INTERNATIONAL DESIGN CONFERENCE - DESIGN 2008
Dubrovnik - Croatia, May 19 - 22, 2008.

DESIGN 2008

A RADICAL IMPROVEMENT OF SOFTWARE BUGS DETECTION WHEN AUTOMATING THE TEST GENERATION PROCESS

R. Awedikian, B. Yannou, P. Lebreton, L. Bouclier and M. Mekhilef

Keywords: test design, model based testing, software testing, software quality management, software reliability engineering

1. Introduction

Nowadays, car electronic architectures become more and more complex and carmakers outsource the design of electronic modules to automotive electronic suppliers. The software part of these modules accounts for more than 80% of the total number of defects detected on these modules. In software development [Bertolino 2007] and as illustrated in [Awedikian 2007], a loop-type design process is initiated between the carmakers and their suppliers. After each delivery, carmaker detects some "bugs" (called also software defects). 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 become dramatic for the electronic supplier in financial terms. Therefore, finding bugs earlier in the development cycle and reducing the number of bugs detected by carmakers and end users is a priority for automotive suppliers. In fact, the number of bugs detected later in the development cycle is one of the major metrics used by carmakers when choosing their suppliers.

In this paper, we start by a brief presentation of the software development cycle in the automotive electronic supplier (Johnson Controls). A detailed analysis of the present industrial approach to design tests for software functional testing can be found in [Awedikian 2008]. Nowadays, testers design manually test cases which is strongly dependent on the experience of these testers. In the second section, we develop a new approach and a corresponding software platform to improve the design of test cases. It is mainly based on modelling software specifications, focusing on critical tests to be done (because of their higher probability to reveal a bug) and monitoring the automatic generation of tests by quality indicators. The simulated software specifications model has already been presented in [Awedikian 2008]. In the third section, we define a framework to model the behaviour of a driver using a specific software functionality. We propose, in the fourth and fifth sections, to reuse capitalized software defects and test cases on a specific functionality to design efficient tests. Therefore, in the sixth section, we develop a framework in order to superimpose on the specification model some statistical data concerning the drivers' behavioural profiles and returns of experience in terms of bug detection for similar products. This enrichment of the specification model allows driving the generation of relevant tests in terms of their probability to reveal existing bugs. In the seventh and eight sections, we specify the automatic tests generation and propose a set of quality indicators to monitor and assess the quality of generated tests and for deciding when to stop the tests. In particular, we propose an indicator of functional specifications coverage which is an important contributor to the perceived quality by the carmakers. Lastly, we compare the results of an industrial case study, in terms of quality and efficiency, where both the conventional and our new test design approaches have been applied.

2. Software development cycle

At Johnson Controls, the lifecycle of a software product is divided into 5 global stages (see figure 1): Request For Quotation -, Design, Design Validation, Production Validation and Production.

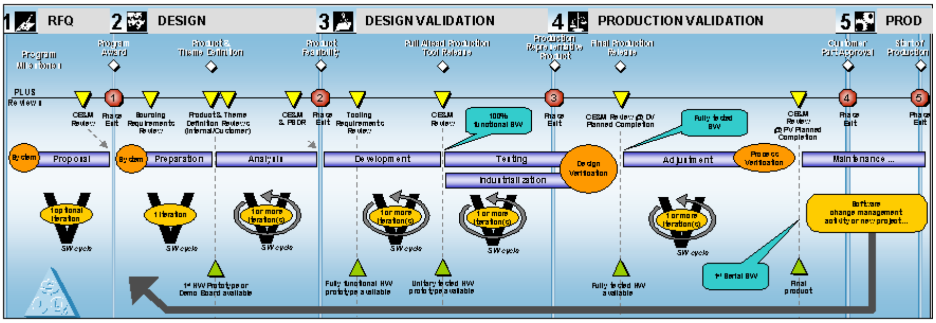


Figure 1. Overall software product lifecycle **(this figure is voluntarily fuzzyfied for confidentiality reasons)**

Within each stage, engineering activities are performed according to the standard V-cycle of the software industry (see figure 2) and in an iterative way in order to take the carmaker constraints and requirements priorities into account. The main engineering processes are *Requirements specification and management*, *Global design*, *Component development*, *Integration* and *Validation*.

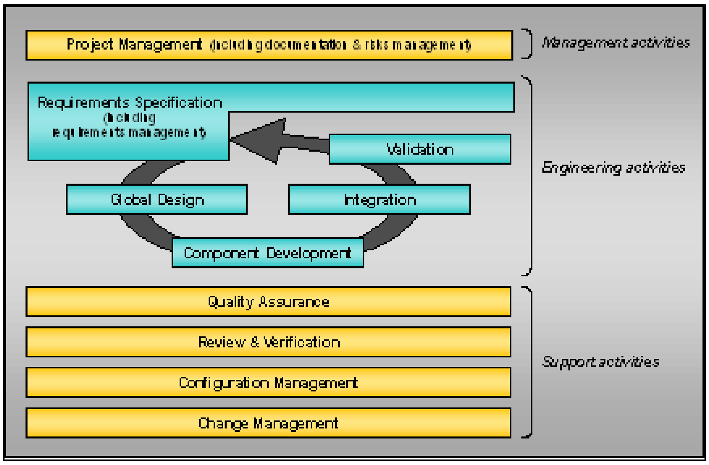


Figure 2. Elementary V-cycle for any of the global stages of the software lifecycle

We notice that each of the development, integration and validation processes perform software testing activities in order to verify and validate the correctness of the software delivered at the end of the process. In [Beizer 1990], the author discussed most software testing techniques (black and white box testing). Software defects are detected in each of these processes, analyzed, corrected and capitalized in the huge bugs database of the company.

The software testing activity consists in:

1. *Analyzing carmaker requirements*: testers who need to design tests, must first read, analyze and understand carmaker requirements.
2. *Designing test cases*: presently, testers proceed to a manual design of test cases. The performance of this activity is mainly based on the experience of the testers. The reader can find in [Awedikian 2008] more details on how testers presently design test cases.

3. *Simulating test cases* on software product and detecting software defects: once test cases are developed, they are simulated on the software product in order to check that they are ok and as free of bugs as possible.

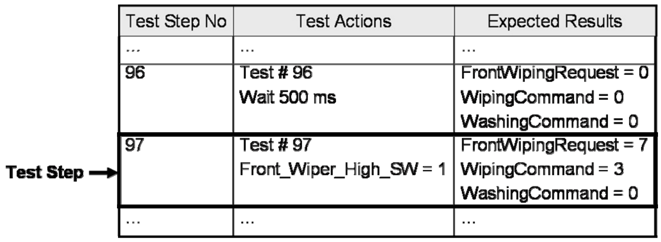
What is a “client functionality”?

A “*functionality*” is a set of services delivered by the software product. A functionality is specified by a set of inputs, outputs and a set of requirements. A “*client functionality*” is a functionality that delivers service to the clients (carmakers and/or drivers). For example, the door lock management functionality.

What is a “Test Case”?

Let’s consider a client functionality with two input signals: *I1(*with domain *D(I1)={0,1}) and I2 (D(I2)={1,2,3}).* We first call “*operation*”, the fact that an input signal is set to a value. For example, I2=3 is an operation. A “*test case*” is the succession of *k* operations separated by time intervals and the expected results on output signals after each operation.
An excerpt from a test case designed is given in Figure 3:

1. In test step 96, testers wait for 500 ms without carrying out any actions on the product and check that the outputs of the product haven’t changed.
2. In test step 97, testers activate a switch, wait for 200 ms and check that the concerned outputs are 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
		FrontWipingRequest = 7
		WipingCommand = 3
		WashingCommand = 0

...	...	...

Figure 3. Excerpt from a test case (two operations) as designed by JCI testers

3. A new platform for automating software tests generation

Our new platform of automated software test generation presents a much different workflow for generating test series than the present one (see figure 4). The new workflow is based on six activities which are manual, semi-automatic or automatic and managed by different individuals (requirement engineers and testers). These activities are :

1. Represent the carmaker functional requirements in our unique model of functional requirements
2. Define some behavioural characteristics of a car driver when using the tested client functionality.
3. Perform a statistical analysis on bugs and test cases respectively detected and developed in the past on the same functionality.
4. Highlight the relevant, critical and mandatory test cases to be chosen from the test design space of the client functionality.
5. Automate the generation of test cases from the enriched model (by stages 2 to 4) of functional requirements.
6. Manage the test generation with cost, delay and quality indicators.

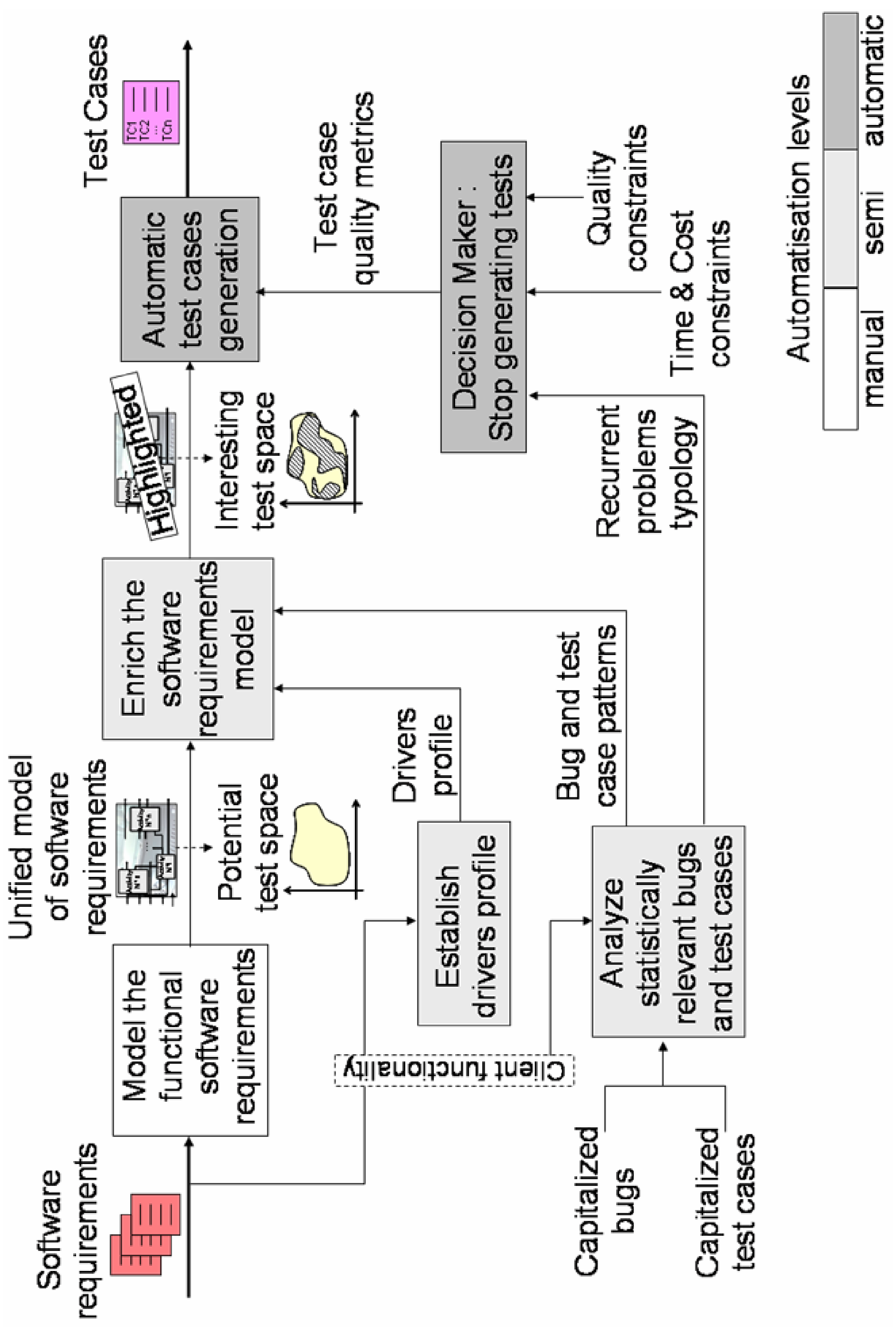

Software requirements
Model the functional software requirements
Unified model of software requirements
Potential test space
Enrich the software requirements model
Highlighted
Interesting test space
Automatic test cases generation
Test Cases
TC1
TC2
...
TCn
Test case quality metrics
Decision Maker : Stop generating tests
Quality constraints
Time & Cost constraints
Recurrent problems typology
Bug and test case patterns
Drivers profile
Establish drivers profile
Analyze statistically relevant bugs and test cases
Client functionality
Capitalized bugs
Capitalized test cases
Automatisation levels
manual
semi
automatic

Figure 4. Our new approach to automatically design relevant test cases

In [Awedikian 2008], a detailed presentation of our unified model for modelling and simulating the software functional requirements can be found. The major concepts (input, intermediate, output signals, elements, decision table, finite state machine, and clock) are introduced, the verification and validation activities and the functional simulation of this model are presented. In the following, we further detail the data enrichment of this functional specification models and the resulting generation of relevant test cases for revealing as many bugs as possible.
In [Utting 2006], we can find a survey on model based testing techniques and tools.

4. Definition of user (driver) profiles

We analyzed a set of software defects detected by carmakers and by end users (drivers) and we came up to the conclusion that some of these bugs need specific tests to be detected. These tests must simulate operations regularly done by the final user of the product. The literature review [Musa 1993] reveals that it is possible to estimate the reliability degree of a software product by analyzing the bugs occurrence when testing with a user approach. In fact, making a good reliability assessment or prediction depends on testing the product as if it were in the field. A stopping criterion based on estimated reliability and confidence is presented in [Dalal 1988]. This criterion relies on reliability and cost target and needs an appropriate data collection on detected bugs. We develop three types of constraints that engineers can affect to an input signal in order to, when generating automatically test cases, eliminate or favour specific "successive" operations. These constraints aim to reduce the number of possible combinations on input signals and to more thoroughly pinpoint which ones have a high potential to detect bugs. These three constraints are:

- *Logical constraint:* this constraint forbids that an input signal switches between inadequate values from a use point of view. That is why, we classified input signals into two types:
    - Acyclic: let's consider the signal *X*, which has a domain *D(X)={1,2,3}*. An input signal is acyclic if, at any moment, all the operations (*X=1* or *X=2* or *X=3*) on the signal are possible.
    - Cyclic: let's consider the signal *Y*, which has a domain *D(Y)={1,2,3}*. An input signal is cyclic if future operation (*X=1* or *X=2* or *X=3*) on the signal depends on the one did in the past.
- *Succession constraint:* in practical use of an electronic product, two or more operations must intuitively succeed. For example, close the driver door and switch on the car engine. Through this type of constraint, we favour such successive operations.
- *Conditional constraint:* this constraint characterizes a specific user behaviour between two or more correlated input signals. In other words, when one or more inputs fulfil specific conditions, the domain of other inputs is automatically adapted.

5. Statistical analysis of capitalized bugs and test cases

Using capitalized bugs and test cases seems to be beneficial in automotive context since more than 50% of functionalities performed by software product are common to any series of cars. Therefore, when testing a client functionality that we already implemented in the past on another project, it is judicious to:

- *Avoid recurrent type of software defects*. In fact, when testing a client functionality, we classify detected bugs from two points of view:
    - <u>Functional classification:</u> this type of classification enables to identify critical functional requirements of a client functionality where software developers are liable to introduce bugs. Consequently, these requirements must be tested mandatorily.
    - <u>Structural classification:</u> we have defined bug categories adapted to efficient retrievals of similar (related to the same cause) bugs. Consequently, we can better address the problem of detecting bugs by generating specific tests to check the non-existence of recurrent bugs in the software under test. After a literature review on bugs classification and root cause analysis (see [Freimut 2001]), we identify a set of 8 bug categories well adapted to our context: *requirements bugs*, *control flow and*

sequencing, *processing*, *data*, *code implementation*, *integration*, *system and software architecture*, *test definition or simulation bugs*. The result of such a classification help testers to establish stopping test criteria based on quality objectives and, in consequence, to generate test cases with a high probability to detect recurrent bugs.

- *Reuse existing test cases.* In a software organisation, test cases management and reuse are one of the main characteristics of a high maturity level. For a client functionality under test, our new testing platform is able to analyze test cases developed in the past for similar functionalities and identify recurrent and critical test scenarios. Consequently, when generating a new set of test cases for this functionality, we reduce the test design space by focusing on these scenarios based on our returns of experience.

6. Enrichment of the specification model with knowledge on driver recurrent operations, critical requirements and testers experience

Once constraints on inputs are defined for the client functionality under test (see section 4), ancient bugs and existing test cases which related to previous projects of similar products are statistically analysed (see section 5). In order to enrich the specification model, we propose to set probabilities between all possible successive operations of a client functionality. To do so, we build a matrix that we name "Operation Matrix" which is a square matrix with all possible operations in columns and in rows. Between the two operations of a pair we define:

1. The probability that two operations are in sequence.
2. The time between two operations, modelled as an interval of possible values (a uniform probability)

Let us consider a client functionality with 3 input signals: *I1* (with domain *D(I1)={0,1}*), *I2 (D(I2)={1,2,3}*) and *I3 (D(I3)={0,1})*. The "operation matrix" associated to this example can be seen in figure 5.

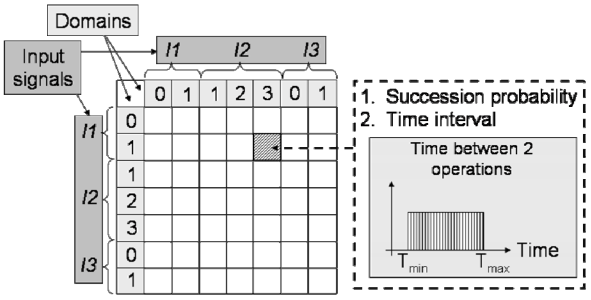


Figure 5. Operation matrix

7. Automatic generation of test cases

Once the simulated model of software requirements is ready and "Operation matrix" are established, generating automatically a test case requires to generate a set of test steps until the stopping criteria are reached. Two automated activities are necessary to generate a test step:

1. Perform a Monte Carlo simulation on "Operation matrix". Two steps are required:
   - *Step 1:* an operation is chosen according to the probabilities on successive operations. For the first time, an operation is randomly chosen. In software testing [Marre 1992], this technique is known under the statistical testing technique.
   - *Step 2:* the inter-operation time is also randomly chosen within the time interval.
2. Simulate the functional requirements model and assess the expected values that must be checked on output signals. In [Awedikian 2008], we develop in details the functional simulation mechanism of our unified model.

8. Monitoring the test case generation by a set of quality indicators

Testing software exhaustively remains a very hard problem. Therefore, software testing must often be based on specific assumptions and objectives which help practitioners and managers to decide when to stop testing protocol. Therefore and in order to monitor the test case generation, we define some indicators related to the quality of a test case. In our approach, testers can generate test cases according to their objectives in term of:

- *Structural (code) coverage* (see figure 6): the coverage rate of statements (lines of code), procedures, conditions (control flow) and decisions in the software product under test. A survey on code coverage based testing tools is done in [Yang 2006]. This is the solution currently used in JCI company.

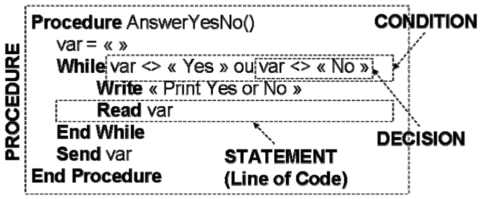


Figure 6. Structural (code) coverage indicators

- *Functional coverage*: A stop testing criterion based on covering software specification was proposed in [Offutt 1999]. He primarily discussed the transitions coverage of a graph-based specification. In our approach (see figure 7), we consider the coverage rate of functional requirements (Decision Table and Finite State Machine elements) but also the coverage rate of signals (inputs, outputs, intermediates) domains and operation matrix (successions between pairs of operations visited). Such indicators help practitioners and managers to assess the quality of the software under test.
- *Tests cost*: Presently, in automotive industry, the time and money spent to test a software product is the major criterion to stop testing. In our approach, since we automatically generate test cases, we only consider the time and money spent to simulate the generated tests on the software product. In [Chavez 2000], the author discusses a cost-benefit stopping criterion. It is based on estimating the defects remaining in the system, the cost to repair them both before and after release, and the carmaker dissatisfaction. As a perspective, it seems interesting to use a more advanced cost model of tests and bugs in order to improve our model of stopping criteria.

In definitive, we have developed a panel interface to allow the test designer to set precise targets on the three subsets of quality indicators (15 indicators). The quality indicators are most of the time expressed in terms of ratios of coverage and, then, are normalized indicators which aim to reach a value of 100%. During one test generation session, the targets may be completed following different orders and the first target completed does not immediately stop – this is not a hard limit – since we stop only when the aggregated preference *(F)* of these indicators *(Q)* has attained a minimal value:

$$F = \sum \left| Q_{Target} - Q_{Current} \right| \times p_i$$ with p_i being a set of weights.

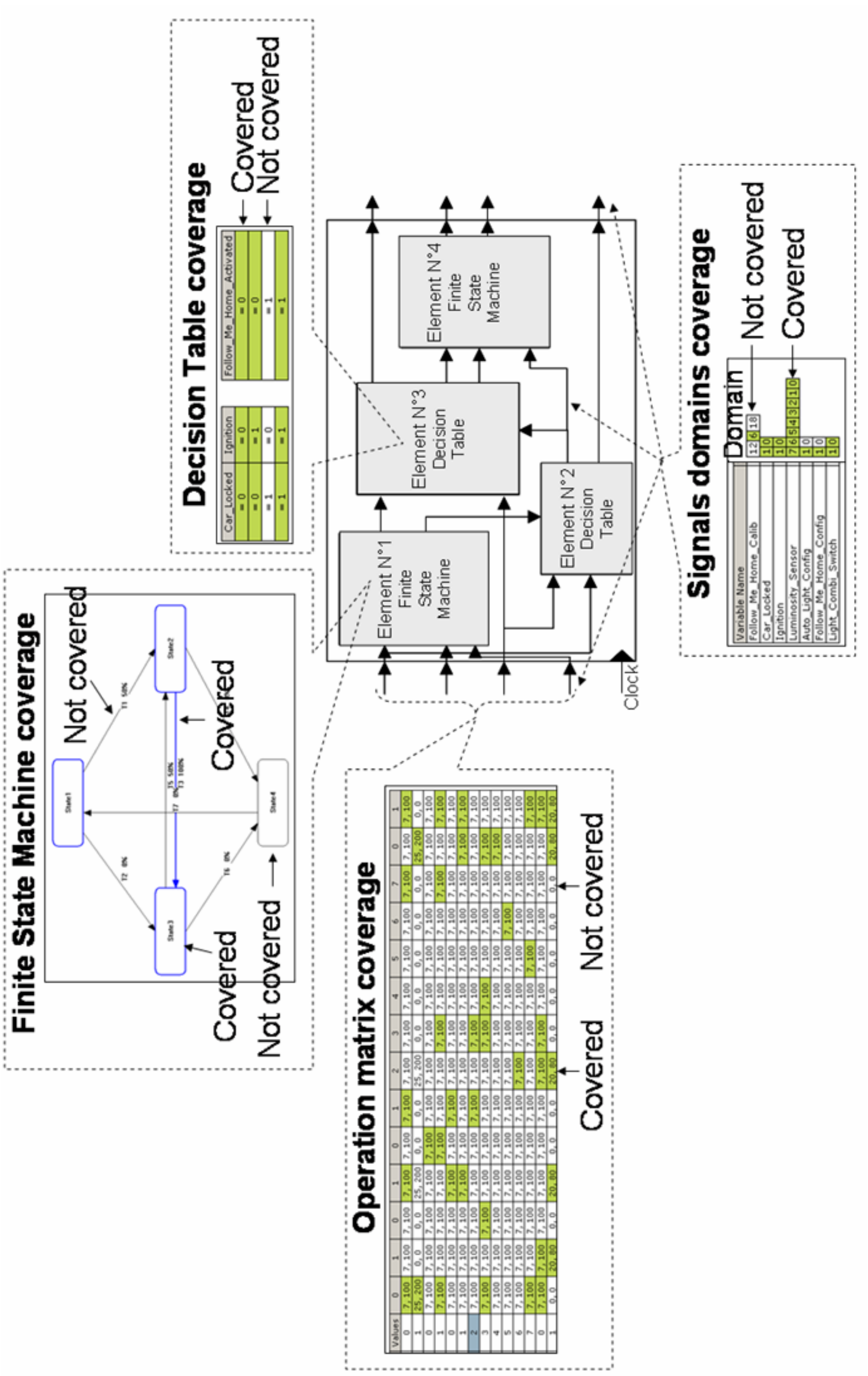


Figure 7. Functional coverage indicators9. A case study to validate our new approach

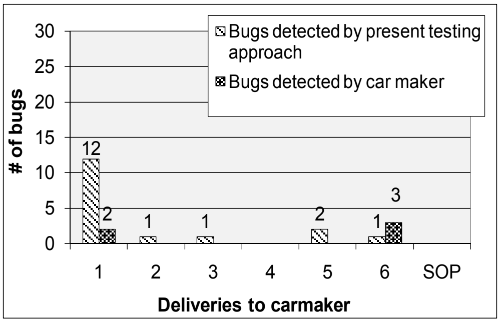
Let us consider a practical software testing problem. It consists in verifying that the developed software product is compliant to the carmaker software functional requirements. The studied client functionality consists in managing the front wiper in a vehicle. This functionality is implemented with other functionalities in an electronic product, named body controller module.
In our case study, we isolate the software component which fulfils the front wiper functionality and we test it through our approach. This component is made of 1229 Lines of code (blank and comment lines excluded), 18 input signals and 8 output signals. This functionality has already been tested with the conventional approach of manual test design in Johnson Controls and 22 software defects were detected until the last carmaker delivery of the software product. 17 bugs were detected by the Johnson Controls software testing processes and 5 bugs by the carmaker after intermediate delivery. In the figure 8, we show the distribution of bugs detected on each intermediate carmaker delivery. It must be noted that, after developing the software product for the first time, only 12 bugs were detected during the first software testing stage. Therefore, a delivery was performed and the carmaker immediately detected 2 more bugs. In the meantime and before the second carmaker delivery, testers tried to improve their test cases and designed some new tests. In consequence, they have been able to detect one more bug and after the second intermediate carmaker delivery, no new bug was detected by the carmaker. The complete scenario of bugs detection until the last carmaker delivery is summarized in the histogram of figure 8.
We estimate the time spent to analyze carmaker requirements, design and simulate test cases and manage internal and carmaker detected bugs. We note that more than 50% were spent to manually design test cases and 10% to manage bugs detected by carmakers. Approximately 54 eight-hour days were spent to fully test the front wiper functionality using the current Johnson Controls testing process.
Our new testing approach is much different and leads to notably improved results. We start with the first version of the software component before the Johnson Controls testing phase. We first model the software functional requirements of the front wiper functionality, then we design automatically two operation matrix:

- A "nominal" operation matrix which allows all successions of two operations with the same succession probabilities. According to experts, we define one standard time interval and we affect it to all successive operations.
- A "driver profile" operation matrix which eliminates successive operations not used by end users (drivers) and favours specific recurrent operations with specific time interval.

Therefore, we define a test plan that we perform on the first version of this software component:

1. Firstly, we generate six test cases from the "nominal" operation matrix with the objective of covering at 100% the input, output and intermediate signals domains, the functional requirements in decision table and finite state machine elements and the operation matrix.
2. Secondly, we generate six test cases from the "driver profile" operation matrix, with the objective of covering at 100% the operation matrix.

Figure 8 illustrates the bugs distribution in the case where our approach would have been applied since the beginning. 16 bugs over the 17 bugs (94%) which were detected by Johnson Controls all along the 6 testing stages have been directly detected here. In addition, 3 bugs over the 5 bugs (60%) detected by the carmaker along the carmaker deliveries have been directly detected by our platform. Moreover, we have also detected 5 "minor" bugs that were not detected by Johnson Controls nor by the carmaker, and which were, in definitive, delivered to the carmaker. In addition, we have analyzed the three bugs that our approach didn't detect and we come up to the conclusion that these bugs could be detected by our approach since we reach a 100% of functional coverage. These non-detected bugs are related to specific functional requirements that weren't covered by our generated test cases. Indeed, when generating tests, our computational algorithms didn't succeed to reach 100% of functional coverage (maximum of 85%). To overcome this lack, we plan to improve our computational algorithms in order to focus on covering the non-covered zones of the specification.



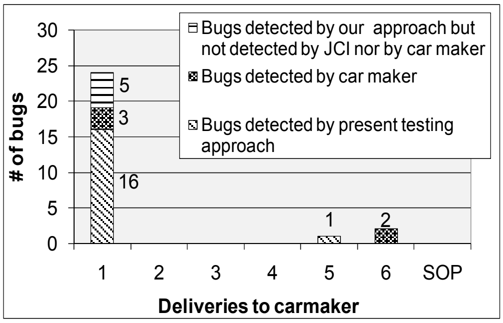


Figure 8. Bugs occurrence when using our approach to design test cases vs the conventional one

In addition, the results of this case study confirm the necessity to generate tests with a driver profile strategy. Indeed, through the test cases generated from the "nominal" operation matrix, we detect 18 bugs and 6 more bugs were detected when simulating the test cases generated from the "driver profile" operation matrix.

As a conclusion of the case study:

- We *increase by 100%* the number of bugs detected since the first testing phase (from 12 to 24 bugs)
- We *decrease by 60%* the number of bugs detected by the carmaker (from 5 to 2 bugs)
- We *increase by 40%* the number of bugs detected by JCI (from 17 to 24 bugs)
- We *detect about 20%* of new bugs (5 over 22 bugs). These new bugs are "minor" and they were not detected by JCI nor by the carmaker
- We *reduce by 25%* the time spent in testing the software (from 54 to 41 eight-hour days)

In fact, across our new software testing approach, the image of Johnson Controls in front of carmakers will be improved and as a direct impact, the number of tenders will grow.

10. Conclusions and perspectives

In this paper, we propose an approach to design efficient and intelligent test cases for software product. The basic of this approach is to represent formally software functional requirements and to reduce the test design space by focusing on important and critical tests to be done. The tests generation is automated and monitored by quality and cost objectives. Indeed, testers can generate test cases that fulfil a predefined set of objectives (in terms of functional requirements coverage, code coverage and test cost). A case study on historical data has also been developed in this paper and potential benefits have been highlighted.

Some managers and practitioners were interested in implementing our software testing approach as a long term solution in their business units. Moreover, we were asked to implement some parts of our approach in the testing process of a new product scheduled for 2009. Consequently, we plan to strongly validate our approach, first by demonstrating the benefits of our approach through historical data (see the case study in this paper) and second by integrating our approach into the global design process of the company (implementation on future projects). To do this, we plan to develop a second larger case study on historical data from another type of product and client functionality. Then we have to assess on our approach statistical properties such as robustness, repeatability and reproducibility. We also plan to develop a model of software defect cost which takes the phase in which the bug was detected into account, its severity and occurrence, the cost and time spent to manage it and the impact on Johnson Controls image, carmaker and end user. This detailed cost model will allow estimating the return on investment (ROI) if using our proposed approach to test software products. Finally, we have to manage the change of the testers practices and activities. Indeed, testers technical skills will have to switch from a manual design to a high level modelling of test scenarios and objectives in using in a flexible manner our design platform.

References

Software Testing Research: Achievements, Challenges, Dreams. Antonia Bertolino. In Future of Software Engineering, 29th International Conference on Software Engineering, May 2007.

Awedikian, R., Yannou, B., Mekhilef, M., Bouclier, L., Lebreton, P., "Proposal for a holistic approach to improve software validation process in automotive industry", Proceedings of the 16th International Conference on Engineering Design - ICED 2007, Paris, 2007, pp. 695-696.

Awedikian, R., Yannou, B., Mekhilef, M., Bouclier, L., Lebreton, P., "A simulated model of software specifications for automating functional tests design", Proceedings of the 10th International Design Conference - DESIGN 2008, Dubrovnik, 2008. (Submitted)

Beizer B. Software Testing Techniques, second edition, 1990. (Van Nostrand Reinhold).

M. Utting and B. Legeard. Practical Model-Based Testing - A Tools Approach. Morgan and Kaufmann, 2006.

Musa, J. D., "Operational Profiles in Software-Reliability Engineering", IEEE Software, 10(2), 1993, pp. 14-32

Dalal, S. R., Mallows, C. L., "When should one stop testing software?", Journal of the American Statistical Association, 83(403), 1988, pp. 872-879.

Freimut, B., "Developing and Using Defect Classification Schemes", Technical Report, IESE Report No. 072.01/E, 2001.

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).

Yang, O., Jenny, Li J., Weiss, D., "A Survey of Coverage Based Testing Tools", International workshop on Automation of Software Test, AST '06, Shanghai, China, 2006, pp. 99-103.

Offutt, J., Xiong, X., Liu, S., "Criterion for generating specification-based tests". Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems - ICECCS 99, Las Vegas, 1999, pp. 199-131.

Chavez, T., "A decision-analytic stopping rule for validation of commercial software systems". IEEE Transactions on Software Engineering, 26(9), 2000, pp.907-918.

Roy Awedikian
PhD Candidate
Johnson Controls Automotive Experience
18, chaussée Jules César
BP 70340-Osny
F-95526 Cergy-Pontoise Cedex
France
Phone: +33 1 3017-5099
Fax: +33 1 3017-6445
Email: roy.awedikian@jci.com