Design of a validation test process of an automotive software

Roy Awédikian, Bernard Yannou

Design of a validation test process of an automotive software

Roy Awedikian, Bernard Yannou

Ecole Centrale Paris
Laboratoire Genie Industriel
Grande Voie des Vignes
92290, Chatenay-Malabry, France
roy.awedikian@graduates.centraliens.net, bernard.yannou@ecp.fr

Abstract: Testing software for defects exhaustively remains a computationally intensive problem. Therefore, deciding when to stop the test of a software product is one of the main issues in software engineering. Introducing fewer defects, detecting defects earlier in the development process and reducing the time to delivery are the primary objectives of software organizations. In particular, we have been working to address this problem within the realm of the automotive suppliers of car-embedded electronic modules and have developed case research with Johnson Controls. In this paper, we describe our approach to automatically generate test cases for a software product and develop the details of our objective and constraint functions for optimizing the test generation. It is based on a compromise between the structural and functional formal coverage and the cost of the generated tests. Finally, we propose a plan to validate and monitor the new software testing process.

Keywords: test design space, exploration, decision making, software testing, software quality.

1- Introduction

Since the 1970s, software engineering methods have been focused on overcoming the quality problems in software systems. In software development [B1], once the design of a software module is completed, the module is sent to a testing group which must check that the module meets the client's requirements. For that reason, testers design a set of test cases to be simulated on the software module. Presently, at Johnson Controls (and in many automotive companies), they still manually design the test cases whose size is often much larger than that of the module code itself. The stopping criteria when generating the series of tests are most often based primarily on covering the software program and on the budget and time constraints. Since there is no unified model to represent the software functional requirements in the automotive industry, assessing a functional coverage of these requirements strongly depends on the organization's internal definition of the requirements.

In [AY1], we (the authors) have recently developed and experimented a new approach to automatically generate test cases for a software module. The test generation is monitored by a set of formal quality indicators such as the structural and functional coverage but also the tests cost that we thoroughly present in this paper. In the next (second) section, we characterize the conventional software test design process presently used at Johnson Controls and we identify the issues stemming from the existing lack of a stopping criteria model. In the third section, we discuss a measurement to stop software testing. In the fourth section, we briefly present our new test design approach. In the fifth section, we focus on the objective and constraint functions used for optimizing the test generation and more precisely on the functional coverage of our simulated model to represent the functional requirements. We also describe the optimization process of designing a test case (selection of test steps). In the sixth section, we outline our plan to validate our new approach; this results in a set of quality attributes related to the testing process that we intend to use for driving a continuous improvement on this software testing process. In the final section, we make some concluding remarks.

2- The software test design process at Johnson Controls

At Johnson Controls, the lifecycle of a software product is divided into 5 global stages: *Request For Quotation, Design, Design Validation, Production Validation and Production*.
Within each stage, the engineering activities are performed according to the standard V-cycle of the software industry 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*. We notice that each of the development, integration and validation processes perform a software testing activity in order to verify and validate the correctness of the software delivered at the end of the process. Software defects are detected in each of these processes, analyzed, corrected and capitalized in the defects database of the company.
The software testing activity consists in:

1) *Analyzing the carmaker software requirements*: testers who need to design tests must first read, analyze and

understand the carmaker software requirements.

2) *Designing the test cases*: presently, testers proceed to a manual design of the test cases. The performance of this activity is mainly based on the experience of the testers.
3) *Simulating the test cases on the software product and detecting the defects*: once the test cases are developed, they are simulated on the software product in order to check that they are "ok" and as free of defects 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 a service to the clients (carmakers and/or drivers). For example, the door lock management functionality.

What is a "test case"?

Let us 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 the output signals after each operation.
An excerpt from a test case designed is given in figure 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.
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	Output_1 = 0 Output_2 = 0 Output_3 = 0
Test Step →	97	Test # 97 Input_1 = 1	Output_1 = 7 Output_2 = 3 Output_3 = 0
	…	…	…

Figure 1. Excerpt from a test case (two operations) as designed by Johnson Controls testers

3- A semi-formal measurement to stop software testing

At the beginning of a project, automotive suppliers officially receive the carmaker requirements. The software department of Johnson Controls like many software organizations adopts the SRS (Software Requirement Specification) model to express the various expectations of a software product. In [I1], we can find the IEEE (Institute of Electrical and Electronics Engineers, Inc.) recommended practices for the SRS. Once the SRS document is ready, it is the unique source of inspiration for testers to design their test cases for the software product. Presently, the major number of automotive suppliers has a manual test design process which is highly dependent to the experience of testers. Indeed, for each client functionality, we can associate a potential test space. But, as the software product becomes more and more complex, it is illusory to be able to check that the software product responds correctly to all possible operations. Therefore, each tester has a different perception of the possible test space and designs the test cases according to this perception. In addition, the criteria to stop designing test cases are based, on the one hand, on a non expressed combination of code and/or requirements coverage (semi-formal measurements) and, on the other hand, on time and money remaining for the project. In software engineering, the term "coverage" means the extent or degree to which something is observed, analyzed, and reported.

3.1- Structural coverage

Currently, during the *structural testing* of a software component at Johnson Controls, practitioners mainly use the code coverage as a criterion to stop testing (A survey on code coverage based testing tools is done in [YJ1]). Code coverage is a way to measure how thoroughly a set of test cases cover a program (see figure 2): the coverage rate of *statements* (lines of code), *procedures*, *conditions* (control flow) and *decisions* in the software product under test. There is a large variety of code coverage measurement criteria but the four listed before are the most used in automotive industry.

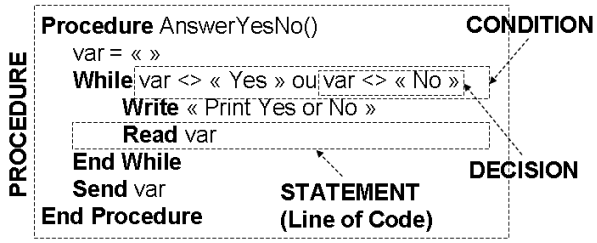

Figure 2. Structural (code) coverage indicators

These criteria are apparently relevant since the goal of the testing activity is to check if all the pieces of the software module have been tested. But it is not that simple!

3.2- Functional coverage

The criterion of structural coverage does not directly assess the compliance of the software module to the carmaker requirements; this is a biased indicator. In fact, the functional coverage is related to the coverage of the functional requirements of the software under test. Through a literature review (see [OX1]), several stop testing criteria based on covering software specification have been proposed. They primarily deal with the transitions coverage of a graph-based specification. At Johnson Controls, the carmaker requirements are referenced and managed using professional tools (Doors) and therefore the coverage rate of these requirements is used as the primary criterion to stop software functional testing. Paradoxically and even a 100% coverage of the functional requirements has been reached during the Johnson Controls testing of a software module, the carmaker is able to detect a nonconformity between the code and their specification. This could lead to the conclusion that the present Johnson Controls definition of a requirement is not

enough refined. In fact, presently, one requirement can hide two or more other requirements. Let us consider an excerpt of software functional requirements as they were defined by a Johnson Controls requirement engineer (see figure 3). These requirements have two inputs and one output: *I1(*with domain *D(I1)={0,1}), I2 (D(I2)={0,1}), O1 (D(O1{0,1}).*

Requirement 1:
In case of input *I1* is equal to 1 and input *I2* is equal to 0, therefore the output *O2* must be set to 0
Requirement 2:
In other cases, Output *O2* is always set to 1

Figure 3. An excerpt of software functional requirements as defined by a Johnson Controls requirement engineer

While testing the conformity of the developed code regarding these two carmaker functional requirements, testers designed two tests in order to cover the previous requirements:
Test 1: set *I1* to 1, *I2* to 0 and check if *O1* is equal to 0
Test 2: set *I1* and *I2* to 1 and check if *O1* is equal to 1
Therefore, he decides to stop testing these requirements and to set them as covered. In fact, through the test #1, the tester covers at 100% the first requirement but the test #2 does not cover at 100% the second requirement. Indeed, the second requirement can be splitted into three requirements to be tested:

- In case of input *I1* is equal to 1 and input *I2* is equal to 1, therefore the output *O2* must be set to 1 – *covered by test #2*
- In case of input *I1* is equal to 0 and input *I2* is equal to 1, therefore the output *O2* must be set to 1 – *not covered by the tests*
- In case of input *I1* is equal to 0 and input *I2* is equal to 0, therefore the output *O2* must be set to 1 – *not covered by the tests*

Consequently and in order to automate the design of test cases and to monitor this automation by quality indicators such as a formal functional coverage, we develop a new approach based on representing the software functional requirements in a simulated model.

4- Our new approach for automating the software test generation

Our new approach of automated test generation presents a much different workflow for generating test series than the present one. The new workflow is based on seven 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 a simulated model that we developed keeping in mind the automotive context and its constraints. In fact, each client functionality has a set of input, output and intermediate signals. These signals are interconnected through elements. An element is a set of functional requirements of the same type. We propose at a first level two types of functional requirements :

Combinatorial if the outputs values at instant *t* depend on the sole inputs values at instant *t*.
Sequential if the outputs values at instant *t* not only depend on inputs values at instant *t* but also on the outputs values at instant *t-1*.
Therefore, we propose to model these two types of functional requirements thanks to two types of modelling elements, namely *Decision Table* (DT) [C2] and *Finite State Machine* (FSM) [G1]. We provide, in figure 4, a graphical illustration of our functional requirements model. This example has 4 input signals, 4 output signals, 5 intermediate signals and 4 elements. A *"clock"* signal is required since the behaviour of a software product is ruled by synchronism. In fact, a clock is just a signal that alternates between zero and one, back and forth, at a specific pace (cycle time). It sets the "pace" for the functional simulation of the model.

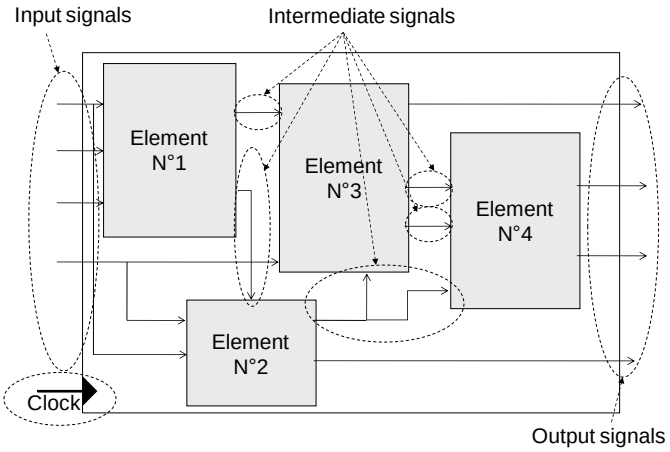

Figure 4. Graphical illustration of our simulated model to represent functional requirements

We can find a detailed description of our model for modelling and simulating the software functional requirements in [AY3].

2. Verify and validate the consistency and compliance of the software requirement model [S1]. Verification is done to ensure that the consistency and completeness of the developed model. To do so, we defined a set of integrity rules to be checked automatically once a model is developed. We can find these rules in [AY3]. Verification ensures that mistakes have not been made in implementing the model but does not ensure the compliance of the model to the carmaker software requirements which is the scope of the model validation. In fact, we proposed three main scenarios to validate a software requirement model in our context. We can find more details on these scenarios in [AY3]
3. Define some behavioural characteristics of a car driver when using the client functionality under test. To do so, we develop in [AY2] three types of constraints that engineers can affect to an input signal of the requirements model in order to eliminate or favour specific "successive" operations. These constraints aim to reduce the number of possible combinations on the input signals and to more thoroughly pinpoint which ones have a high potential to detect defects.

4. Reuse the defects and the test cases respectively detected and developed in the past on the same client functionality [F1]. In fact, using capitalized defects and test cases seems to be beneficial in automotive context since more than 50% of the functionalities performed by a software product are common to any series of cars. To do this, we developed a framework which takes the capitalized defects (under a specific format: operations that lead to the defect and the time intervals between these operations) and existing test cases and generates automatically a set of successive operations with time intervals to be primarily simulated on the functionality under test. Therefore, we can better address the problem of detecting defects by generating tests to check the non-existence of recurrent defects and reducing the test design space by focusing on the test scenarios based on our returns of experience.
5. Enrich the requirement model with knowledge on the driver recurrent operations and the testers' experience [M1]. To do so, we propose to set probabilities between all possible successive operations of a client functionality. Therefore, 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:
   - The *probability* that two operations are in sequence.
   - The *time between these 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 fig. 5.

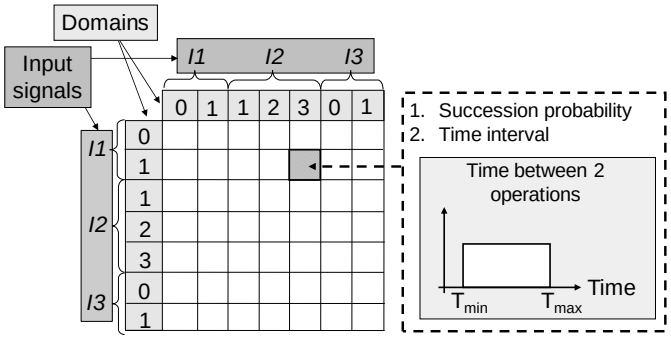

Figure 5. Operation matrix

We can find more details on activities 3, 4 and 5 in [AY2].

6. Manage the test generation with the cost, delay and quality indicators.
7. Automate the generation of test cases from the enriched model (by activities 3 to 5) of functional requirements.

In the following, we further detail our objective and constraint functions for optimizing the test generation and our optimization algorithm to design efficient test cases fulfilling one or more predefined targets on the quality indicators.

5- A compromise to stop designing test cases when fulfilling the test objectives and constraints

Testing software exhaustively remains a very hard problem to solve. Therefore, software testing must often be based on specific assumptions and objectives which help practitioners and managers to decide when to stop the testing protocol [DM1]. Several stopping criteria have been proposed in the software testing literature. In [AY4], we (the authors) made a detailed survey on these criteria. In this paper and in order to monitor the automatic generation of test cases, we develop:

- an objective function based on formal structural (code) and functional (specification) coverage,
- a constraint function based on test execution time and cost,

and an optimization algorithm which aims to fulfill the test objectives while respecting the time and cost constraints. Indeed, we develop a panel interface to allow the test engineers to set the test generation objectives and constraints (see figure 11). We also define a set of weights (*wi*) that test engineers can associate for each defined objective or constraint: 0 (to be ignored), 1 (not very important), 5 (important), 10 (very important). The panel helps test engineers to express their objectives and constraints in terms of the required software quality and tests cost and therefore generate test cases fulfilling their expectations.

5.1- Objective function

The *objective function*, $F_{Objective}$, is defined as in formula (1).

$$F_{Objective} = \underbrace{\sum | StrucCov_{Target} - StrucCov_{Current} | \times w_i}_{\text{Structural coverage}} + \underbrace{\sum | FuncCov_{Target} - FuncCov_{Current} | \times w_i}_{\text{Functional coverage}} \quad (1)$$

where *StrucCovTarget* and *FuncCovTarget* are the coverage goals as defined by the test engineers, *StrucCovCurrent* and *FuncCovCurrent* are the coverage reached by the test case under design and *wis* are weights. The structural and functional coverage are expressed in terms of ratios of coverage and, then, are normalized which aim to reach a value of 100%.

5.1.1- Structural coverage

While generating a test case, and for each test step generation, we simulate the test step on the software product under test and we evaluate the code coverage in terms of statements, procedures, conditions and decisions coverage (see figure 2). To do so, we use *C-Cover* from *Bullseye* as a code coverage measurement tool.

5.1.2- Functional coverage

Once we define a model to formally represent the software functional requirements, we consider a formal coverage rate of the requirements model (see figure 6):

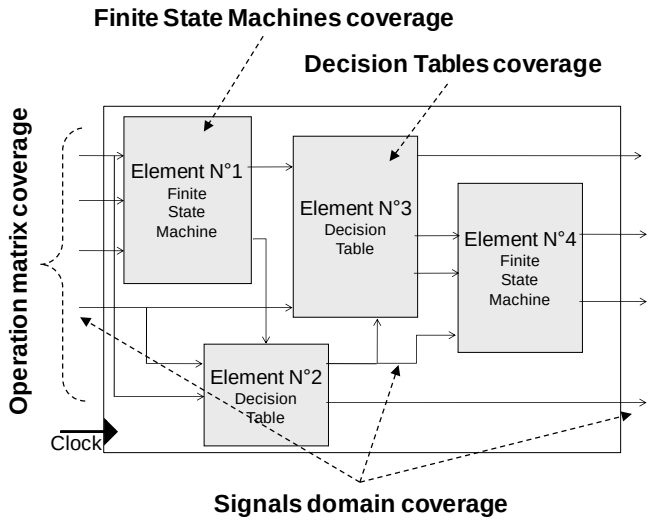

Figure 6. Functional coverage indicators

The coverage rate of an element

There are two types of elements, namely:
Decision table (see figure 7): We use a DT to characterize a set of combinatorial software functional requirements. A DT is a tabular form that presents a set of exclusive conditions on the inputs *(Ci)* and their corresponding set of actions on the outputs *(Ai)*. A condition (for example, *C1*) must require that at least one input is set to a specific value (*i3=1*), the other inputs may be indifferent (∀).

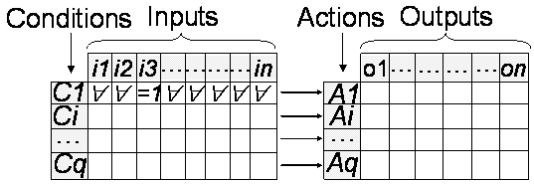

Figure 7. A decision table (DT) element

Finite state machine (see figure 8): We use a FSM to characterize a set of sequential software functional requirements. A FSM is a model of behaviour composed of an initial state, a finite number of states with a set of actions on the outputs *(A)*, a set of transitions between these states and, for each transition, a set of exclusive conditions on the inputs *(C)* that allows the transition to be passed.

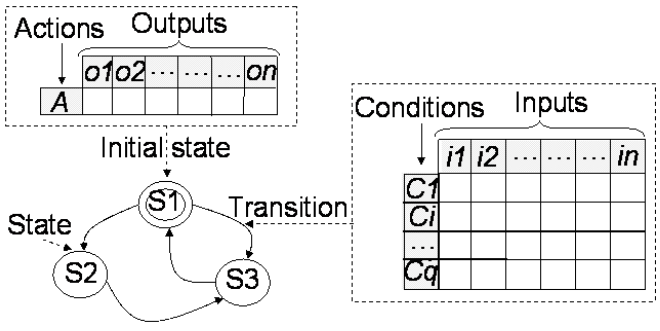

Figure 8. A finite state machine (FSM) element

The *element* coverage of a requirements model consists of the coverage rate of the conditions of each Decision Table and the coverage rate of the states, transitions and conditions of each Finite State Machine. Moreover, when designing the functional requirements model, practitioners can affect to conditions, states and transitions a normalized criticity level between 0 and 1. Consequently, we define a second set of coverage rates for expressing the degrees of coverage of the most critical elements (in fact, this is a weighted coverage of an element).

The coverage rate of a signal domain

In fact, each signal (input, output or intermediate) has a discrete domain. The *signal domain* coverage of a requirements model consists of the coverage rate of the domains of the inputs, outputs and intermediates signals. In addition, since testing the boundary values of a signal often reveals many defects, we also assess the coverage rate of the minimum and maximum values of each signal. In figure 9, we illustrate the coverage of a signal by a practical example. In fact, the signal *"Signal_3"* is covered at 100% while the two values of this signal were visited at least once during the generated tests. The signal *"Signal_1"* has a coverage rate of 33,33% (1 value visited over 3 values in total).

Signal name	Signal domain
Signal_1	12 6 18
Signal_2	1 0
Signal_3	1 0
Signal_4	7 6 5 4 3 2 1 0
Signal_5	1 0
Signal_6	1 0
Signal_7	1 0

Figure 9. Signals domain coverage

The coverage rate of an operation matrix

The *operation matrix* coverage of a requirements model consists of the coverage rate of all successions between pairs of operations visited. Once a succession probability is set between each two operations, we define a coverage rate of the critical successions where the succession probability is above a certain level defined by the practitioner. Let us consider the example of figure 10. After generating a test case, some cases of the matrix have been highlighted. In fact, in the generated test case, the operation #4 has followed the operation #1, the operation #2 has followed the operation #2, the operation #2 has followed the operation #3 and so on. This way, we compute the coverage rate of successions between pairs of operations (around 38% : 5 successions of operations were covered over 13 possible successions)

Operations	Op1	Op2	Op3	Op4
Op1	0,2 - [100;200]	0,1 - [200;200]	0,4 - [100;300]	0,3 - [100;200]
Op2	0	0,5 - [100;200]	0	0,5 - [100;100]
Op3	0,2 - [200;200]	0,4 - [100;200]	0,2 - [100;300]	0,2 - [100;300]
Op4	0,2 - [100;400]	0,2 - [100;200]	0	0,6 - [100;200]

Figure 10. Operation matrix coverage

Finally, while generating the tests, practitioners can visualize in real time the covered zones of the functional requirements

model. An estimate of the coverage rate is computed after each test step generation.

5.2- Constraint function

The *constraint function*, $F_{Constraint}$, is defined as:

$$F_{Constraint} = \sum | Cons_{Target} - Cons_{Current} | \times w_i \quad (2)$$

where $Cons_{Target}$ are the constraints' values as defined by the test engineers, $Cons_{Current}$ are the constraints' values reached by the test case under design and $w_i s$ are weights.

5.2.1- Test execution time and cost constraints

Unfortunately, in the automotive industry, the time and money spent to test a software product is the major criterion to stop testing. When generating test cases automatically, one can have a tendency to generate too many tests. In fact, executing test cases on the software under test and analyzing the results can cost too much time and money and more especially when the execution is performed manually by a test engineer. In the proposed approach and when generating a test case, test engineers can set a group of time and cost constraints that should be respected:

- Constraint 1: *Execution time*. The time that a test engineer will spend in executing manually the generated test case on the software product.
- Constraint 2: *Number of test steps* in the generated test case.
- Constraint 3: *Number of "distinct" test steps* in the generated test case. Two test steps are distinct if they have different input data.

In order to have a consistent aggregated constraint function ($F_{Constraint}$), we normalize to 100% the time and cost constraints. These constraints are expressed in *millisecond* (*ms*) and in number of generated test steps. We illustrate the normalization process of these constraints through an example. At each time, test engineers decide to set a constraint C_i, the normalized target of this constraint $Cons_{Target}(C_i)$ is immediately set to 100%. For instance, once a test engineer decide to generate a test case that the total execution time do not exceed 108000 *ms*, the normalized target of the test execution time constraint is set to 100% ($Cons_{Target}$*(test execution time)*= 100%). After generating a set of test steps, the normalized current value of this constraint ($Cons_{Current}(C_i)$) is assessed by calculating the ratio (*current_constraint_value*100/target_constraint_value*).
When generating a set of test steps with a total execution time of 21600 *ms*, $Cons_{Current}$*(test execution time)* is assessed to (21600*100)/108000 ($Cons_{Current}$*(test execution time)*= 20%).

Constraints on time and cost are helpful in case of tight planning and budget on the project. It can also be useful on projects where the test execution is performed manually. In that case, the execution time and number of test steps must be reduced and the repetitive test steps or succession of test steps must be avoided. Typically, when testing a *Graphical User Interface* (*GUI*), test engineers have to check visually the expected results. Nevertheless, new testing platforms allow even to automate the testing of GUI using a camera system.

5.3- How to design optimal test cases?

Once the functional requirement model is ready and at least one "operation matrix" is established, automatically generating a test case requires performing a set of test steps until the stopping criteria are reached. Two automated activities are necessary to generate one test step:

Activity 1: Perform a Monte Carlo simulation on an "operation matrix"
Two sub-steps are required:
Step 1: an operation is chosen according to the probabilities between each two successive operations [MT1].
Step 2: the inter-operation time is randomly chosen within the time interval.

Activity 2: Simulate the functional requirements model and calculate the expected values of the output signals
A synchronized functional simulation is performed on the model of software functional requirements. The simulation is done with an oriented acyclic logic going from the input to the output signals of the client functionality. The simulation order of the elements has to be defined when designing the requirements model (element 1 then element 2 then element 3 …). A "clock" input synchronizes the behaviour of the functional model. At each cycle time of the clock, all elements are simulated following the predefined order. Simulating an element consists of calculating its output signals values according to its input signals values. In [AY3], we can find a detailed presentation of the simulation of our software requirements model.

Through the proposed approach, the process of designing a test case is monitored by an optimization algorithm based on a mix of *simulated annealing* and *look-ahead* strategies. The aim of this algorithm is to reach the coverage objectives while respecting at most the time and cost constraints. During a test case design session and after each test step design (after activity 1 / *step 1*), the contribution of the designed test step to the objectives fulfillment is assessed. In case of no contribution, the designed test step is rejected and a new test step is designed. In the other case, the coverage and constraint indicators are updated and the objective and constraint functions are assessed. The quality objectives (structural and functional coverage) may be fulfilled following different orders and the first objective fulfilled does not immediately stop the process. We stop the process:

1) When the objective function ($F_{Objective}$) attains zero.
2) When the constraint function ($F_{Constraint}$) increases for a certain number of successive generated test steps without any improvement in the objective function ($F_{Objective}$).

In fact, for a set of targets and weights on the quality indicators, the practitioner can generate more than one test case fulfilling these predefined objectives. Afterwards, the optimal test case is automatically selected. To do so, we compare the generated test cases in pairs and we select the one which has the lowest value of the aggregated preference F of the quality indicators. If the two test cases have the same value of F, we select the utmost one that meets each individual targets going from the higher to the lower weights.

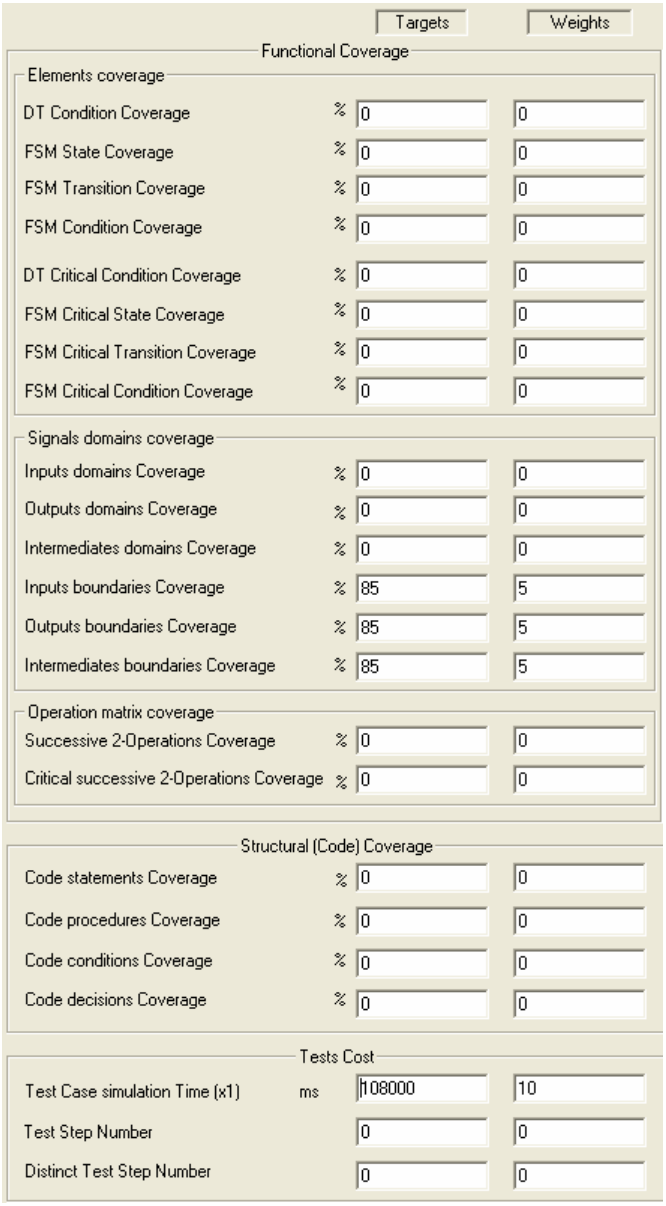

Figure 11. Panel of the quality indicators for monitoring the automatic test case generation

Let us consider a practical software testing problem in order to illustrate the purpose of our objective and constraint functions. Through the experience feedback of the software testing experts, some software defects often occur when a signal is set to its boundaries values. Consequently, test practitioners could always decide to generate a test case (a set of test steps) which aims to detect potential defects related to the boundaries values. Hereafter, we consider the client functionality which consists in managing the front wiper in a vehicle. The corresponding software component is made of 1229 Lines of code (blank and comment lines excluded), 18 input signals and 8 output signals. We decide to generate a test case fulfilling the following objectives, constraints and weights (see figure 11):

- Cover the boundaries input signals at 85% with a weight of 5
- Cover the boundaries output signals at 85% with a weight of 5
- Cover the boundaries intermediate signals at 85% with a weight of 5
- Do not exceed 30 minutes (108000 ms) of tests simulation with a weight of 10

Please note that, in this example, we do not deal with criticity, structural coverage and non-repetitive test step number. In figure 12, and after generating one test case with objectives and constraints defined below, we can see the current (reached) target and weight of each the quality indicators. In fact, even if the *inputs and outputs boundaries coverage* have respectively reached and exceeded their targets (respectively 85% and 94% of coverage), our optimization algorithm did not stop the tests generation expecting that the *intermediate boundaries coverage* reaches its target. But once the maximum *test simulation time* which has a weight of 10 (very important) has been exceeded (110255 ms instead of 108000 ms), the optimization algorithm decides to stop generating test steps even if the *intermediates boundaries coverage* is not already reached.

TC Quality Indicators	Progress	Current	Target	Weight
****Functional Coverage****				
DT Condition		68 %	0 %	0
FSM State		63 %	0 %	0
FSM Transition		27 %	0 %	0
FSM Condition		30 %	0 %	0
Inputs domains		83 %	0 %	0
Outputs domains		84 %	0 %	0
Intermediates domains		78 %	0 %	0
Inputs boundaries		85 %	85 %	5
Outputs boundaries		94 %	85 %	5
Intermediates boundaries		77 %	85 %	5
Successive 2-Operations		5 %	0 %	0
****Test Case Size****				
TC simulation Time (ms)		110255	108000	10
Test Step Number		445	0	0

Figure 12. A result of a test case generation

6- Validating and monitoring the new software testing process

The validation of our new approach consists of two initiatives.

***In the first stage*,** we verified the results on two case studies (the "front wiper" and "fuel gauge" functionalities) with historical data. The second case study differs from the first one in that it contains different formats of the carmaker's software requirements and it is considered to be a more complex software product. The aim of the experiment was to test our new approach with a software product that was already tested through Johnson Control's conventional approach. The results of these case studies highlight the benefits of our new approach:

- We *increase by at least 100%* the number of defects detected since the first testing phase (*front wiper:* from 12 to 24 defects, *fuel gauge:* from 2 to 18 defects)

- We *decrease by at least 60%* the number of defects detected by the carmaker (*front wiper:* from 5 to 2 defects, *fuel gauge:* from 5 to 1 defects)
- We *increase by at least 25%* the number of defects detected by Johnson Controls and not by the carmaker (*front wiper:* from 17 to 24 defects, *fuel gauge:* from 18 to 22 defects)
- On the first case study, we *detect 5 new "minor" defects.* These defects were not detected by Johnson Controls nor by the carmaker
- We *lower by at least 20%* the time spent in testing the software (*front wiper:* 39 instead of 53,75 eight-hour days, *fuel gauge:* 41,5 instead of 50 eight-hour days)

In addition to estimating the benefits in terms of defects detection and time spent in testing the software, we plan to verify the three properties of a reliable measurement system [B2]: reproducibility, repeatability and accuracy.

Reproducibility: In our new approach of software testing, two main activities depend on the operator (i.e., human intervention). The first one is the design of the requirements model and the second one is the definition of a set of targets and weights. Therefore, two operators must independently model the same software requirements and define a set of targets from the same global test objectives. Finally, each operator has to generate automatically a set of test cases fulfilling the predefined targets.

Repeatability: Since our test optimization algorithm is partly based on a stochastic process, the repeatability must be verified. Consequently, for each operator, we perform two automatic test generations from the same model and targets. Once simulating, on the software module under test, the test cases generated by the two operators, we can measure the number of detected defects by piece. In our case, a piece can be associated with a *C* (programming language) procedure of the software module. Consequently, we can assess the *R&R* (Reproducibility & Repeatability) degree for which the automotive industry standard is 90% for 30 pieces.

Accuracy: It is important to assess the correctness of the results delivered by our measurement system. Therefore, we plan to measure:
- The ratio between the number of new defects detected by our approach and the total number of defects already detected through the conventional testing approach.
- The ratio between the number of *"true"* defects detected by our approach and the total number of defects already detected through the conventional testing approach.
- The ratio between the number of *"false"* defects detected by our approach and the total number of defects already detected through the conventional testing approach.

***In the second stage*,** we plan to monitor the quality of our new testing process. To do so, it seems that within the *Design for Six Sigma (DFSS)* framework, the *Define, Measure, Analyze, Design, Optimize, and Verify (DMADOV)* methodology is the appropriate approach. This will allow us to put the proper focus on the up front design of the testing process. Therefore, we need to establish the set of measurable, customer-oriented attributes, which can be defined, measured, analyzed, optimized and verified (DMADOV) in the software testing process. These attributes need to be directly built into the testing process so that it is specifically geared to producing pre-defined quality limits. This means embedding specific design intent within the software testing algorithm to meet specific and understood, customer-facing performance metrics. Below, we identify two types of *critical-to-customer* metrics concerning the software testing process at Johnson Controls. We plan to assess the following metrics on each software module that undergoes testing:

Critical-to-Quality (CTQ) metrics:

Y1. The capacity to reach 100% of the functional coverage: the reached functional coverage
Y2. The capacity to reach 100% of the structural coverage: the reached structural coverage
Y3. The accuracy of the testing process: the ratio between the number of "false" defects detected and the total number of defects
Y4. The capacity to reduce the number of defects detected by the carmaker: the ratio between the number of defects detected by carmakers and the total number of defects
Y5. The capacity to reduce the number of defects detected by the end user (driver): the ratio between the number of defects detected by the end users and the total number of defects

Critical-to-delivery (CTD) metrics:

Y6. The time spent to test the software module
Y7. The capacity to detect the defects earlier in the software development cycle: the ratios between the number of defects detected by Johnson Controls during the different testing phases and the total number of defects
Y8. The number of versions of the software module
Y9. The capacity to deliver software module free of defects since the first delivery: the ratio between the number of defects detected by Johnson Controls in the first testing phase and the total number of defects

Since we place a high premium on reducing the number of defects detected by carmakers and end users (Y4 and Y5), one solution is to increase the functional and structural coverage. But, experimentations reveal that some defects cannot be detected even if our functional requirements model and code are covered at 100%. This leads to the realization that we need to refine our functional coverage model. Typically, we can consider the coverage rate of the succession of two transitions in a FSM element.

7- Conclusion

In this paper, we focus on the objective function that we developed and implemented in our global automatic test approach in order to optimize the test generation. The basic aim of this model is to make a compromise between the software quality and the cost of testing. We assess the quality through two indicators: the structural coverage of the program under test and the formal functional coverage of the requirement model. We also initiate the plan to validate our new testing approach through the use of a case study with historical data. The results of this first investigation were promising. Properties such as reproducibility, repeatability

and accuracy will be verified on the developed case studies. Finally, we describe a set of quality attributes to monitor the quality of the software testing process and therefore identify improvement actions to be performed on the process. As a perspective, we have to manage the change of the practices and activities of hundreds of testers at Johnson Controls. Indeed, testers' technical skills will have to switch from a manual design to a high level modelling of the test scenarios and objectives in using in a flexible manner our design approach.

8- References

[AY1] Awedikian R., Yannou B. Automatic test case prioritization through a model-based statistical testing approach: application in the automotive industry. Software Testing, Verification & Reliability. Submitted on May 2010.

[AY2] Awedikian R., Yannou B., Mekhilef M., Bouclier L. and Lebreton P., "A Radical improvement of software defects detection when automating the test generation process", Proceedings of the 10th International Design Conference - DESIGN 2008, Dubrovnik, 2008.

[AY3] Awedikian R., Yannou B., Mekhilef M., Bouclier L. and Lebreton P., "A simulated model of software specifications for automating functional tests design", Proceedings of the 10th International Design Conference - DESIGN 2008, Dubrovnik, 2008.

[AY4] Awedikian R., "Quality of the design of test cases for automotive software: design platform and testing process." Doctoral disseration, Extended Abstract, Ecole Centrale Paris, Châtenay Malabry, France, 2009. Available at http://www.lgi.ecp.fr/uploads/PagesPerso/PhD_Extended_Abstract.pdf [26 September 2010].

[B1] Beizer B., "Software Testing Techniques", second edition, 1990 (Van Nostrand Reinhold).

[B2] Breyfogle III Forrest W., "Implementing Six Sigma Smarter Solutions Using Statistical Methods", 1999. (John Wiley & Sons).

[C1] Chávez T., "A decision-analytic stopping rule for validation of commercial software systems". IEEE Transactions on Software Engineering, 2000, 26(9):907-918.

[C2] Chvalovsky V., "Decision tables", Software: Practice and Experience, Vol. 13, No.5, 1983, pp. 423-429.

[DM1] Dalal, S. R., Mallows, C. L., "When should one stop testing software?", Journal of the American Statistical Association, 83(403), 1988, pp. 872-879.

[F1] Freimut, B., "Developing and Using Defect Classification Schemes", Technical Report, IESE Report No. 072.01/E, 2001.

[G1] Gill A, "Introduction to the theory of finite-state machines", McGraw Hill, NJ, 1962.

[I1] Institute of Electrical and Electronics Engineers (IEEE), "IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications" (SRS), 1998.

[M1] Musa, J. D., "Operational Profiles in Software-Reliability Engineering", IEEE Software, 10(2), 1993, pp. 14-32.

[MT1] 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).

[OX1] 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).

[S1] Sargent, R. G., "Verification and Validation of Simualtion Models", Proceedings of the 37th Winter Simulation Conference - WSC 2005, Orlando, FL, USA, 2005, pp. 37-48.

[YJ1] 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.