[3] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications (SRS), 1998.

[4] Péraire C. Formal testing of object-oriented software: from the method to the tool. *PhD report*, Computer department, Ecole Polytechnique Fédérale Lausanne, 1998.

[5] Beizer B. *Software Testing Techniques, second edition*, 1990. (Van Nostrand Reinhold).

[6] Martin R. W. and Michael A. H. Strategic benefits of software test management: a case study, *Journal of Engineering and Technology Management*, 2005, 22(1-2), 113-140.

[7] Marre B., Thévenod-Fosse P., Waeselynck H., Le Gall P. and Crouzet Y. An experimental evaluation of formal testing and statistical testing. In *Safety of Computer Control System, SAFECOMP '92,* Zurich, Switzerland, 1992, pp. 311-316 (Heinz H. Frey edition).

[8] Yang O., Jenny Li J. and Weiss D., A Survey of Coverage Based Testing Tools, In *international workshop on Automation of Software Test, AST '06*, Shanghai, China, May 2006, pp. 99-103.

[9] Offutt J., Xiong X. and Liu S. Criterion for generating specification-based tests. In *the Fifth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS '99*, Las Vegas, October 1999, pp. 199-131 (IEEE Computer Society Press).

[10] Dalal S. R. and Mallows C. L. When should one stop testing software? *Journal of the American Statistical Association*, 1988, 83(403):872-879.
[11] Chávez T. A decision-analytic stopping rule for validation of commercial software systems. *IEEE Transactions on Software Engineering*, 2000, 26(9):907-918.
[12] Fraser M.D., Kumar K. and Vaishnavi V.K. Strategies for Incorporating Formal Specifications in Software Development. *Communications of the Association for Computing Machinery (ACM),* 1994, 37(10), pp. 74-86.
[13] Ben Ahmed W. Safe-next: a systemic approach for knowledge discovery in databases. Application in accident scenario development and interpretation. *PhD report,* Industrial Engineering laboratory, Ecole Centrale Paris, 2005.
[14] Angéniol S., Yannou B., Longueville B. and Chamerois R., Supporting saving ideas reuse with an ontology based tool. In *ASME International Design Engineering Technical Conferences & the Computers and Information in Engineering Conference, IDETC/CIE' 2006*, Philadelphia, September 2006.
[15] Musa J. D. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 1993, 10(2), pp. 14-32 (IEEE Computer Society Press).

Contact: Roy Awédikian
Johnson Controls Automotive Group
18, chaussée Jules César
BP 70340-Osny
F-95526 Cergy-Pontoise Cedex
France
Phone: +33 1 3017-5099
Fax: +33 1 3017-6445
E-mail: roy.awedikian@jci.com