A practical model-based statistical approach for generating functional test cases: application in the automotive industry

Roy Awédikian, Bernard Yannou

Paper title: A practical model-based statistical approach for generating functional test cases: application in the automotive industry

Authors:

Roy AWEDIKIAN (Corresponding Author)
Affiliation 1 : (Permanent address)
Ecole Centrale Paris
Laboratoire Genie Industriel
F-92295 Châtenay Malabry Cedex
France
Affiliation 2 :
Johnson Controls Automotive Electronics
Electronics Division Europe
Parc Saint Christophe
95892 Cergy Pontoise Cedex
France
Mobile : +33 6 80 02 75 92
Fax : +33 1 41 13 12 72
Email : roy.awedikian@graduates.centraliens.net

Bernard YANNOU
Affiliation :
Ecole Centrale Paris
Laboratoire Genie Industriel
F-92295 Châtenay Malabry Cedex
France
Tel : +33 1 41 13 15 21
Fax : +33 1 41 13 12 72
Mobile : +33 6 64 25 96 59
Email : bernard.yannou@ecp.fr

1 Abstract

With the growing complexity of industrial software applications, industrials are looking for efficient and practical methods to validate the software. This paper develops a Model-Based Statistical Testing (MBST) approach that automatically generates online and offline test cases for embedded software. It discusses an integrated framework that combines solutions for three major software testing research questions: 1) how to select test inputs; 2) how to predict the expected results of a test; and 3) when to stop testing software. The automatic selection of test inputs is based on a stochastic test model that accounts for the main particularity of embedded software: time sensitivity. Software test practitioners may design one or more test models when they generate random, user-oriented, or fault-oriented test inputs. A formal framework integrating existing and appropriate specification techniques was developed for the design of automated test oracles (executable software specifications) and the formal measurement of functional coverage. The decision to stop testing software is based on both test coverage objectives and cost constraints. This approach was tested on two representative case studies from the automotive industry. The experiment was performed at unit testing level in a simulated environment on a host PC (automatic test execution). The two software functionalities tested had previously been unit tested and validated using the test design approach conventionally used in industry. Applying the proposed MBST approach to these two case studies, significant improvements in performing functional unit testing in a real and complex industrial context were obtained: more bugs were detected earlier and in a shorter time.

Keywords: software testing, model-based, statistical testing, automation, embedded software, automotive.

2 Industrial context and problem

2.1 Growing complexity of automotive software

Nowadays, car electronics represent more than 30% of the total cost of a car [1]. As architectures for car electronics become more and more complex, carmakers outsource the design of some electronic modules to automotive electronics suppliers. The design of a module typically represents 24 months of development and involves around 25 management and technical engineers with a range of hardware, software and mechanical competencies. The software testing activity takes up to 50% of the total time spent in management and technical activities and the software components of such a module accounts for more than 80% of the total number of defects detected on the module.

In the automotive industry, the engineering processes of software development are performed according to the standard V-model of the software industry [1]. However, an iterative and incremental design process is also initiated between the carmakers and their suppliers in order to take the carmaker's constraints and prioritization of requirements into account. The number of increments (deliveries) is defined based on the complexity of the project and adjusted in accordance with the carmaker's inputs and project constraints. In a fairly complex project, ten is the typical number of increments [2]. After each delivery, despite the verification and validation (V&V) activities of the supplier, the carmaker still detects a number of software nonconformities (in this article, the term "bug" is used instead of "software nonconformities"). This number depends on the size (in terms of lines of code), complexity, and maturity of the delivered software. Moreover, once an electronics module is launched on the market (i.e., integrated into a vehicle), an average of one bug per year is detected by end-users [2], which may lead to significant financial consequences for the electronics supplier. Therefore, finding bugs earlier in the product life cycle, specifically in the development phase (thus reducing the number of bugs detected by carmakers and end-users) is a priority for suppliers of automotive embedded software.

2.2 Automotive software V&V techniques

In the automotive industry, both static and dynamic software V&V techniques [3] are practiced in order to ensure that the resulting software product meets the customer's expectations. Testing activities represent up to 90% of the time spent in the V&V of an automotive software product. Unit tests act on a specific component of the system, while validation tests act on the system as a whole. Many automotive industrials have invested in automating test execution; however, test design is still a manual activity, completely based on the practitioner's experience.

The main purpose when unit testing a software component is to cover 100% of the component's source code (100% of the structural flows). This activity, illustrated in Figure 1, is performed by the individuals who develop the component. These developers analyze the structure of the software component being tested (White-Box approach) and select a test input. Afterwards, by analyzing the source code of the component, they predict the expected outputs to be checked against the actual output signals. Developers do not check the behavior of all the output signals that correspond to each test input of the software, but rather only those that correspond to the performed operation. If the designed test step (test inputs and expected outputs) covers all of the source code, developers stop designing test steps. If not, developers thoroughly analyze the non-covered areas of code with the goal of designing one or more test steps that address these areas.

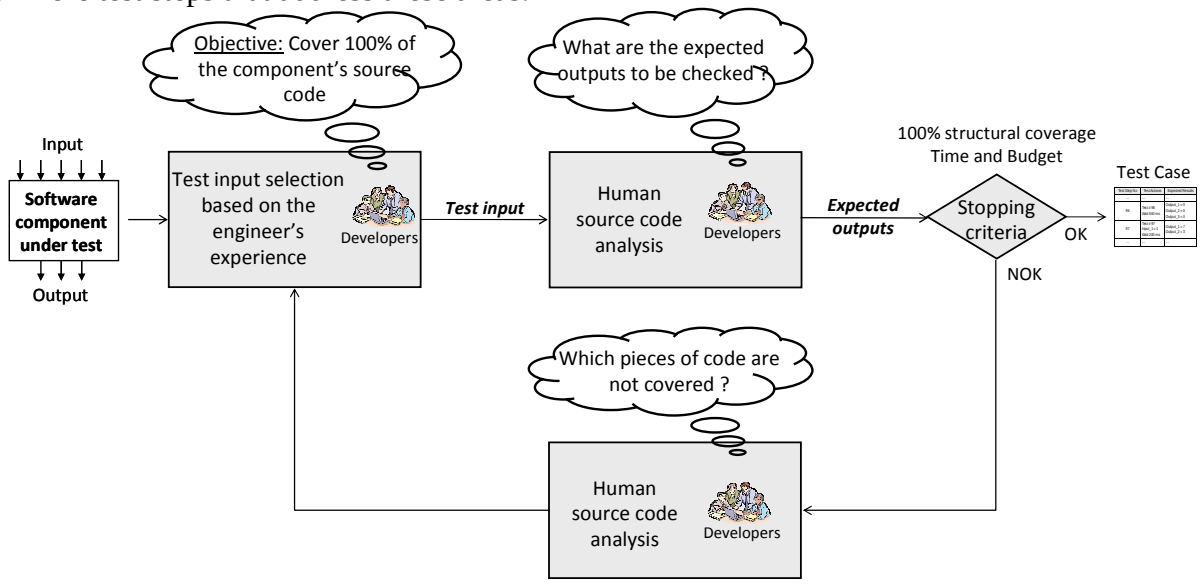


Figure 1 – Conventional unit test design approach in the automotive industry

At present, the unit test is not responsible for ensuring that a software component is compliant with the carmaker requirements. Instead, once a set of unit-tested components are integrated, test practitioners must ensure that the whole software product is compliant with the carmaker requirements. As illustrated in Figure 2, they analyze one or more software requirements (Black-Box testing) and select a test input. Afterwards, by analyzing the carmaker requirements, they predict the expected values to be checked against the actual output signals. As with the design of a test case (set of test steps) for the unit test, test practitioners check only some output signals. They check that the behavior of the output signals matches their understanding of the carmaker requirements. If the designed test steps cover the carmaker requirements concerned, test practitioners stop designing test steps. If not, they thoroughly analyze the requirements under consideration with the goal of designing one or more test steps that completely cover them.

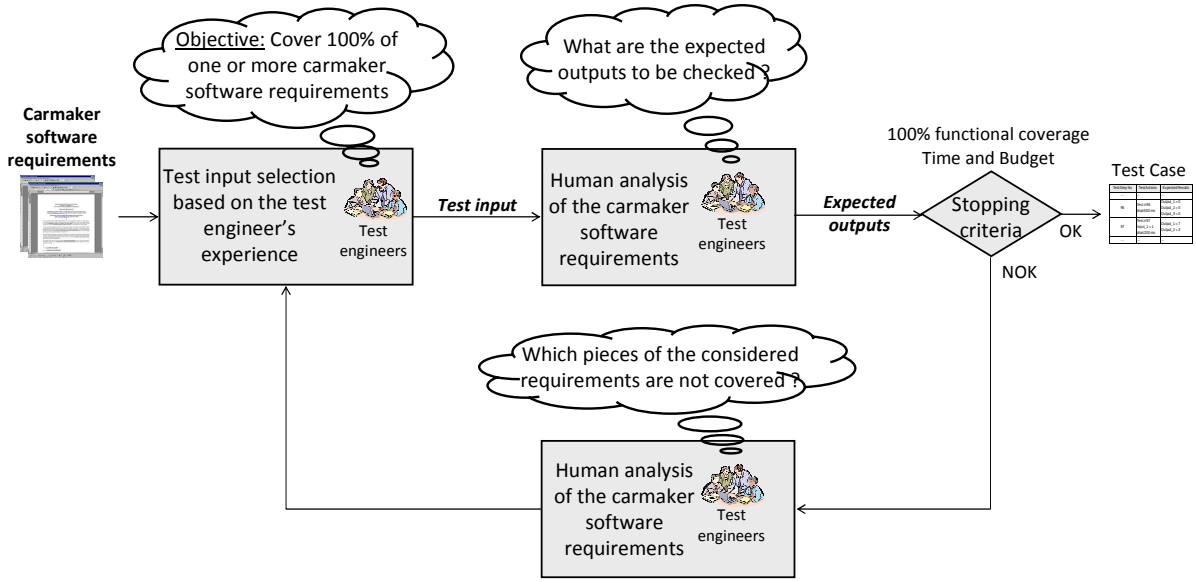


Figure 2 – Conventional validation test design approach in the automotive industry

Sometimes, for time and budget reasons, managers may decide to stop testing software even if 100% structural (code) and/or functional (specification) coverage are not reached. However, the carmaker must be notified of the parts that are not covered.

As software products become more and more complex, it becomes impossible to be able to check that they respond correctly to all possible test inputs. Seroussi and Bshouty [4] show that the design of an optimal exhaustive test case for software is an NP-complete problem. In the automotive industry, a software product is always tested against predefined objectives such as structural (code) and functional (specification) coverage. While structural coverage can be formally measured using computer tools [5], functional coverage is more difficult to measure formally, especially when specifications are expressed in an informal language. From an analysis of 10 software specification documents from different carmakers [2], the authors find that natural languages are still often used when specifying software functionality in the automotive industry.

2.3 Industrial needs and expectations

Facing this growing complexity, carmakers and automotive electronics suppliers are looking for efficient methods to validate software. As the automotive market becomes more and more competitive, decreasing the development time of outsourced parts and decreasing the number of problems detected downstream in the process become of major importance to carmakers and, consequently, become major indicators in the selection of automotive suppliers; the carmakers' process for assigning new projects to suppliers is mainly based on feedback from previous projects. In consequence, suppliers work on reducing the development time of their products and detecting the maximum number of bugs as early as possible in the development process.

A report from the National Institute of Standards and Technology [6] shows that the majority of bugs is introduced during the first part of the software development phase (around 90% in requirements analysis, design, and implementation activities) and detected in the latter part (around 80% during unit testing, validation, and serial production). It also illustrates the growing cost of bug correction once detected downstream in the software life cycle. Two complementary approaches may lead to delivery of bug-free software:

- Lower the number of bugs introduced in the software (prevention approach)
- Detect and handle all the bugs that have been introduced in the software as early as possible (detection and handling approach).

While sophisticated bug-prevention methods and tools are widely used in industry [7], a report from the National Institute of Standards and Technology [6] points out the lack of methodologies, tools, and knowledge in bug-detection techniques, and, more particularly, in testing techniques. In this paper, an integrated Model-Based Statistical Testing (MBST) approach to improve the performance of the test case design process for automotive embedded software is proposed. Test cases can be generated offline and later executed, or they can be generated and executed online. This approach was evaluated using two typical automotive case studies. Each case study consists of automatically generating test cases (offline) for the functional unit test of a software functionality. Each functionality has already been developed and validated (unit and conformance testing) in the past with the V&V techniques currently used in the automotive industry. The generated test cases were executed in a simulated environment (host PC). The performance of the proposed framework regarding the conventional one was quantitatively measured using two metrics: the number of bugs detected earlier in the software development phase and the time spent in testing the software.

After a characterization of the software design environment in the automotive industry, a literature review on the MBT approach is discussed in section 3. An overview of the integrated model-based statistical approach for generating test cases is provided in section 4. The test oracle, test input selection, and stop testing criteria of this approach are developed in sections 5, 6, and 7, respectively. The performance of the proposed MBST approach through two industrial, practical case studies with historical data is assessed in section 8. The validity threats of the experimental results are outlined. Finally, future aims for this research are discussed in section 9.

3 Literature review on model-based testing

Studies show that testing a variety of applications using MBT has been successful. For a sample of such studies, the works of Agrawal and Whittaker [8], Bauer et al. [9], and Bernard et al. [10] on testing embedded controller software were considered; Rosaria and Robinson [11] on testing graphical user interfaces; and Avritzer and Larson [12] and Dalal et al. [13] on testing phone systems. These works indicate that MBT is tailored for small applications, embedded systems, user interfaces, and state-rich systems with fairly complex data. Recently, Siegl et al. [14] present an approach to formalize the requirements specification by test models. These models serve as basis for the testing activities, including the automated derivation of executable test cases from it. Test cases can be derived statistically, randomly on the basis of operational profiles, and deterministically in order to perform different testing strategies. They have applied their approach with a large German OEM in different development stages of active safety and energy management functionalities. A variant of MBT is Model-Based Statistical Testing (MBST), a Black-Box technique that enables the generation of tests that are representative of the perspective of the tester or the user. It has also been used for testing a wide range of applications. These applications vary from sophisticated software engineering environments to databases and large industrial software systems. MBST has also been used in projects involving embedded systems, such as medical devices [15] and automotive modules [16]. Bohr [17] proposes an extension to MBST which deals with the notion of time and concurrency while maintaining all the advantages of MBST. This is done by using an advanced kind of Petri nets as test model. He also shows that it is possible to generate executable test cases (including oracle information) from the Petri nets. Throughout this paper, the utility of a Model-Based testing approach within the embedded software industry is emphasized.

Ozekici [18] discusses some interesting statistical issues that arise in usage testing of software. Wohlin and Runeson [19] also discuss the effect of usage modeling in software certification. A stochastic model of software usage involving Markov chains is employed in Whittaker and Poore [20] and Whittaker and Thomason [21]. In their approach, the sequence

of “inputs” provided by the user is modeled as a Markov chain. This results in a model involving all possible values of input variables. Their model is similar to the proposed one in the sense that they model the sequence of “inputs” by a Markov chain. However, in contrast to the approach proposed in this paper, there is no mention of testing the characteristics that are specific to embedded systems. Embedded systems are very often real-time systems, and an adequate testing approach must consider the properties particular to these systems, especially their time sensitivity. Hessel et al. [22] present principles and techniques for model-based Black-Box conformance testing of real-time systems using the UPPAAL model-checking tool suite. In the proposed approach, real-time constraints are taken into account (specification language and test model) in order to ensure proper testing of timing requirements.

Significant effort has already been invested in the automatic generation of test cases from models of the system being tested. BZ-TESTING-TOOLS [23] is a toolset for automated test case generation from B, Z, and Statechart (Statemate) specifications. Another approach is developed in the AGEDIS project [24] that uses the AGEDIS modeling language as input. The test generation engine used in this project combines the principles of TGV [25] and GOTCHA [26]. Lugato et al. [27] describe the AGATHA toolset, which overcomes the combinatorial explosion problem in software testing. In the proposed approach, test inputs are automatically generated from a test model relying on the Monte Carlo simulation technique.

Many researchers [28] [29] focus on reducing the length and number of generated test cases. These test case reduction techniques (also referred to as test case minimization in the literature) seek to reduce the number of test cases while retaining a high percentage of the original suite’s fault detection effectiveness. Most approaches to this problem are based on eliminating test cases that are redundant in some of their coverage criteria. These approaches are similar to the proposed one, since structural and functional coverage are used as criteria in deciding to reduce the length of test cases and to stop testing software.

Finally, there are few industrial papers dealing with the full software testing process (test input selection, test oracle, stop testing criteria, and test execution). Most of the research on test case generation treats simple examples that do not reflect the real complexity of modern industrial applications. In this research, the proposed MBST approach was tested on two typical automotive industrial case studies. According to Johnson Controls software experts, one of them (fuel gauge functionality) is considered to be one of the most complicated functionalities in a modern car.

4 An integrated model-based statistical approach for generating functional test cases

4.1 Overview

In this section, an integrated approach to automatically generating functional test cases for automotive embedded software is presented. Test cases can be generated offline and later executed, or they can be generated and executed online. The purposes of the proposed approach are 1) ensuring conformance to specification, 2) ensuring code coverage and 3) avoiding recurrent bugs. Through this approach, the following three software testing topics [30] were simultaneously addressed, while taking the industrial automotive context into account:

Research topic 1: Test oracle

In the automotive industry, semi-formal and formal methods are used more and more to specify software functional requirements. However, there is no standardized formalism shared between carmakers and suppliers; for each project, the supplier has to adapt its processes to the specification language used by the carmaker. Therefore, a formal framework integrating existing and appropriate description techniques was developed. This allows deriving

automated test oracles (executable software specifications) from any formalism of software specifications.

Research topic 2: Test input selection

A probabilistic test model based on Markov Chains was developed. The whole set of states of a Markov Chain represents all regarded inputs for the software being tested. Transitions between states in the Markov Chain represent orders of succession of two inputs. Each transition is associated with a number that represents the probability that one input succeeds the other and a time interval that models the wait time between two inputs that are in succession. For practical reasons, the test model is graphically represented through a matrix called transition matrix. A Monte Carlo simulation process is used to select inputs from a transition matrix.

Research topic 3: Stop testing criteria

An objective function based on formal measurement of the structural and functional coverage was developed. A constraint function in order to take test duration and cost constraints into account was also developed. An optimization algorithm monitors the generation of each test case in order to reach the test practitioner's objectives (in terms of coverage) and constraints (in terms of planning and cost). The generation of a test case is completed when the test objectives are fulfilled or the test constraints are disregarded. Indeed, the objective and constraint functions are calculated during the test case generation (after each test step generation). If structural coverage criteria are included in the test objectives, online test execution shall be chosen.

The proposed approach presents a workflow for generating test cases that is different from an investigated conventional approach in the automotive industry. The new workflow illustrated in Figure 3 is based on eight activities that are manual, semi-automatic, or automatic and that might be managed by different individuals (requirement and test practitioners). These activities are:

1. Design an automated test oracle (executable software specification) of the functionality being tested.
2. Verify (correct implementation) and validate (intended purpose) the test oracle.
3. Define some behavioral characteristics of a driver when using the functionality being tested.
4. Perform a statistical analysis on test cases developed (in the past) for similar functionalities.
5. Perform a statistical analysis on bugs detected (in the past) in similar functionalities.
6. Generate one or more transition matrices.
7. Generate executable test cases.
8. Monitor the generation of each test case using test coverage objectives and cost constraints.

The interface between the test generation and execution platforms depends on the technology of the test execution platform (computer language and environment) and on the test coverage tool used to measure the structural coverage of the software under test.

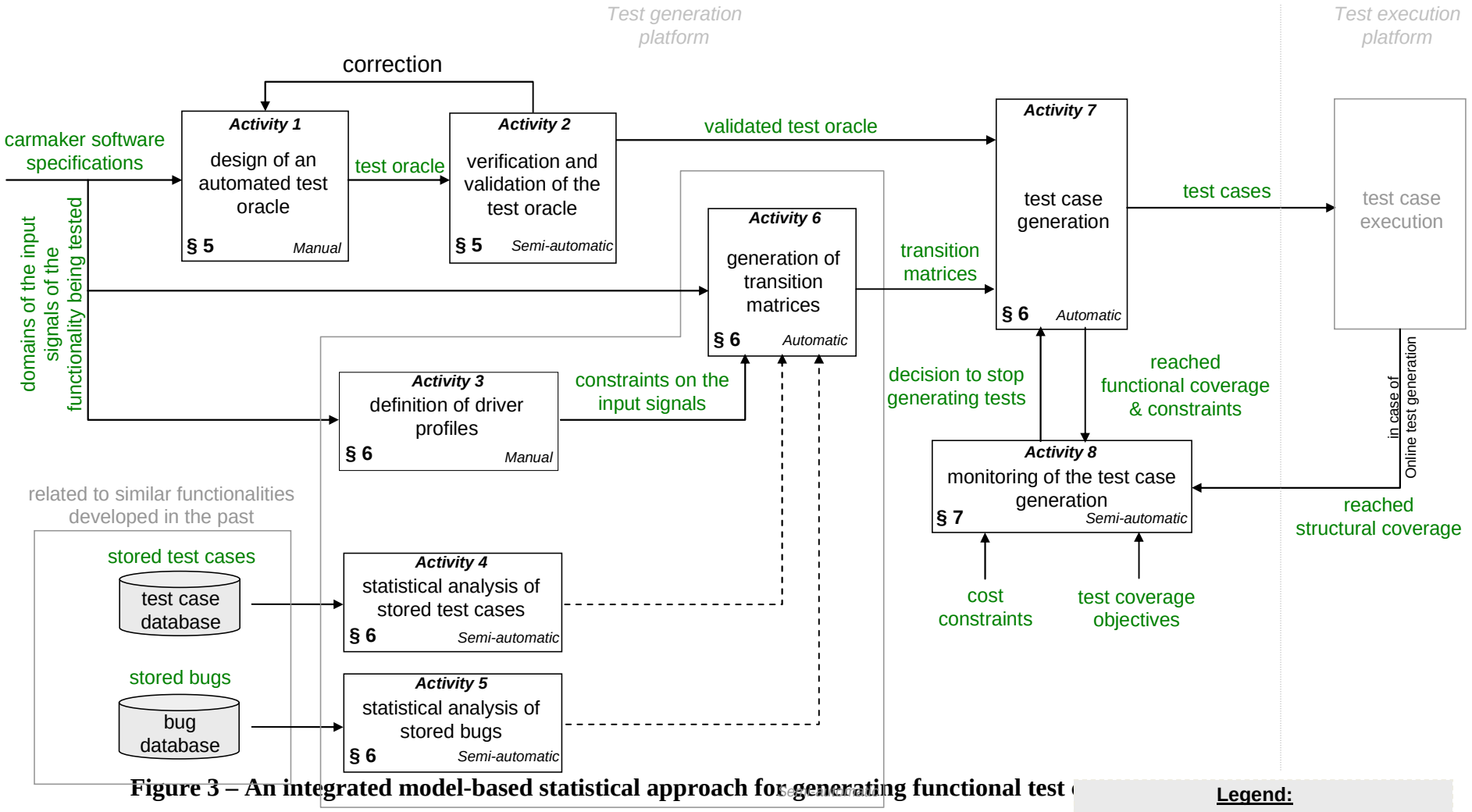


Figure 3 – An integrated model-based statistical approach for generating functional test

4.2 Human intervention

Nowadays, more and more testing techniques rely on human intervention in order to ensure their applicability in an industrial context. A compromise must be found between the relevance of the test cases (requiring significant expert intervention) and automation. The proposed approach addresses this issue in seeking more automation, with human intervention restricted to activities requiring insight. In Figure 3, human intervention throughout the proposed MBST approach is clearly identified (manual and semi-automatic). The three main manual or semi-automatic activities are:

- Design, verification and validation of test oracles (Activities 1 and 2): in current industrial practice, the software testing oracle is often a human being. In the proposed approach, practitioners must manually derive an automated test oracle from software specifications. Consistency in the designed test oracles is ensured by the semi-automatic verification and validation of the test oracle.
- Preparation of data for the generation of transition matrices (Activities 3, 4 and 5): this semi-automatic activity uses human expertise to test specific situations that can not be targeted by a systematic model coverage approach. For example, it relies mainly on test practitioner experience in order to establish end-user profiles or select stored bugs or stored test cases that may be used in the design of new transition matrices.
- Monitoring of the test case generation (Activity 8): the decision to stop testing software is completely automated with an optimization algorithm. This algorithm accounts for the fulfillment of the test objectives while respecting the cost constraints. The definition of these objectives and constraints is still manual since it is often based on informal customer and project expectations.

A technical report [31] developed roles and skills required of the practitioner for each of these manual or semi-automatic activities. Trainings, tutorials, and best practices could be developed to assist practitioners in designing relevant test oracles and transition matrices. Such an approach would be beneficial in an automotive context, as more than 50% of functionalities performed by software products are common to any series of cars. Test oracles and transition matrices could be easily reused and improved from one project to another.

5 Research topic 1: Test oracle

5.1 Literature review

Deciding whether a test outcome is acceptable is the so-called test oracle problem. Although it is obvious that a test execution for which a test practitioner is not able to distinguish between success and failure is a useless test, and although discussion of the criticality of this problem is a long existing topic in the literature [32], the oracle problem has received little attention in research, and, in practice, few alternative solutions exist to human “eyeballing”. Nardi et al. [33] highlighted the heightened interest on research related to test oracles in the last 10 years, notably after 2001.

The research literature on test oracles applicable to dynamical systems comprises a relatively small part of the research literature on software testing. Analyses proposed in earlier research are based either on the availability of pre-computed input/output pairs [34] or on a previous version of the same program that is presumed to be correct [35]. Weyuker [36] presented some of the basic problems and argues that truly generalized test oracles are often unobtainable. A survey of oracle solutions is provided by Baresi et al. [37] and Nardi et al. [33]. The survey proposes approaches to automated test oracles that are generalized in the sense that they require neither pre-computed input/output pairs nor a previous version of the system being tested. The authors group oracle systems based on implementation approaches (i.e. embedded assertions, execution log analyzers) and on the kinds of specifications they

accept (i.e. interface specifications, design models, Model-Based specifications). Four categories of oracles have been identified: specification-based, metamorphic relations, n-version and neural network. There are publications of specification-based oracles since 1991 and they represent up to 70% of the total number of publication. Examples of specification languages are: Z, Object Z, OCL, Eiffel, VDM, JML, state machine, SDL and Mitl. Kanstren [38] cited the lack of empirical studies on the use of state machines. The specification of a system provides a source of information about the correct behavior of the implementation and thus it is a valuable source for test oracles [39]. The specification can be used to describe the expected behavior of a system at different abstraction levels [40].

In current industry software testing practice, the oracle is often a human being. While the human “eyeball oracle” has advantages over more technical means of interpreting incomplete, natural-language specifications, humans are more prone to error when assessing complex behaviors or detailed, precise specifications, and the accuracy of the “eyeball oracle” drops with an increase in the number of test cases. In addition, the “eyeball oracle” becomes a limiting factor when other parts of testing are automated. Given that a test engineer can make a mistake while calculating an expected output and the large number of outputs to be compared during the test phase illustrate the obvious interest in creating automated oracles.

5.2 A framework for deriving automated test oracles from software specifications

A previous paper [2] performed a study on the evolution of languages used by carmakers to specify software functional requirements. Through this study, an increased use of formal languages and a decreased use of informal and semi-formal languages were highlighted. However, within the formal languages, there is no standard formalism shared between carmakers and suppliers. Rather, for each project, the supplier must adapt its processes to the specification language used by the carmaker. Many researchers [41] [42] state that there are no software specification languages today that fit all intents and purposes. For each context, decisions must be made as to what language (or collection of languages) is most suitable. No large-scale studies have been made to confirm the claims regarding any particular language.

Nardi et al. [33] and Baresi et al. [37] surveyed a range of frameworks for the derivation of automated test oracles from specific software specification methodologies. The main challenge posed by using a specification language is that effective procedures for evaluating the predicates or carrying out the computations they describe are not generally a concern in the design of these languages. Since there is no standard formalism for the specification of software behavior in the automotive industry, a framework to manually derive automated test oracles (executable software specifications) from any software specification language was developed. In order to avoid the propagation of the same specification error in the test oracle and implementation, test oracles shall be designed, verified and validated by another team than the one who performs the implementation. The proposed framework is general in the sense that the same designed oracle can be used for any arbitrary execution, i.e., the oracle is independent from test case selection or generation. Apart from automating the test oracle, the motivations behind transcribing carmaker software specifications into executable software specifications are 1) to avoid ambiguities, inconsistencies and misunderstandings of the carmaker requirements, 2) to explicit the behavior of the system when invalid entries are given and 3) to be able to formally measure the coverage of the software specification (coverage of the test oracle).

Practitioners have to manually derive an automated test oracle based on the specification of the functionality being tested. This manual task is the most time-consuming task in the proposed MBST approach. In section 8.2.1, the time spent in designing the automated test oracles for two sets of industrial software specifications is dicussed. Specifications that are

already expressed in a formal or semi-formal language are more obvious to interpret into the design of an automated test oracle.

5.2.1 Appropriate specification techniques

5.2.1.1 Typology of software functional requirements in automotive industry

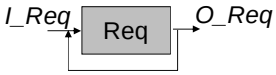
In the automotive industry, a software functionality is composed of features that are described by requirements. In this study, non-functional requirements were not taken into account; the focus was on specifying software functional requirements. A software functionality has a set of configuration *(Config)*, input *(I)*, output *(O)* and intermediate *(Int)* signals with discrete domains. Configuration signals allow for the parameterization of the software functionality (for instance, by activating or deactivating one feature). Input signals might be switches, sensors, or car environment variables (for instance, vehicle speed). Output signals might come from actuators or any type of command (for instance, the wiper motor command). Finally, intermediate signals make it possible to manage and share data between two or more features. These signals interconnect the features (F) of the functionality, and each feature is based on one or more requirements of the same type. Two types of software functional requirements were identified:

- Combinatorial (see Figure 4): when the values of the requirement output signals at instant *t* (O_Req_t) depend only on the values of the requirement input signals at instant *t* (I_Req_t).

$O\_Req_t = f(I\_Req_t)$

Figure 4 – Combinatorial functional requirement

- Sequential (see Figure 5): when the values of the requirement output signals at instant *t* (O_Req_t) not only depend on the values of the requirement input signals at instant *t* (I_Req_t) but also on the values of the requirement output signals at instant *t-1* (O_Req_{t-1}).



$O\_Req_t = f(I\_Req_t, O\_Req_{t-1})$

Figure 5 – Sequential functional requirement

5.2.1.2 Two types of specification techniques

It is potentially advantageous to use existing specification techniques, rather than inventing new ones for the sole purpose of creating test oracles. After considering a variety of techniques in the literature [42], it was decided to specify the two sets of automotive software functional requirements with the following two specification techniques:

- Decision Table (DT): DT technique is used to specify a feature that is based on one or more combinatorial functional requirements (stateless description). A DT is a table that presents a set of exclusive input signal conditions (C_q) and the corresponding set of output signal actions (A_q). Each set of conditions (C_q) represents a requirement in DT.
- Finite State Machine (FSM): FSM technique is used to specify a feature that is based on one or more sequential functional requirements (stateful description). In this paper, in order to address the particularities of embedded software, each FSM may have a timing signal *(FSMTempo)* and a set of internal signals ($FSMInt_m$). The timing signal specifies timing requirements, and the internal signals characterize the states of an FSM. The internal signals are identified by analyzing the sequential functional

requirements. They are required, when more than one state of the FSM are activated in the same time for a specific set of inputs values. An FSM is composed of:

- An initial state (S_0) and a finite number of subsequent states (S_i) with a set of actions (A_i) defining the FSM output, internal, and timing signals. The FSM timing signal is set to 0 each time the state of the FSM changes. The FSM timing signal computes the time spent in each state.
- A set of transitions (T_{ij}) from a start state (S_i) to an end state (S_j), and for each transition (T_{ij}), a set of exclusive FSM input, internal, and timing conditions (C_{ij},q). Each set of conditions (C_{ij},q) represents a requirement in FSM.

A detailed description of the conditions (Cq), actions (Aq), states (Si) and transitions (Tij) characteristics is provided in a technical report [31].

In Figure 6, a graphical illustration of an automated test oracle manually derived from software specifications is provided. The software functionality ("Auto_Light") has 3 configuration signals, 5 input signals, 2 output signals, 2 intermediate signals, and 3 features. The detailed description of the design of this test oracle is provided in a technical report [31].

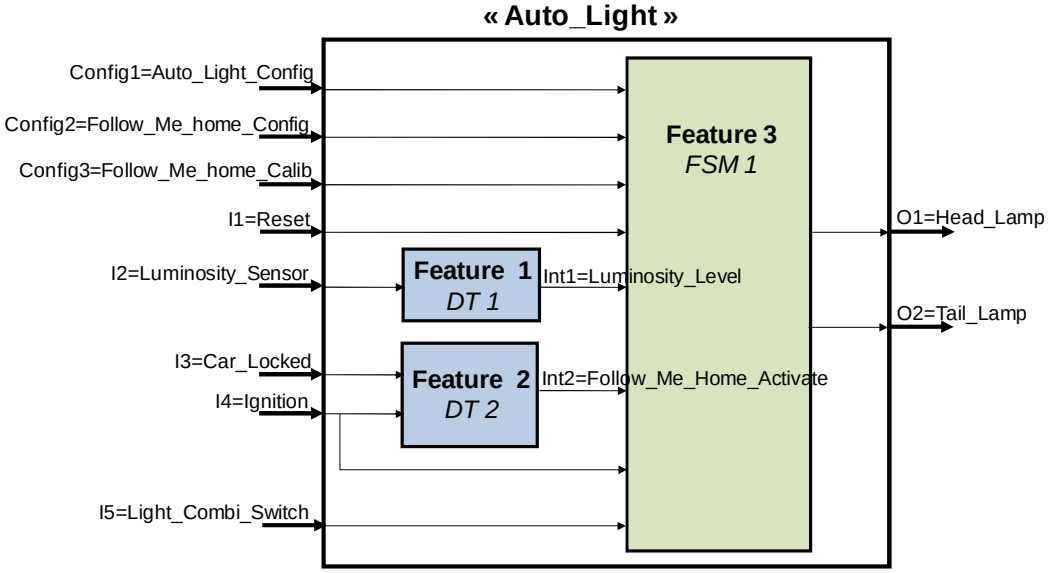


Figure 6 – Graphical illustration of an automated test oracle

5.2.2 Automation of test oracles

The expected outputs of a test are automatically predicted through an automatic run of the test oracle. This run is synchronously executed with an acyclic logic going from input to output signals of the test oracle. The run order of the features must be defined when designing the test oracle (Feature 1 then Feature 2 then Feature 3). The behavior of the test oracle is synchronized with a "clock" signal that alternates between zero and one, back and forth, at a specific pace (cycle time). The value of the cycle time depends on timing characteristics of the software functionality. It is defined by test practitioners when analyzing and designing the test oracle. At each cycle time, all the features are run following the predefined order. Running a feature consists of predicting its output signals according to its input signals.

In the case of a feature modeled using Decision Table technique, all conditions (C_q) are checked. There is no specific checking order for these conditions since only one condition can be satisfied at a time. DT output signal values are updated according to the action associated with the satisfied condition. Note that, in some cases, none of the conditions (C_q) are satisfied and therefore no DT output signal actions (A_q) are carried out. In fact, the DT conditions do not always consider all possible combinations of the values of the DT input signals.

In the case of a feature modeled using Finite State Machine technique, one state is always activated. When running an FSM, all conditions of all the transitions that start from the activated state are checked. There is no specific checking order for transitions and conditions since they are exclusive and only one condition (or only one transition) can be satisfied (made) at a time. Therefore, after having run each FSM, a maximum of one transition is made. The start state of the transition is deactivated, the destination state is activated, and output values are updated. However, in some cases, none of the transitions that start from the activated state are satisfied, the activated state remains unchanged as a result, and no FSM output signal actions are carried out. The conditions of all the transitions that start from the same state do not always consider all possible combinations of the values of the FSM input, internal, and timing signals.

A more descriptive explanation of how DT and FSM are executed to determine the expected results is provided in a technical report [31].

5.2.3 Test oracle correctness

It is often too costly and time-consuming to establish that a test oracle is absolutely valid over its full domain of intended applicability. Therefore, a contextual and semi-automatic framework to help practitioners assess confidence in a test oracle and decide in this way whether or not it is possible to stop verifying and validating it was developed. Tests and evaluations are conducted until there is sufficient confidence that a test oracle can be considered valid for its intended application. Sargent [43] proposes a simplified way of designing and validating a test oracle. The Problem Entity is the system (real or proposed), idea, situation, policy, or phenomena to be modeled; the Conceptual Model is the mathematical/logical/verbal representation of the Problem Entity, developed for a particular study; and the Computerized Model is the Conceptual Model implemented on a computer. The Conceptual Model is developed through an analysis and modeling phase, the Computerized Model is developed through a computer programming and implementation phase, and inferences about the Problem Entity are obtained by conducting computer experiments on the Computerized Model in the experiment phase.

The main three model verification and validation stages proposed by Sargent are:

1. Conceptual Model Validity (i.e. clarification of the carmaker's needs and requirements), ensuring that 1) the underlying theories and assumptions of the Conceptual Model are correct, and 2) the model representation of the Problem Entity and the model's structure, logic, and mathematical and causal relationships are "reasonable" for the intended purpose of the model.
2. Computerized Model Verification (i.e. check of the model programming rules), ensuring that the computer programming and implementation of the Conceptual Model are correct.
3. Operational Validity (i.e. check of the model accuracy), concerned with determining that the model's output behavior has the accuracy required for the model's intended purpose over its intended domain of applicability. This is where most of the validation and evaluation techniques take place.

More than 77 verification and validation techniques for simulation models are identified and classified by Balci [44]. Most of these techniques come from the software engineering discipline, and the others are specific to the modeling and simulation field. Unfortunately, no algorithms or procedures exist to decide which techniques to use. In the next three sections, techniques, rules, and scenarios to help modelers in validating the Conceptual Model, verifying the Computerized Model, and finally checking the Operational Validity of a test oracle are presented. The proposals take both Sargent's recommendations and the industrial context of this research into account.

5.2.3.1 Conceptual Model validity

A Conceptual Model of the test oracle is developed through analysis and modeling of a software specification as it was delivered by the carmaker. For each software functionality and based on the carmaker software specifications, modelers draw a sketch of the test oracle by:

1. identifying the input and output signals and their domains;
2. grouping the functional requirements according to their types (combinatorial or sequential);
3. identifying the features (DT and FSM) and the intermediate signals and their domains;
4. and finally, specifying each feature: for a DT, identifying the conditions and their associated actions; for an FSM, identifying the states and their associated actions, the transitions and their associated conditions, and, if needed, the internal and timing signals.

Once the Conceptual Model of the test oracle is designed, each feature and the test oracle must be manually evaluated to determine if they are reasonable, correct, and complete in terms of the carmaker's requirements. The Face validity and Turing tests [44] may be used in order to clarify the carmaker's needs and requirements and validate the conceptual model; expert knowledge is the main basis for this validation. Individuals knowledgeable about the system being tested are asked to judge the test oracle against the carmaker's software specification and to give their level of confidence in the test oracle and/or its behavior.

5.2.3.2 Computerized Model verification

The Computerized Model of the test oracle is developed through computer programming and implementation of its Conceptual Model. A high-level graphical language [31] to help modelers computerize their Conceptual Models of test oracles was developed in a technical report. The use of a graphical language generally results in fewer errors, and programming time is usually reduced significantly. Moreover, in order to detect all the programming errors and ensure that a valid computer model of a test oracle is obtained, a set of integrity rules [31] to be checked automatically against this computer model was developed.

5.2.3.3 Checking Operational Validity

Computerized Model verification ensures that mistakes have not been made in the computer implementation of the test oracle. It does not ensure the compliance of the test oracle with the (original) carmaker requirements. The Operational Validity stage aims to ensure that the test oracle behavior is compliant with the carmaker's requirements and has the accuracy required by the carmaker. To do this, computer experiments must be conducted on the Computerized Model of the test oracle. This is where most of the model deficiencies are detected. There may be errors in the Conceptual Model of the test oracle or programming errors in its computerization. Three possible actions [31] to help modelers validate the Computerized Model of a test oracle against its original requirements were identified. These actions are semi-automatic and can be carried out concurrently (when all the input data are available) or separately:

1. First action: have experts (in the software functionality under test) run the Computerized Model
2. Second action: execute the test cases delivered by the carmaker on the Computerized Model
3. Third action: execute a set of test cases on the carmaker's software specification (in case of executable specification) and the Computerized Model in order to compare the two.

The principles, grammar, and validity of the proposed test oracle model are discussed in detail in a technical report [31].

6 Research topic 2: Test input selection

6.1 Literature review

Selection of the most suitable test inputs to be executed in the software being tested is a complex problem that has inspired much research, because selection of the test cases greatly influences test efficacy. Many researchers have proposed criteria for picking out a “good” sample of potential test cases. A comprehensive survey of the research on this topic was done by Zhu et al. [45]. An important point is that a “good” test case is not universally “good” but rather depends on the testing context (time and resource constraints, etc.), the software being tested (criticality, etc.), and the testing goal (100% structural and/or functional coverage, increase in confidence, etc.). The most common interpretation for “good” would be “able to detect a high number of bugs”. Basili [46] and Wood [47] experimentally observe that different test selection techniques could ensure different test purposes. Therefore and while having more than one test purpose, it may be preferable to apply a combination of diverse techniques, rather than focusing on just one.

It is difficult to find a system for classification of all test selection techniques. The one proposed in Bertolino et al. [48] may be seen as a compromise. It is based on how tests are generated from test practitioners’ intuition and experience, the specifications, the code structure, the faults to be discovered, and, finally, the nature of the application.

Paradoxically, test input selection seems to be the lowest priority problem for test practitioners in automotive industry. A demonstration of this low priority is the paucity of commercial tools that aid test input selection [30], in comparison with the large quantity of support tools that handle test execution, regression, and documentation. Much progress has been made in test input selection techniques over the last twenty years, but this progress remains almost unknown in the automotive industry. The most-practiced test selection technique is still dependence on the expertise of the tester.

6.2 A test model based on Markov Chains

The proposed test model (transition matrix) does not model dynamic probabilities. The probabilities in a transition matrix are static and pre-defined (before testing). In this research, the same assumption as Bauer et al. [9] was made. They use Markov Chains to model a car’s operational software system and assume that future inputs only depend on current, and not past, inputs. In some cases, this assumption proves incorrect. For instance, if a gearbox is in 4^{th} gear, the probability of a shift to 5^{th} gear will depend on what gear it was in before it went into 4^{th} (it's more likely to go 3^{rd}->4^{th}->5^{th} than to go 5^{th}->4^{th}->5^{th}). To overcome this problem, the state space should be expanded. This leads to another problem: having to estimate many more transition probabilities. An exhaustive test selection approach should consider the probabilities associated with a given sequence of each pair of N possible inputs, where N is the total number of possible inputs of the functionality being tested. In this paper, test practitioners have the option of designing additional constraints on a transition matrix in order to give weight to a sequence of more than 2 inputs, taking into account the dynamic nature of software behavior. This sequence might be more likely (from past experience) to contain a bug, or simply a sequence often performed by the end-user of the product.

6.2.1 Characteristics and illustration

A specific class of Markov chains, discrete-parameter, finite-state, time-homogenous, irreducible Markov chains, has often been used to model the usage of software. These Markov chains are structurally similar to finite state machines and can be thought of as probabilistic automata. The body of literature on Markov chains in software testing is substantial. Work done on testing particular systems is detailed by Avritzer and Larson [12].

A Markov chain is described as follows: considering a set of states $S = \{S_1, S_2, \ldots, S_r\}$. The process starts in one of these states and moves successively from one state to another. Each move is called a transition. The controlling factor in a Markov chain is the transition probability p_{ij}, a conditional probability that the system will go to a particular new state *Sj*, given the current state *Si* of the system. The system can remain in the state it is in, and this occurs with probability p_{ii}. In the context of this research, each state represents a possible input and each transition is associated with a probability associated with a particular sequence of two inputs (linked states). Of those included in the reviewed literature, none of the researchers integrates the wait time between two inputs into the software usage model. However, in many embedded software systems, this transition time between two inputs plays a major role in detecting bugs relating to real-time constraints. As a consequence, each transition of the transition matrix is associated with a time interval from which transition time are selected. Moreover, all the sequences of inputs that can occur from an electronic (hardware) point of view could be taken into account, even if they are illogical from the software behavior point of view, as a malfunctioning of electronics (sensors, etc.) could cause an unexpected sequence of test inputs.

In this paper, a Markov chain is represented as a square matrix (called a transition matrix) with the states as indices and the transition probabilities as entries, to which a time interval was also added. Considering a software functionality being tested with 3 input signals: *I1, Domain = {0, 1}*; *I2, Domain = {1, 2, 3}*; *I3, Domain = {0, 1}*. The template of the transition matrix for this functionality is illustrated in Figure 7. It is a 7-by-7 matrix where 7 is the number of all possible values of the functionality input signals *(I1, I2 and I3)*. For each entry in the matrix, two pieces of information are required:

1. The transition probability, i.e. the probability that the two inputs are in sequence. The total of the probabilities in a row must be equal to 1. After selecting an input *(row: I3=1)*, test practitioners may either select the same one again *(column: I3=1)* or select another one *(column: I1=0, I1=1, I2=1, I2=2, I2=3, I3=0)*.
2. The transition time between the two inputs, modeled as an interval of possible values $[T_{min}; T_{max}]$ with a uniform probability of being selected for the test.

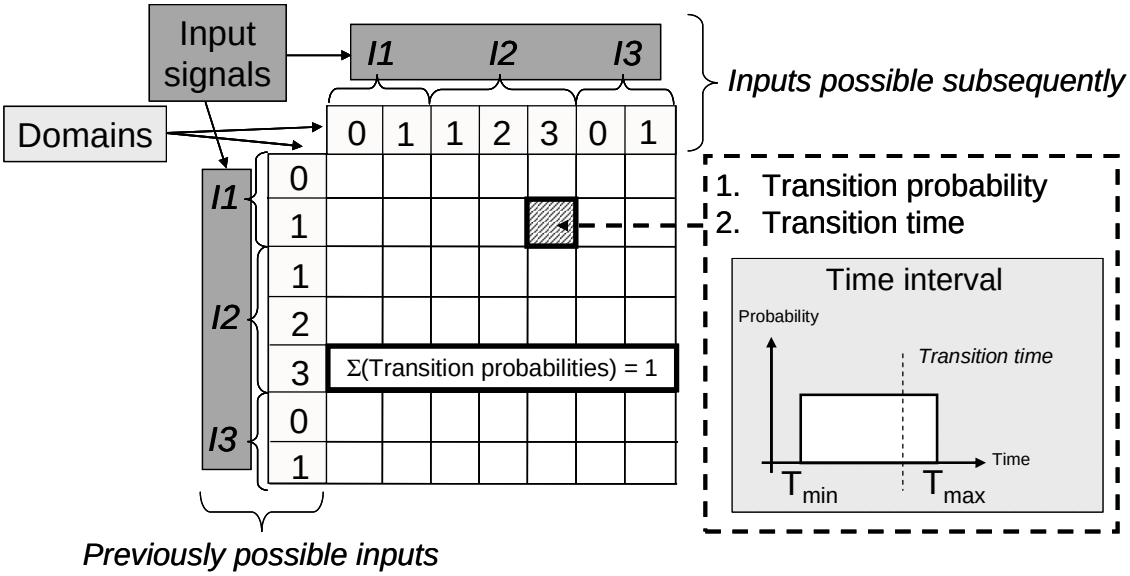


Figure 7 – An example to illustrate the transition matrix

For each software functionality being tested, test practitioners may design one or more transition matrices that illustrate the dynamic behavior of the functionality in different usage circumstances.

6.2.2 How to design a transition matrix?

One major question is: how can a practitioner design a transition matrix? The basic solution is to manually fill in each entry of the matrix with a transition probability and a time interval. However, a functionality can have more than 20 input signals and 100 possible values for these signals. Consequently, a transition matrix can easily reach 10000 entries, which becomes inconceivable to fill in manually!

Whittaker [49] discusses manually building finite state models in a hierarchical fashion. El-Far [50] describes a framework for automatically building finite state models from an informal specification of the system being tested. There are also some works [51] on automating the generation of probabilities for Markov chains. In this paper, a semi-automatic process to design transition matrices for software functionalities is developed. The designed models may be based on random assumptions, on the end-user's behavior, or on the test practitioner's experience from previous or similar development. The design of a transition matrix does not require much human intervention. In section 8.2.2, the time spent in designing a set of transition matrices for two typical industrial case studies is discussed.

The **first step** involves a manual analysis to classify the inputs of the functionality being tested into subpopulations. Classification is based on input type:

- Configuration and calibration: parameters of the functionality
- User: user inputs, actuators
- System environment: internal variables
- Sensor: sensor inputs

The **second step** consists in manually selecting a sample of inputs from the continuous input domains (i.e. temperature, speed). Many existing methodologies [52] focus on the sampling problem by using heuristics to determine which input values to consider. In many of these methodologies, no real attempt is made to measure whether usage of the software that falls outside the sample will succeed. A notable exception is category and partition testing [53], in which inputs are partitioned into equivalence classes whose points are equally characteristic of the functionality being tested, and it is therefore sufficient to test one representative input from each class. In this study, the partitioning strategy is concurrently based on the software specification, the code structure, and experts' knowledge:

- A specification-based partition might divide the input domain into inputs required to invoke one or several software features.
- A code-based partition might consider inputs that do or do not force use of a potentially bugged data structure (i.e., using boundary values, or using at least one even and one odd value as inputs, etc.).
- An expert-based partition might consider inputs that have a high probability of occurring from a usage point of view. For instance, when sampling the "vehicle speed" variable, it is judicious to select more values around 50, 90 and 130 km/h since these values are the most used in France (French speed limits).

Having classified and selected a sample of the inputs of the functionality being tested, the **third and last step** consists in automatically generating one or more transition matrices with all possible inputs (all inputs after partition) in columns and in rows. The entries (transition probabilities and time intervals) of these matrices are based on one of the assumptions discussed shortly below and derived from the work of Bertolino et al. [48]. As highlighted in Figure 20, these assumptions are complementary and no single type of transition matrix would be able to detect all the bugs.

A detailed description of how each transition matrix is populated is provided in a technical report [31]. It discusses basic software routines that attribute probabilities and time intervals between successive test inputs, taking into account assumptions and constraints.

6.2.2.1 Random assumptions

The aim of a transition matrix based on random assumptions is to test the software against unrealistic input sequences. The two types of transition matrix presented in this section are unrealistic from a usage point of view (many of the transitions are not possible). However, through these matrices, completely random input sequences could be generated and therefore test the robustness of the software against abnormal behavior in the inputs.

One solution is to consider that all the sequences of input signals values to the functionality being tested are possible and have the same probability of occurrence. This is what is called the Nominal 1 transition matrix. Another similar solution is to consider that all the sequences of input signals values are possible and that all the input signals have the same probability of undergoing a change. This is what is called the Nominal 2 transition matrix. Since the aim of these two transition matrices is to test the behavior of the software against abnormal inputs, it is judicious to choose a practical time interval and eliminate any malfunctioning that may be caused by transition time.

6.2.2.2 End-user profile

There is no better way to test a product than to test it in the way that it will be used. The main work in this field is that of Musa [54]. He presents a case for using the operational profile in software reliability engineering. In this paper, a framework in order to generate test cases that simulate the behavior of the end-user of the functionality is developed. Four types of constraints were defined. These constraints can be instantiate by test practitioners as many times as they choose on one or more input signals of the functionality being tested (operational profile). The four types of constraints are:

- Logical constraint: This constraint prohibits an input signal from switching between values that are illogical from a usage point of view.
- Conditional constraint: This constraint characterizes the correlation between two or more input signals that do not have any succession conditions on the inputs of these signals. In other words, when one or more inputs satisfy specific conditions, the domain of other inputs is adapted (reduced) automatically.
- Succession constraint: In practical use of an electronic product, two or more inputs may have a high probability to chronologically follow one another (and sometimes necessarily do follow one another). Through this type of constraint, such sequential inputs are modeled.
- Time constraint: Johnson Controls software experts agree that the time interval between inputs plays a major role in bug detection. Either two specific inputs can be executed within a specific time interval or a single specific input can be executed during a specific time interval. Through this type of constraint, such specific timing behavior is modeled.

These constraints are static and independent. They aim at reducing the number of possible combinations of input signals and more rigorously pinpointing the combinations that are frequently seen once the product is launched on the market. Once identified and manually designed by test practitioners, they are automatically implemented into a transition matrix called End-user Profile transition matrix. In other words, the entries (transition probabilities and time intervals) of this matrix are automatically defined based on the constraints that the test practitioners have already set on each input signal. An End-user Profile transition matrix makes it possible to generate test cases where illogical (from the end-users' viewpoint) sequences of inputs are eliminated and typical sequences of inputs are favored.

6.2.2.3 Management of stored bugs

In the proposed approach, stored bugs are resued in order to generate test cases that prove the non-existence of recurrent bugs. When testing a functionality in a new project, test practitioners may go to a database and select all bugs detected in this functionality in previous projects. Each bug is automatically translated into a transition matrix called Bug transition matrix, where sequences of inputs that reveal the recurrent bugs are favored. This proposal is mainly based on the assumption that a standard formalism (see Figure 8) is used to describe the initial conditions and the sequential inputs that lead to detection of a bug. In Figure 8, an illustration on how the "problem description" attribute of a bug should be described is given. In Step 7 of this problem description, the observed output values are different from the expected values (this is a symptom of the bug). A glossary of the input signal names used in the previous and current projects is also necessary. The test cases generated from Bug transition matrices make it possible to check for the bugs that were detected in the past, in order to see if they are present or not in the current product.

Problem description	
Initial inputs values	I1=1; I2=1; I3=0
Step 1	
Test input #1	I1=0
Transition time (ms)	50
Expected outputs values	*O1=0; O2=0*
Observed outputs values	*O1=0; O2=0*
Step 2	
Test input #2	I1=1
Transition time (ms)	200
Expected outputs values	*O1=0; O2=0*
Observed outputs values	*O1=0; O2=0*
. . .	
Step 7	
Test input #7	I3=0
Transition time (ms)	150
Expected outputs values	***O1=1; O2=0***
Observed outputs values	***O1=1; O2=1***

Figure 8 – Problem description of a stored bug

6.2.2.4 Reuse of stored test cases

Using stored test cases seems to be beneficial in the automotive context, since more than 50% of functionalities performed by software products are common to any series of cars. In the proposed approach, test cases developed in the past are automatically analyzed and a transition matrix called Test Case transition matrix is automatically generated for each functionality. This matrix attributes high probabilities to the succession of inputs regularly executed in the stored test cases. It also contains the set of time intervals applied between each pair of inputs. Consequently, when generating test cases from these matrices, test scenarios will be based on the experts' experiences. The proposal to reuse existing test cases from previous projects is based on the assumption that a unique test case format is used. This format is independent from the test execution platform and is defined in a technical report [31]. The proposed approach uses also this format for the generated test cases. A glossary of the input signal names used in the previous and current projects is also necessary.

6.3 A test generation algorithm

Automatically generating a test case from a transition matrix requires generating a set of test steps until a stop criterion is reached. Test cases can be generated offline and later executed, or they can be generated and executed online. Online generation of tests means that the test generation tool is directly connected to the software under test and tests it dynamically (see Figure 3). Each generated test step is directly executed on the software under test. This assumes that the interface between the test generation and execution platforms is setup.

In what follows, a detailed description of how a test step is designed is given. Designing a test step requires the selection of an input and transition time, and the prediction of expected results that will be checked against the output signals of the software being tested. In the proposed approach, two automated activities are necessary to generate a test step; they are discussed hereafter and illustrated through an example in Figure 9.

6.3.1 Activity 1: Perform a Monte-Carlo simulation

In order to choose an input and a transition time, a Monte Carlo simulation is automatically performed on a transition matrix. Two steps are required:

- Step 1: an input is chosen according to its transition probabilities in the transition matrix. This technique is known as the statistical testing technique and was developed decades ago (see Marre et al. [55] for a thorough description). Before starting the generation of a test case, the input signals of the software being tested are set to specific values (initial values). Therefore, the starting input of the test case may be 1) randomly chosen among these initial values or 2) chosen by a test expert in order to favor a specific sequence of inputs at the beginning of the test case. The test practitioner may set probabilities in the transition matrix that are intended to drive the selection of specific inputs. In practice, this may be useful when the software being tested requires specific conditions (engine switching on, etc.) in order to be completely functional.
- Step 2: a transition time is randomly chosen within the time interval of the selected test input. Test coverage in the transition time between two inputs could be ensured by several test cases.

6.3.2 Activity 2: Run the test oracle (executable software specification)

The chosen inputs are set as the input signals of the test oracle, and a run of the test oracle (synchronized with the cycle time of the “clock” signal) is performed until the transition time has expired. The values of the output signals of the test oracle are the expected results of the test step being designed.

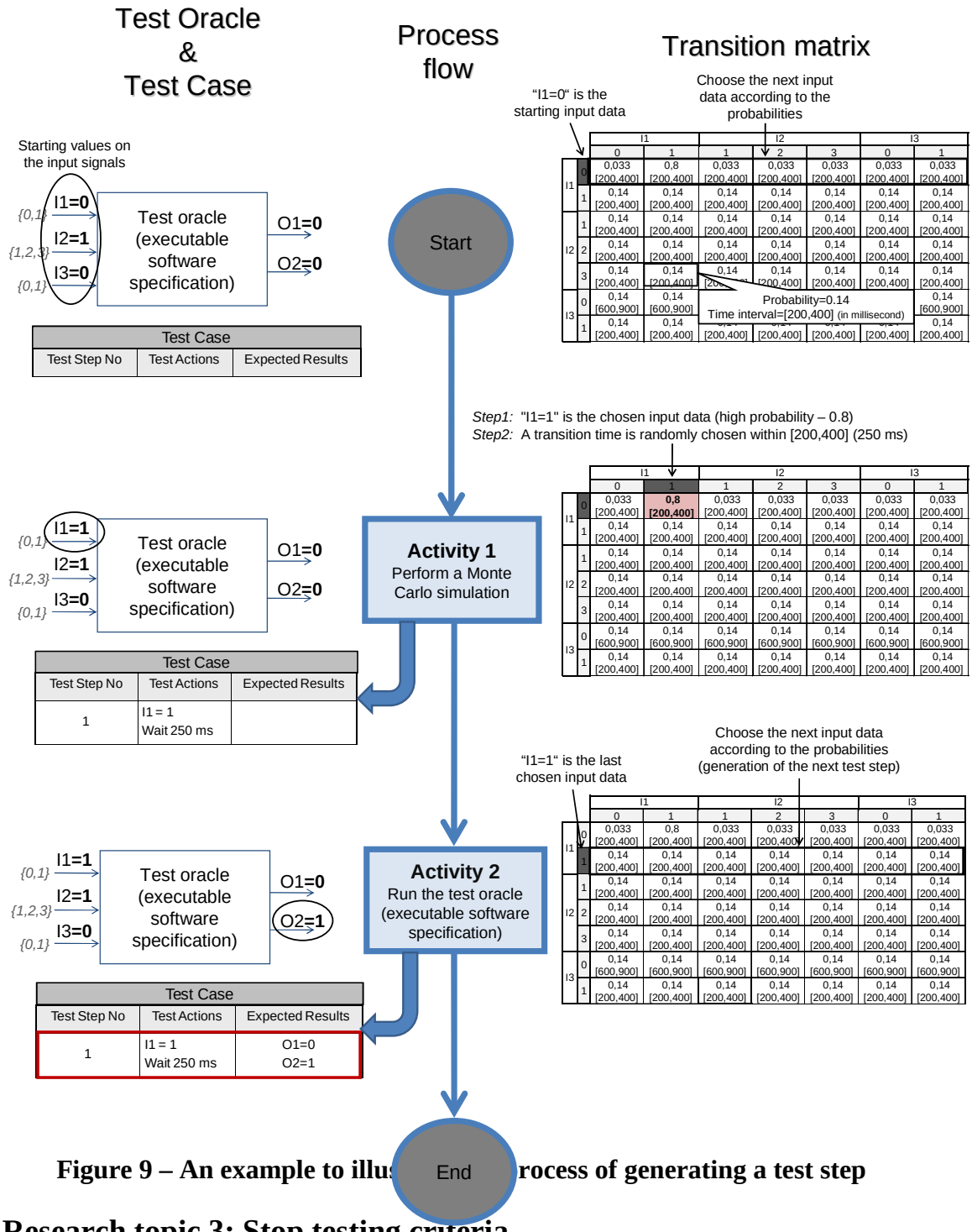


Figure 9 – An example to illus rocess of generating a test step

7 Research topic 3: Stop testing criteria

7.1 Literature review

Exhaustively testing software and being sure that it is bug free remains a major problem from a computational point of view. In other words, it is very complex, even impossible, to test all the inputs, combinations of inputs, and paths of a software. Several stopping criteria are proposed in the software testing literature. A stopping criterion based on stochastic similarity is proposed by Whittaker [56] and refined by Sayre [57]. A stopping criterion based on estimated reliability and confidence is proposed by Littlewood [58]. A cost-benefit stopping criterion based on estimates of the errors remaining in the product and the cost to repair them both before and after release is proposed by Dalal [59]. A more sophisticated version which includes costs due to lost business and customer dissatisfaction is proposed by Chavez [60]. And finally, stopping criteria based on test coverage are presented by Offutt [61].

Determining when to stop testing and release software is an important management decision. There is always a necessary compromise between the decision to continue testing or to stop: (a) if testing stops too early, many bugs remain, and thus supplier incurs losses due to customer dissatisfaction and the cost of later bug-fixing – the cost of fixing a bug after release is higher than the cost of fixing it while testing; (b) if testing continues up to the maximum allowable time, then there is the cost of testing effort and loss of business. In an industrial context, software testing is often based on specific assumptions and objectives that help test practitioners and managers to decide when to stop the testing protocol.

7.2 An aggregate stop testing criterion

In the automotive industry, structural and functional coverage of software are major quality indicators required by carmakers. In order to monitor the automatic generation of each test case:

- an objective function based on formal structural and functional coverage;
- a constraint function based on test execution cost;
- and an optimization algorithm which aims to fulfill the test objectives while respecting the cost constraints.

were developed. Consequently, a panel interface (see Figure 16) that allows test practitioners to set the test generation objectives and constraints were proposed. A set of weights *(wi)* that test practitioners may apply to each defined objective or constraint: 0 (to be ignored), 1 (not very important), 5 (important), 10 (very important) were also defined. The panel helps test practitioners express their objectives and constraints in terms of the required test coverage and cost and therefore generate test cases fulfilling their expectations.

The objective function, $F_{Objective}$, is defined as:

$$F_{Objective} = \underbrace{\sum_i \left| StrucCovTarget_i - StrucCovCurrent_i \right| \times w_i}_{\text{Structural coverage}} + \underbrace{\sum_j \left| FuncCovTarget_j - FuncCovCurrent_j \right| \times w_j}_{\text{Functional coverage}}$$

where $StrucCovTarget_i$ and $FuncCovTarget_j$ are the coverage goals as defined by the test practitioners, $StrucCovCurrent_i$ and $FuncCovCurrent_j$ are the coverage ratios reached by the test case under design, and $w_i s$ are weights. The structural coverage is measured in terms of statements, procedures, conditions and decisions coverage of the tested software. The functional coverage is measured in terms of elements (DT and FSM), signals domains and transition matrices coverage of the formal specification (test oracle). The structural and functional coverage are expressed in terms of ratios of coverage and are then normalized in order to reach a value of 100%. A detailed description of the structural and functional coverage metrics is provided in a technical report [31].

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

$$F_{Constraint} = \sum_k \left| ConsTarget_k - ConsCurrent_k \right| \times w_k$$

where $ConsTarget_k$ are the values of the constraints as defined by the test practitioners, $ConsCurrent_k$ are the values of the constraints in the test case being designed, and $w_k s$ are weights. When generating a test case in the proposed approach, test practitioners can set a group of cost constraints to be respected:

- Constraint 1: Execution time. The time that a test practitioner will spend in manually executing the generated test cases on the software product. For instance, if the test practitioner has 1 person day (pd, where 1 pd = 8 work hours) to manually execute the generated test cases, the execution time (i.e. the total of all the transition time) of these test cases should not exceed 28800000 ms (8h x 60m x 60s x 1000ms).
- 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 inputs.

In order to have a consistent aggregate constraint function ($F_{Constraint}$), the cost constraints were normalized to 100%. These constraints are expressed in milliseconds (ms) and in number of generated test steps, respectively. In the following, the normalization process of these constraints is illustrated through an example. Each time test practitioners decide to set a constraint *k*, the normalized target of this constraint $ConsTarget_k$ is immediately set to 100%. For instance, once a test practitioner decides to generate a test case for which the total execution time does not exceed 108000 ms (*target_constraint_value*), the normalized target of the test execution time constraint is set to 100% (*ConsTarget(target_constraint_value)* = 100%). After generating a set of test steps, the normalized current value of this constraint ($ConsCurrent_k$) 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 (*current_constraint_value*), *ConsCurrent(current_constraint_value)* is assessed to be (21600*100)/108000 (*ConsCurrent(current_constraint_value)* = 20%).

Throughout the proposed approach, the automatic generation of tests (performed used a Monte Carlo simulation process) is monitored by an optimization algorithm based on a combination of simulated annealing and look-ahead strategies [62]. The aim of this optimization algorithm is to reach the test coverage objectives in the most efficient manner possible while respecting the cost constraints as much as possible. During a test case design session and after each test step design, functional coverage of the formal specification (test oracle) is assessed. The coverage rate of the transition matrix from which the inputs have been selected is also considered. If the designed test step does not contribute to functional coverage, it is rejected, and a new test step is designed. In the case of online test case generation, the retained test step is executed on the software product being tested, and the structural coverage is updated. At the end, the objective and constraint functions are assessed. As the test coverage objectives may be fulfilled in different orders, the first objective fulfilled does not immediately stop the process. The process is stopped when one of the following criteria is met:

(1) The objective function ($F_{Objective}$) is equal to zero. In other words, the target coverages are reached.

(2) 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 this case, additional test cases should be generated. The one that fulfills the test objectives shall be selected. If none, the test coverage of the generated test cases could be combined.

8 Implementation, validation, and impact of the proposed approach in a real industrial context

In this section, the proposed approach is assessed on real industrial data coming from an automotive electronic supplier called Johnson Controls. Two industrial case studies with historical data were considered. Each case study considers one software functionality that has already been developed and validated (unit and conformance testing) in the past with the V&V techniques currently used in the automotive industry and developed in section 2.2. For each delivery to the carmaker with the software functionality updated, historical data on the time spent to validate this functionality and on the bugs detected by the supplier and by the carmakers are available. The first version of the two software components (corresponding to the two functionalities) as they were delivered for the first time by the development team to the validation team was treated. The version of the carmaker requirements of this functionality at the moment when the software components were delivered for the first time to the carmaker

was also analyzed. For each functionality, at unit testing level, the proposed model-based statistical approach was executed to automatically generate functional test cases, and the performance of this approach was assessed against the existing one. The test execution was automated in a simulated environment [2]. Since the interface between the test generation and execution platforms was not yet developed, test cases were generated offline and later executed. The performance of the proposed approach was quantitatively measured using two metrics often used by carmakers to assess their suppliers' capability: 1) the number of bugs detected downstream in the process (after delivery to a carmaker) and 2) the time spent before delivering the software (testing and debugging time).

8.1 Introduction

8.1.1 Selection of the two software functionalities

Selection of the software functionalities is delicate, and many criteria guide this choice (products, carmakers, management teams, development teams, validation teams, levels of complexity, and software specification languages). The heart of the proposed approach is the design of an automated test oracle based on an executable software specification. Therefore, one important criterion in choosing the functionalities in the two case studies was that they exemplify the diversity of carmaker software specification languages. These case studies would prove that whatever the language used by the carmaker to specify their software functional requirements, the proposed approach may be used to automatically generate test cases. Based on this criterion, the front wiper functionality of a body controller module and the fuel gauge functionality of a car dashboard were chosen. A body controller module is an electronic product that manages the entire internal circuit of a car (door lock, lights, electrical windows, etc.), and a car dashboard is a control panel located under the windshield of the car. These two products were developed for the same carmaker but not for the same car platform, and therefore the carmaker's integration teams were not the same. The software functional requirements of the front wiper were specified in a formal language (Statechart) while those of the fuel gauge were specified in an informal and natural language (textual language). An excerpt from the software functional requirements of the front wiper is given in Figure 10 (Input/Output variable list) and Figure 11 (Expected behavior specified using a state machine).

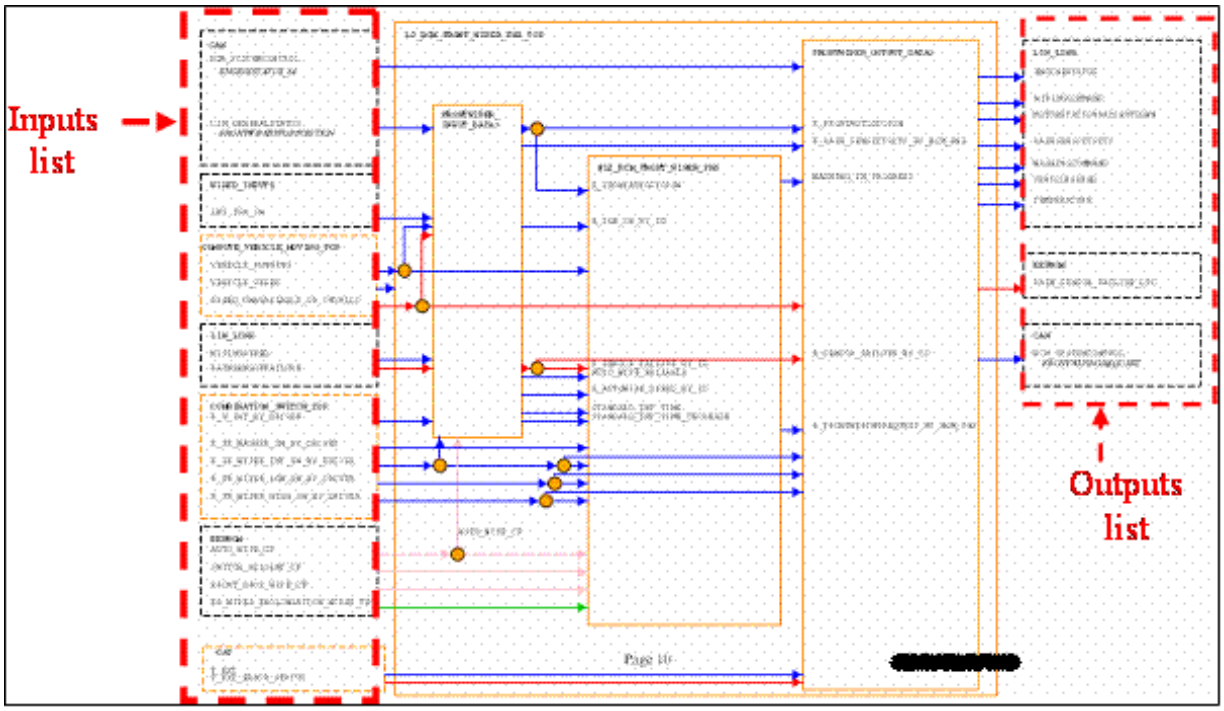


Figure 10 – Input/Output variable list of the front wiper functionality (this figure is voluntarily fuzzyfied for confidentiality reasons)

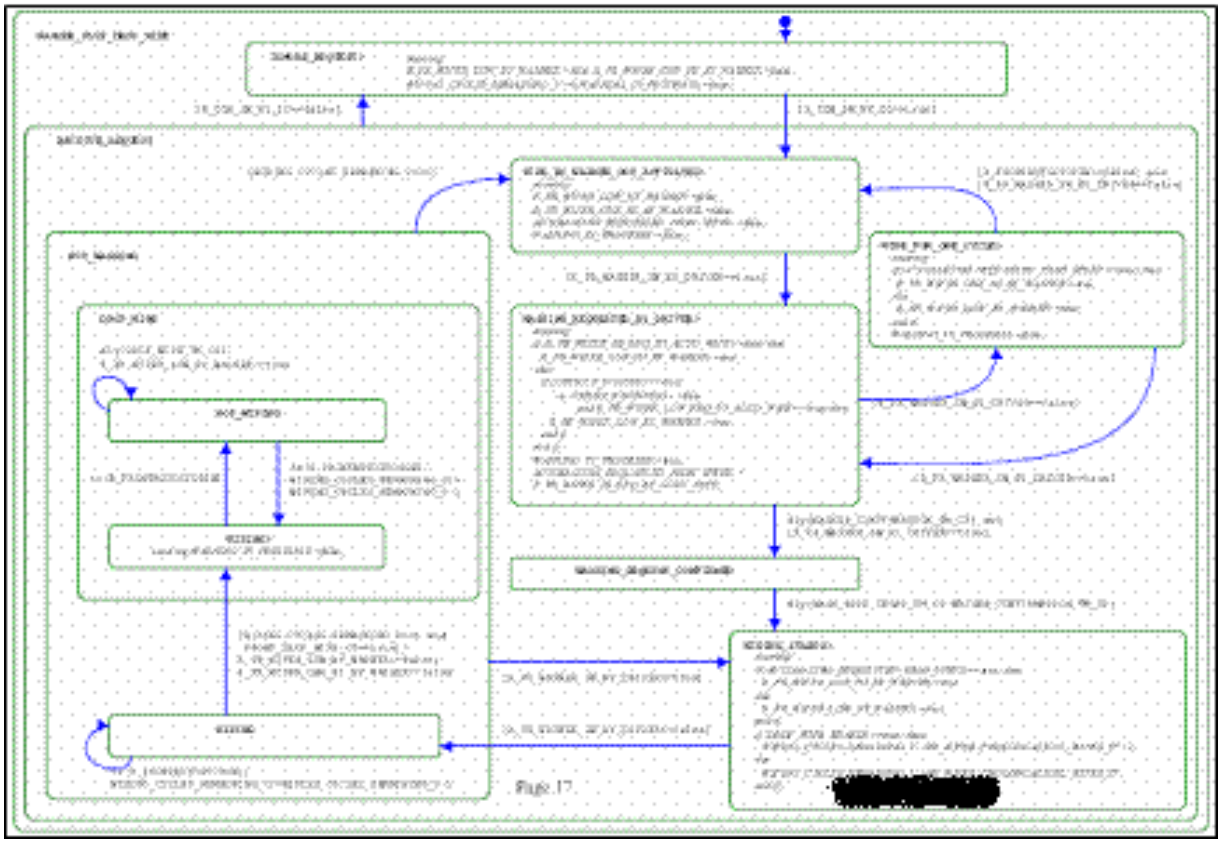


Figure 11 – An extract from the expected behavior of the front wiper functionality as illustrated by the carmaker (this figure is voluntarily fuzzyfied for confidentiality reasons)

The respective sizes of the software components developed for these two functionalities were 1229 and 1500 Lines of Code (LOC). These functionalities were validated in 2005 and 2006, respectively, by two different teams of the automotive electronics supplier in two different locations (countries).

8.1.2 Historical data for the conventional approach

8.1.2.1 Bug detection

The distribution of bugs detected in the two functionalities using the conventional approach is illustrated in Figure 12. These bugs are related to the internal behavior of these functionalities. Between the first and last deliveries to the carmaker, inclusive, 22 bugs were detected in the front wiper software functionality, and 23 bugs in that of the fuel gauge. These bugs were detected in the two functionalities before (by supplier testing) and after (by carmaker testing) the deliveries to the carmaker. Considering the front wiper example in Figure 12; 17 bugs were detected in the supplier testing phases and 5 bugs by the carmaker after intermediate delivery. It must be noted that, after developing the front wiper software functionality for the first time, only 12 bugs were detected during the first testing phase. Therefore, a delivery ensued, and the carmaker immediately detected 2 more bugs. In the meantime, before the second delivery, test practitioners tried to improve their existing test cases and design new test cases. Consequently, they were able to detect one more bug; after the second intermediate delivery, no new bug was detected by the carmaker. For the fourth intermediate delivery, no new test cases were developed. The complete scenario of bug detection through the last delivery to the carmaker for the two functionalities is summarized in the histograms in Figure 12.

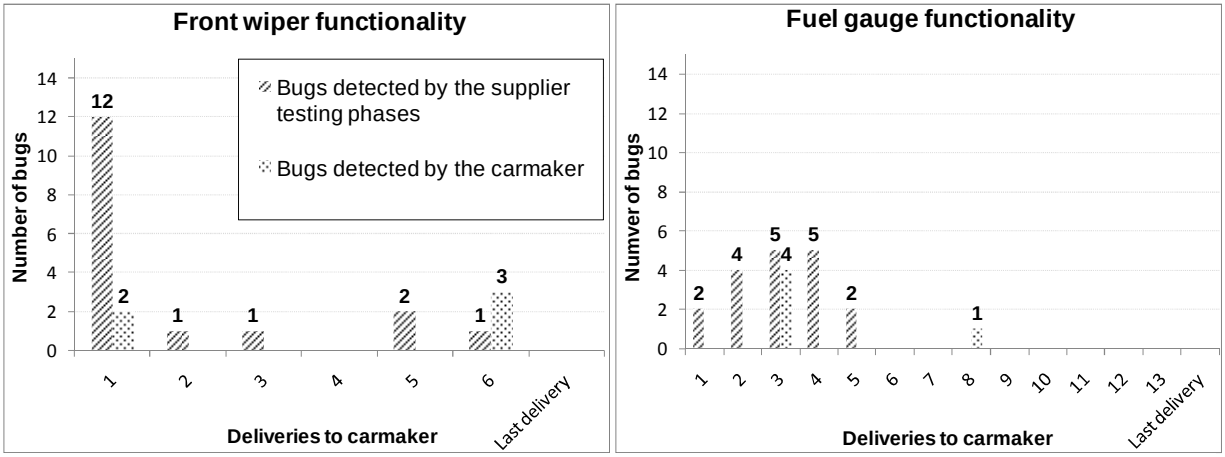


Figure 12 – Distribution of bugs detected throughout the deliveries to the carmaker – conventional approach (manual test design)

Among the bugs detected in the two functionalities, 5% are *Coding and typographical*, 45% are *Control flow and sequencing*, 25% are *Data definition, access and handling* and 25% are *Processing*. Moreover, some are considered to be more critical than others. Severity and occurrence are two attributes of most bug models [63]. Severity refers to the severity of a resulting or potential failure on the behavior of the entire product, whereas occurrence describes the probability that a failure appears. A set of definitions for each of these two attributes was proposed by Johnson Controls software experts (see Table 1).

Severity	Occurrence (probability)
Secondary – cosmetic failure, not customer relevant	**Once** (< 1%) – low probability, unlikely failure
Minor – cosmetic failure, customer relevant	**Very Rare** (> 1% and < 5%) – low probability, few failures
Major – workaround exists	**Rare** (> 5% and < 10%) **–** moderate probability, occasional failures
Critical – no workaround exists	**Often** (> 10% and < 100%) – high probability, repeated failures
Catastrophic – system crash of the vehicle system (risk of person injury)	**Systematic** (= 100%) – failure unavoidable

Table 1 – Severity and occurrence attributes as defined by Johnson Controls software experts

According to these experts, despite these definitions, the attribution of a severity and occurrence for a bug detected internally remains a subjective question. Most test practitioners do not have a global view of the system that allows them to assess the impact of the detected bug on the end-user. However, the severity and occurrence of bugs detected by the carmakers are closer to reality since the carmaker is the one who sets the specifications. On the other hand, the carmaker tends to overstate the criticality of the bugs in order to have a faster response from the supplier. For the front wiper functionality, about 76% of the total bugs are (Minor, Systematic), and for the fuel gauge functionality, about 72% of the bugs are (Major, Systematic). These results could be explained by the fact that the functionality of managing the fuel level in a car is more critical than that of managing the wipers. As a consequence, bugs in the fuel gauge functionality are considered to be more critical than those in the front wiper functionality.

8.1.2.2 Testing time

In Figure 13, the time spent by the two different validation teams in order to test the software components (developed by two different development teams) of the two functionalities using conventional testing techniques is depicted. The main activities are:

- Analyze the carmaker requirements
- Design the test cases
- Execute the test cases and analyze the results
- Address the bugs detected internally (before delivery to the carmaker) and by the carmaker

50% (29.5 and 27 pd) and 10% (6.75 and 7 pd) of the total testing time were spent manually designing the test cases and managing the bugs detected by the carmaker, respectively. Using the current testing practices of the automotive industry, approximately 54 pd were spent testing the front wiper and 50 pd testing the fuel gauge (see Figure 13).

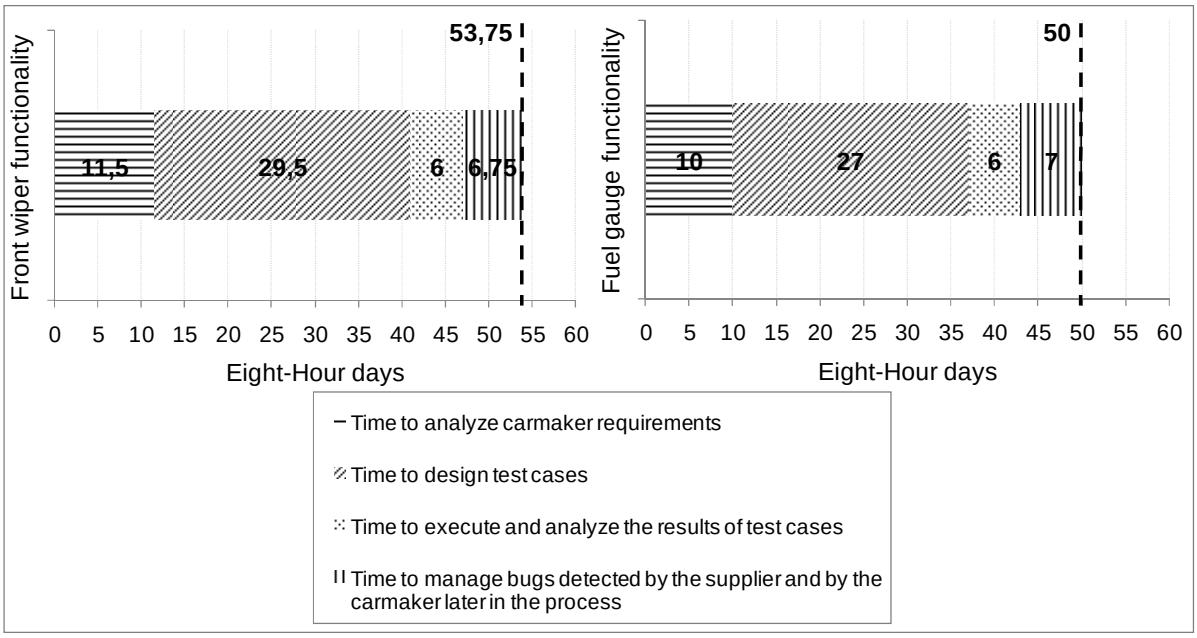


Figure 13 – Total time spent in testing the two functionalities – conventional approach (manual test design)

8.2 Experiment

For each of the two case studies, four stages were necessary to automatically generate test cases using the proposed MBST approach. Test cases were generated offline and later executed. The **first stage** consisted of designing automated test oracles (executable software specifications) for each of the software functionalities being tested. The **second stage** consisted of designing one or more transition matrices for each of the software functionalities being tested. The **third stage** focused on tuning the automatic generation of test cases. The **fourth and last stage** consisted of generating test cases and then executing them on the corresponding software components.

In the text that follows, the results of completing these four stages for each of the two functionalities described above are detailed. The team performing the experiment was composed of two individuals: an automotive test practitioner and an inexperienced engineer. During the experiment, advice from other Johnson Controls automotive experts was taken into account.

8.2.1 Stage 1: Design automated test oracles (executable software specifications)

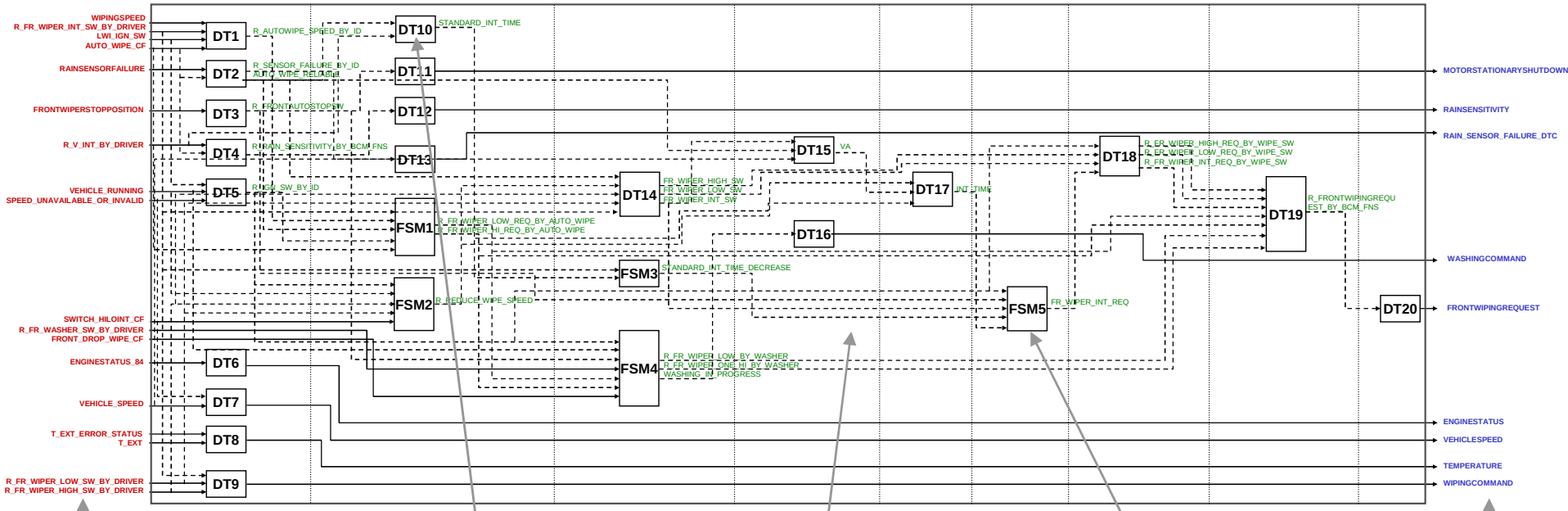
Four steps were necessary for designing an automated test oracle for each functionality. The first consisted of analyzing and understanding the software specifications. A loop process was initiated with software experts internal to Johnson Controls in order to understand and clarify the specification. The second step consisted of sketching the test oracle "on paper". The input, output, and intermediate signals and the features (Decision Tables and Finite State Machines) for each functionality were identified. Then, each feature was developed by identifying all the states, transitions, and conditions. The third step was the computerization and verification of the test oracle via a software routine specified in a technical report [31]. The final step was the validation of the test oracle. During the verification and validation steps, a total of 15 and 50 anomalies were found in the test oracle of the front wiper and fuel gauge functionalities, respectively. The time spent in completing each of these designing steps for the two functionalities was accounted for and is summarized in Table 2.

Time spent (pd)	Front wiper functionality	Fuel gauge functionality
Analyze the software specification	3	3
Sketch the test oracle "on paper" (manual verification)	5	7
Computerize the test oracle (automatic verification)	12	6
Validate the test oracle	5	20
TOTAL	**25**	**36**

Table 2 – Time spent in designing, verifying and validating the automated test oracles for the two functionalities

The verification and validation steps are very time consuming in having an expert evaluate the correctness against his domain knowledge. For the front wiper and fuel gauge functionalities, they accounted for 22 pd (5 + 12 + 5) and 33 pd (7 + 6 + 20), respectively.

The automated test oracle designed for the front wiper functionality is provided in Figure 14.



Input Signals	
AUTO_WIPE_RELIABLE	R_V_INT_BY_DRIVER
= 0.00	= 1.00
= 0.00	= 2.00
= 0.00	= 3.00
= 0.00	= 4.00

Output Signals
STANDARD_INT_TIME
= 8.00
= 4.00
= 1.00
= 14.00

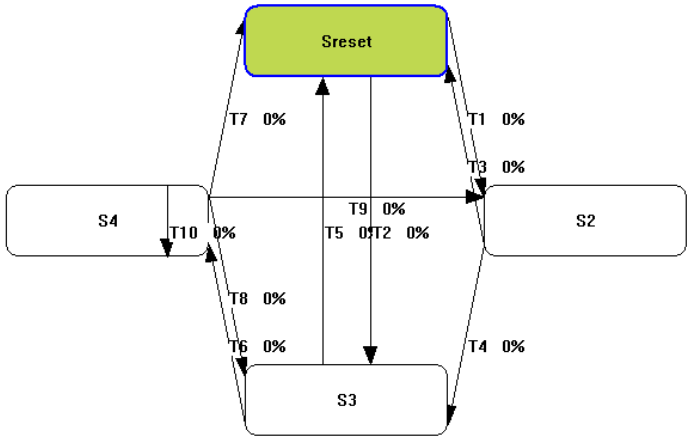


Figure 14 –Automated test oracle designed for the front wiper functionality (this figure

8.2.2 Stage 2: Design transition matrices

There were 9604 entries (98x98, where 98 is the number of possible functionality inputs) in a transition matrix for the front wiper functionality and 7921 (89x89) for the fuel gauge functionality. Entering them manually would have been intractable.

According to Johnson Controls software experts, setting assigning random test inputs (Nominal transition matrices) to the fuel gauge functionality does not make real sense. Thus, the two Nominal transition matrices for the front wiper functionality were automatically generated using a software routine specified in a technical report [31]. Based on assumptions from the Johnson Controls software experts, one standard time interval ([100; 400] was defined, the mean time interval between two operations carried out on an automotive electronics product, in milliseconds), and was applied to all sequential inputs. Based on these matrices, "quasi" random sequences of inputs were generated.

However, as stated in section 6.2.2.2, there is a need to test the input sequences recurrently executed by end-users. Therefore, a group of Johnson Controls software experts were asked to set some of the constraints developed in section 6.2.2.2 on the input signals of each of the two functionalities. Based on these constraints, an End-user Profile transition matrix was automatically generated for each functionality, using a software routine specified in a technical report [31].

A study on the bugs detected in the past on functionalities similar to the ones being tested in these case studies was also performed. The front wiper functionality had been developed in 4 different projects since 1997, in which a total of 55 bugs were detected. Unfortunately, the behavior and concept of the fuel gauge functionality had thoroughly changed in recent projects and it was therefore irrelevant to reuse stored bugs and test cases. One difficult task concerning the front wiper functionality was representing the "problem description" of the 55 identified bugs using the format illustrated in Figure 8. Based on advice from the Johnson Controls software experts, the 10 most critical bugs with enough information to formulate their "problem descriptions" were only considered. Afterwards, the 10 corresponding Bug transition matrices for the front wiper functionality were automatically generated using a software routine specified in a technical report [31].

Finally, the test cases already developed in the past for functionalities similar to the ones being tested were gathered. As stated before, test cases on the fuel gauge functionality could not be reused, and therefore the efforts were focused on the front wiper functionality. In the past, test practitioners have designed many test cases (about 2000 test steps) in order to test this functionality. Using these test cases, one Test Case transition matrix for the front wiper functionality was automatically generated using a software routine specified in a technical report [31].

A summary of the number of "transition matrices" designed for the two functionalities is presented in Table 3. The time spent in designing these transition matrices was measured and is also summarized in Table 3. Just identifying and preparing the stored bugs and test cases took about 1.5 pd.

		Front wiper functionality	**Fuel gauge functionality**
# of the designed transition matrices	Nominal	2	0
	End-user Profile	1	1
	Bug	10	0
	Test Case	1	0
Time spent in designing these transitions matrices		2 pd	0.5 pd

Table 3 – Transition matrices designed for the two functionalities

An illustration of two Nominal transition matrices is provided in Figure 15. These are the Nominal 1 and 2 transition matrices of the example illustrated in Figure 6. Illustrating the transition matrices of the front wiper or fuel gauge functionalities could be illegible due their thousands of entries.

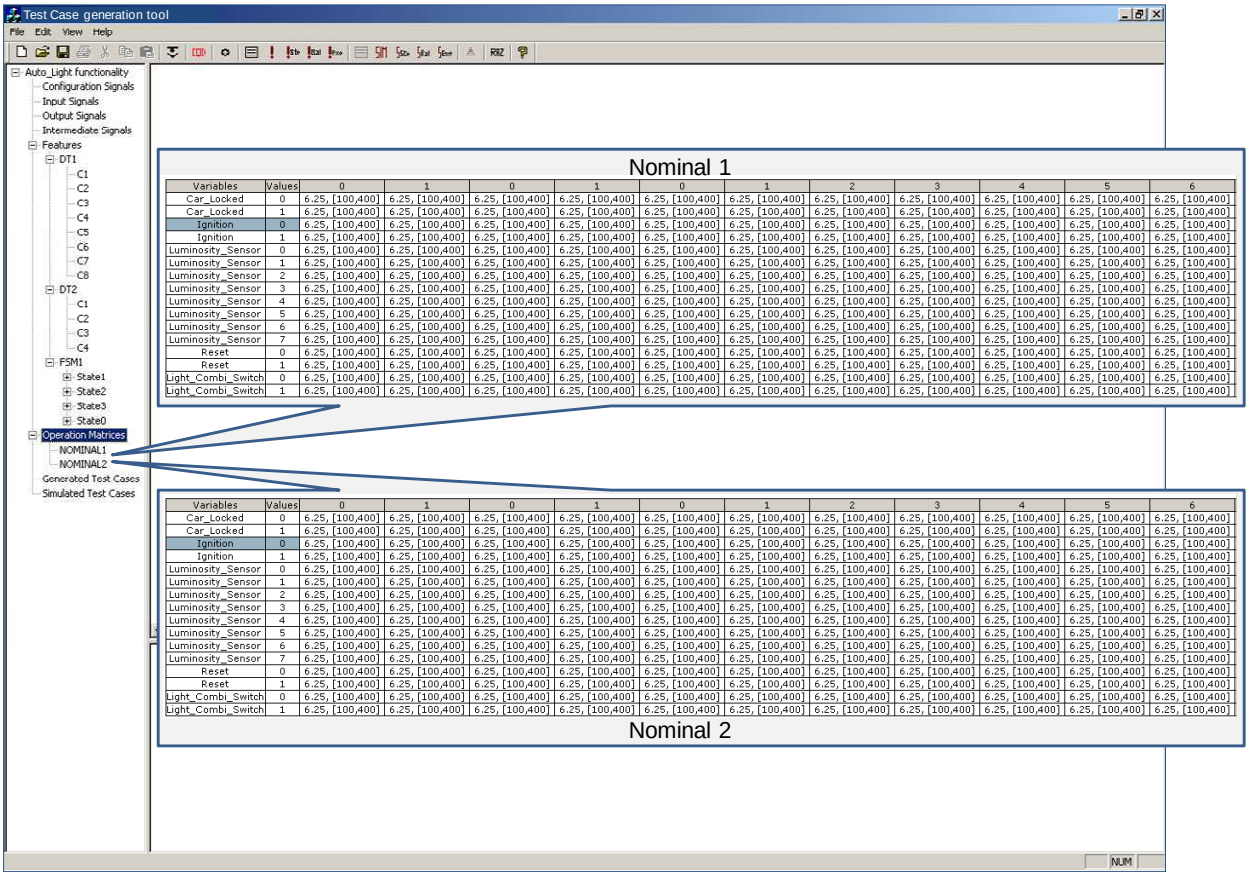


Figure 15 – An illustration of two Nominal transition matrices

8.2.3 <u>Stage 3:</u> Tune the generation of test cases

Three questions were raised at this stage of the experiment:

1. From which transition matrix do we start generating test cases?

It was planned to generate test cases from the transition matrices in the following order: first, from the Bug transition matrices in order to ensure that the software is free from bugs similar to the ones already detected in the past; second from the Test Case transition matrices, which are suitable for bug detection, since they are based on a test practitioner's experience; third, from the End-user Profile transition matrix, which aims to check that the software fulfills the end-user (driver) expectations; and finally, from the Nominal transition matrices, in which improbable successions of test inputs are generated in order to check the robustness of the software. These test generation principles were discussed by Frankl et al. [64]. The authors highlight the two main goals in testing software: 1) to achieve adequate quality by detecting the maximum number of bugs possible (debug testing: Bug, Test Case, and Nominal transition matrices), 2) to assess existing quality and increase confidence in the software reliability (operational testing: End-user Profile transition matrix).

In section 8.2.2, it is noted that only End-user Profile "transition matrix" was designed for the second case study. Therefore, for this case study, test cases are only generated from the End-user Profile transition matrix.

2. How do we tune the test coverage objectives and the cost constraints?

Before generating test cases from a transition matrix, the objectives and constraints shall be defined. According to the type of transition matrix, guidelines for defining the test coverage objectives and the cost constraints were proposed (see Table 4). The problem of when exactly to stop testing depends on the adequacy of the transition matrix, coverage objectives and constraints.

Type of transition matrix	Objectives guidelines	Constraints guidelines
Bug and Test Case	100% coverage of the transition matrix	The number of test steps and the execution time of the generated test case depend on the context (budget, planning, resources) of the project
End-user Profile	100% coverage of the input signals domains	
Nominal	100% coverage of the transition matrix and of the whole test oracle	

Table 4 – Guidelines for defining the objectives and constraints of test case generation

For instance and in case of generating test cases from an End-user Profile transition matrix (see Figure 16), an objective of 100% coverage of the input signals domains should be set. The constraints should be set based on the project context: a budget of 1 hour (3600000 ms) of manual test execution has been scheduled.

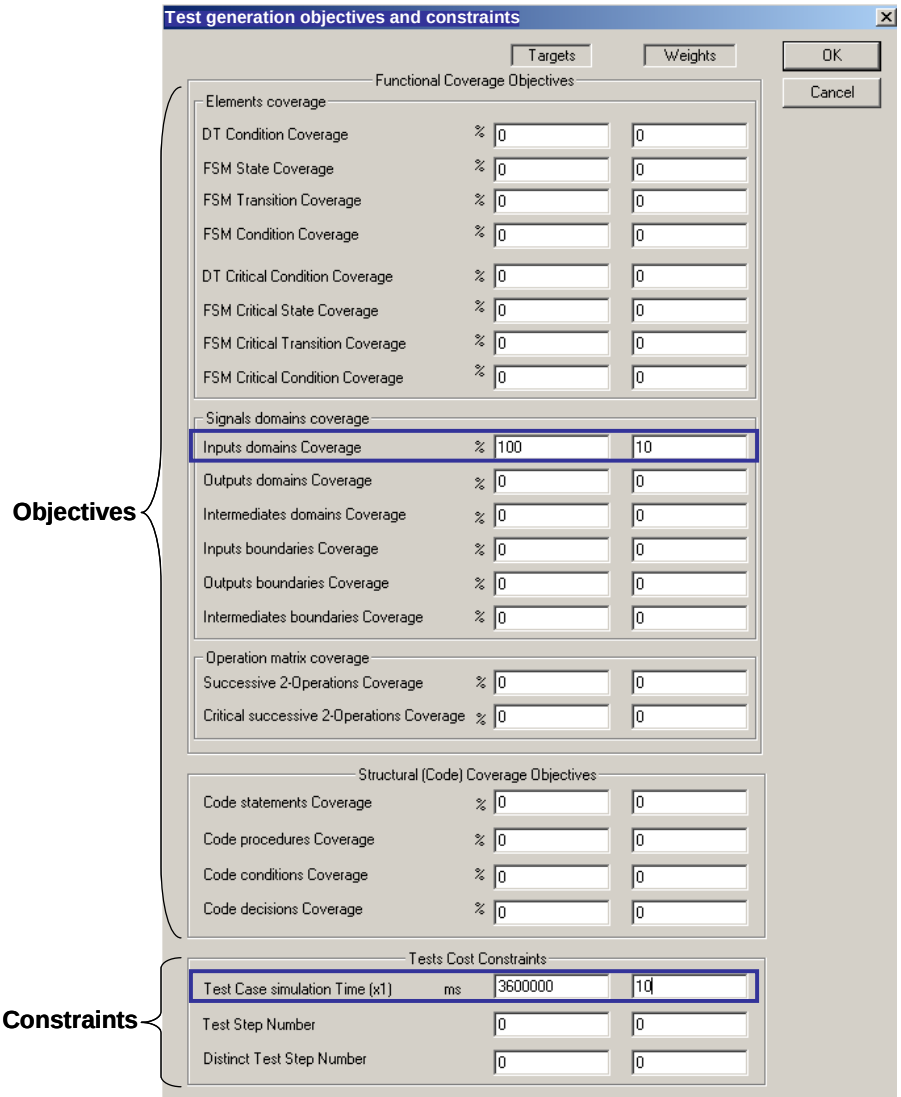


Figure 16 – An illustration of the objectives and constraints when generating test cases from an End-user Profile transition matrix

Since test cases are generated offline and later executed, objectives are set only in terms of functional coverage. The same weight *w* (1, 5 or 10) were also considered for all the coverage goals. The objective function of section 7.2, $F_{Objective}$, is therefore defined as:

$$F_{Objective} = \sum_{j} \left| FuncCovTarget_j - FuncCovCurrent_j \right| \times w$$

Finally and since test cases are automatically executed in a simulated environment (host PC), no constraints were set in terms of number of test steps or execution time of the generated test cases. The constraint function of section 7.2, $F_{Constraint}$, is therefore defined as:

$$F_{Constraint} = 0$$

3. How do we tune the parameters of the optimization algorithm?

After defining objectives and constraints, the optimization algorithm of the test case generation was tuned. In this paper, these parameters were tuned based on the traditional try-and-test protocol. The purpose is to better fulfill and respect the test coverage objectives and the cost constraints. 1 pd was spent in adjusting these parameters for the two case studies.

8.2.4 Stage 4: Generate and execute the test cases

The generation of test cases was carried out automatically and offline using a software routine specified in a technical report [31].

For the front wiper functionality, the following list of test cases was generated:

- 10 test cases (one test case from each Bug transition matrix). Each test case is about 10 test steps. For each test case, objectives were fulfilled at 100%.
- 6 test cases from the Test Case transition matrix. Each test case was about 400 test steps and none of them fulfills at 100% the test objectives. After combining the test coverage of these test cases, objectives were fulfilled at 99%.
- 6 test cases from the End-user Profile transition matrix. Each test case was about 1000 test steps and none of them fulfills at 100% the test objectives. After combining the test coverage of these test cases, objectives were fulfilled at 70%.
- 6 test cases from the Nominal 2 transition matrix. Each test case was about 10000 test steps and none of them fulfills at 100% the test objectives. After combining the test coverage of these test cases, objectives were fulfilled at 90%.

For the fuel gauge functionality, only 6 test cases were generated from the End-user Profile transition matrix. Each test case was about 300 test steps and none of them fulfills at 100% the test objectives. After combining the test coverage of these test cases, objectives were fulfilled at 90%.

An extract from a test case generated for the front wiper functionality is provided in Figure 17.

Configuration signals	
AUTO_WIPE_CF	0
FRONT_DROP_WIPE_CF	1
SWITCH_HILOINT_CF	1
FR_WIPER_PROLONGATION_WIPES_TP	4

		Input signals															Output signals								
Test Step n°	Wait (ms)	T_EXT_ERROR_STATUS	ENGINESTATUS_84	T_EXT	R_FR_WASHER_SW_BY_DRIVER	WIPINGSPEED	SPEED_UNAVAILABLE_OR_INVALID	VEHICLE_SPEED	RAINSENSORFAILURE	R_V_INT_BY_DRIVER	R_FR_WIPER_INT_SW_BY_DRIVER	LWI_IGN_SW	FRONTWIPERSTOPPOSITION	R_FR_WIPER_HIGH_SW_BY_DRIVER	VEHICLE_RUNNING	R_FR_WIPER_LOW_SW_BY_DRIVER	RAIN_SENSOR_FAILURE_DTC	TEMPERATURE	WASHINGCOMMAND	WIPINGCOMMAND	FRONTWIPINGREQUEST	VEHICLESPEED	RAINSENSITIVITY	MOTORSTATIONARYSHUTDOWN	ENGINESTATUS
1	390	0	0	-5	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
2	280	0	0	-5	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0
3	330	0	0	-5	0	0	0	80	0	1	1	0	1	0	0	0	0	0	0	1	0	16	0	1	0
4	200	0	0	-5	0	0	0	80	0	1	1	0	1	0	0	0	0	0	0	1	0	16	0	1	0
5	130	0	0	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	0	0	1	0	16	0	1	0
6	330	0	0	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	0	0	1	0	16	0	1	0
7	240	0	0	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	0	0	1	0	16	0	1	0
8	290	0	0	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	0	0	1	0	16	0	1	0
9	250	0	2	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	0	0	1	0	16	0	1	2
10	270	1	2	-5	0	0	0	80	0	1	1	0	1	1	0	0	0	31	0	1	0	16	0	1	2
11	260	1	2	-5	0	0	0	80	0	1	1	0	1	1	1	0	0	31	0	1	7	16	0	1	2
12	240	1	2	-5	0	0	0	80	1	1	1	0	1	1	1	0	1	31	0	1	7	16	0	1	2
13	350	1	2	-5	0	0	0	80	1	1	1	0	1	1	1	0	1	31	0	1	7	16	0	1	2
14	180	1	2	-5	0	0	0	80	1	1	1	0	1	0	1	0	1	31	0	1	6	16	0	1	2
15	220	1	3	-5	0	0	0	80	1	1	1	0	1	0	1	0	1	31	0	1	6	16	0	1	3

Figure 17 – An extract from a test case generated for the front wiper functionality (this figure is voluntarily fuzzyfied for confidentiality reasons)

One test case was generated based on each Bug transition matrix. A test case of about 10 test steps was enough to fulfill the objectives and constraints defined in Table 4 in the case of a Bug transition matrix. For each Test Case, End-user Profile, and Nominal transition matrix, more than one test case were generated since it was very difficult to generate one test case that fulfills the defined objectives 100%. The length of a test case (number of test steps) depends on the level of difficulty in reaching the defined objectives. Even with a test case with thousands of test steps (Cf. Nominal 2 transition matrix), it was difficult to fulfill most of the objectives using a test generation algorithm based on a Monte Carlo simulation on the transition matrix. As a consequence, 6 test cases were generated from each transition matrix (with the same objectives and constraints). This ensured the repeatability of the results in terms of objective fulfillment since the 6 test cases reached the predefined objectives with a small standard deviation of 10%. In future works (see section 9), it is planned to develop a new and complementary test generation algorithm that focuses on fulfilling the test objectives (i.e. covering non-covered zones of the software specification).

The generated test cases were executed on the first version of the two software components corresponding to the two software functionalities being tested. All the generated test cases were feasible (i.e. executable). The test cases were automatically transcribed into a unit test language (computer-readable) by using a Visual Basic macro [2] and then automatically executed on a unit test execution platform. In fact, all the dependencies and connections between the software components are simulated on computer in order to isolate the tested component from the whole product. The abstract model of the unit test execution platform is illustrated in Figure 18.

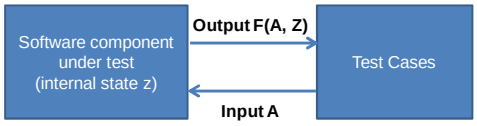


Figure 18 – Abstract model of the unit test execution platform [2]

The unit test uses the inputs and outputs of the software component under test. Test cases should know expected output F when input A is applied. The presently produced output has to be compared with the expectation. If they do not match, an error should be generated in the test report.

It is important to note that the time to generate and execute test cases was trivial from an automotive industry point of view. It is estimated to be 500 test steps per minute. This estimation is given for reference only because it depends on many factors (CPU[1], transition times of the test steps, parameters of the optimization algorithm, and so forth).

Each time an anomaly was detected, it was analyzed in order to identify its origin among the following possibilities:

- A bug in the test oracle.
- A known bug in the software component, which the testing phases of the supplier or the carmaker had already detected (respectively ◊ and □ in Figure 20).
- An unknown bug in the software component, which had not yet been detected by the testing phases of the supplier or by the carmaker (Δ in Figure 20).

Two instances of test execution were performed in parallel:

1. The first followed the test generation order reasoned in section 8.2.2. Whatever the origin of the detected anomalies, the bug was corrected before continuing the execution of the remaining test cases.
2. The second did not follow any predefined order of transition matrices; before executing the test cases associated with each transition matrix, the first version of the software components being tested was considered. When executing the test cases of a transition matrix, whatever the origin of the detected anomalies, the bug was corrected before restarting the execution of the remaining test cases of the same transition matrix. This highlights the need of each type of transition matrix.

The correction of anomalies was instantaneous and assumed to be perfect. Since the test generation and execution were automated, they did not require any human intervention (0 pd). Ten pd were spent in analyzing the execution results of the first case study, and two pd were spent in doing the same for the second case study. This time is proportional to the number of executed test steps (front wiper: 68500 test steps, fuel gauge: 900 test steps).

In section 8.3, the experimental results (in terms of bug detection and time spent on testing activities) of the two instances of test execution are analyzed and discussed. They are compared to results obtained with the conventional approach (see section 8.1.2).

8.3 Analysis of experimental results

8.3.1 Increase the number of bugs detected earlier in the software life cycle

On the one hand, all the generated test cases (in the order defined in section 8.2.2) were executed on the first version of the software components of the two functionalities. A total of 29 anomalies were detected in the first case study and 35 anomalies in the second one. About 17% (5 out of 29) of the anomalies detected in the front wiper functionality were related to bugs in the test oracle, as were about 49% (17 out of 35) of the anomalies detected in the fuel gauge functionality. This may be explained by the fact that the test oracle could not be exhaustively validated, especially the case where the carmaker requirements were expressed in informal language. An in-depth analysis of the remaining anomalies ((29-5) and (35-17)) leads to the following three main conclusions (see Figure 19):

1. Firstly, 86% (19 out of 22) of the bugs already detected by the conventional testing phases in the first case study were detected. 78% (18 out of 23) in the second one. The

[1] Central Processing Unit

remaining 8 bugs ((22-19)+(23-18)) detected by the conventional testing phases and not by the proposed approach were classified by Johnson Controls software experts according to their typology, severity and occurrence: 2 of them were (Control flow and sequencing, Minor; Systematic), 1 (*Data definition, access and handling*, Minor; Systematic), 1 (*Processing*, Major; Often), 1 (Processing, Major; Systematic), 2 (Control flow and sequencing, Major; Systematic) and 1 (*Data definition, access and handling*, Major; Systematic). These bugs are located in non-covered zones of the software specification. All of these bugs could be detected by the proposed approach if the coverage objectives defined in Table 4 are fulfilled at 100% (which was not the case in this experiment). These non-detected bugs are related to specific states, transitions, and conditions of the software specification that were not covered by the generated test cases; when test cases were generated from a Nominal transition matrix, the test generation algorithm based on a Monte Carlo simulation on the transition matrix did not succeed in reaching 100% functional coverage, but reached only 90%. To overcome this shortcoming (see section 9), it is planned to develop a new and complementary test generation algorithm that focuses on covering the non-covered zones of the software specification.

2. Secondly, 5 new “minor” bugs (“minor” from the Johnson Controls software experts’ point of view) were detected in the front wiper functionality. They were not detected neither by the conventional testing phases of the supplier nor by the carmaker test. According to these experts, these bugs have no impact on the end-user (driver). It represents 19% (5 out of (19+3+5)) of the total number of bugs in the functionality (19+3+5).
3. Finally, among the known bugs detected by the proposed approach, some of them were bugs already detected by the conventional supplier testing phases (but later in the testing process) and the others by the carmaker. For the front wiper, the number of bugs detected earlier by the supplier was increased by 41% (from 17 to 24). For the front wiper and fuel gauge functionalities, the number of bugs detected by the carmaker was reduced by 60% (from 5 to 2) and 80% (from 5 to 1), respectively.

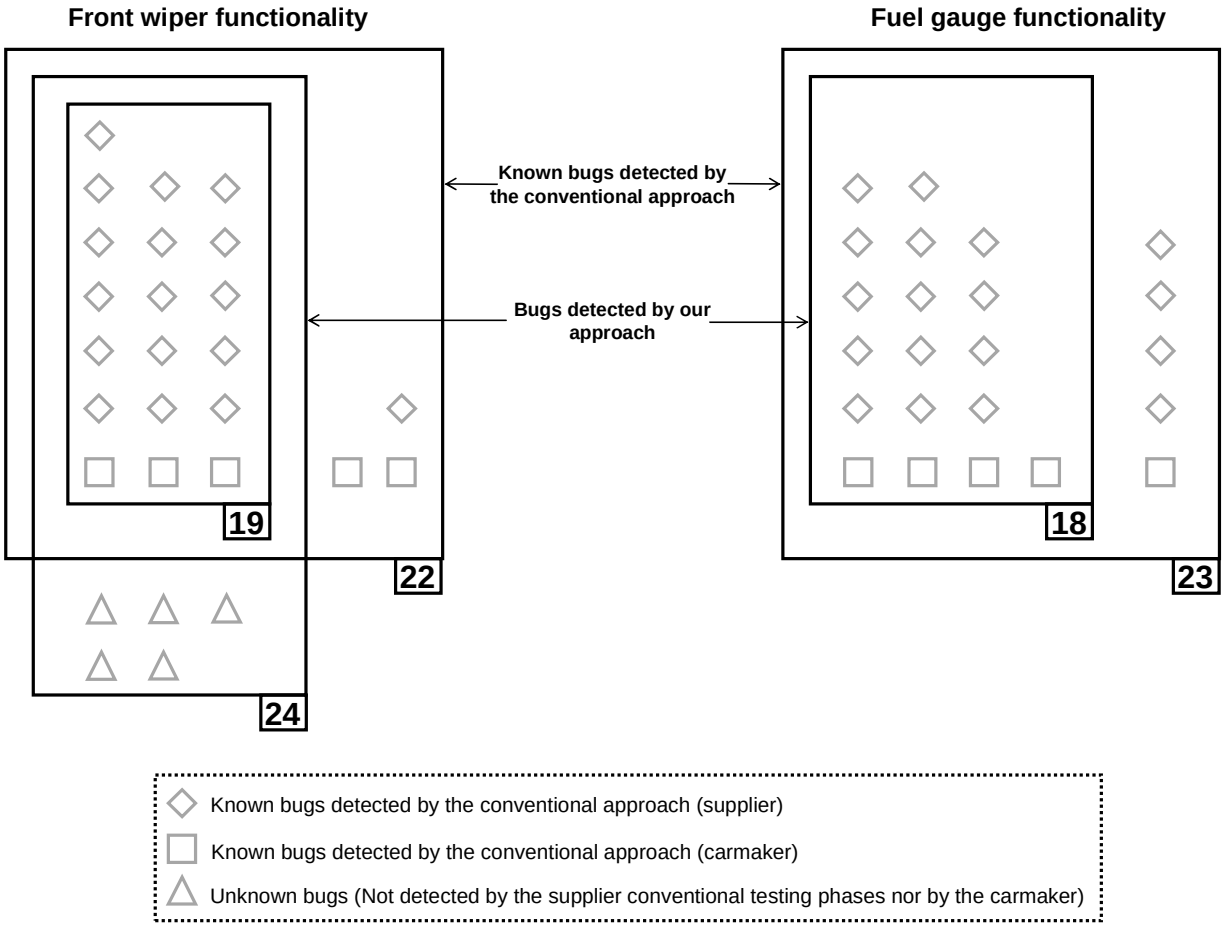


Figure 19 – Conventional approach versus proposed approach

On the other hand, the test cases generated from each type of transition matrix were independently executed (without following the order defined in section 8.2.2). The results of this experiment on the front wiper functionality are depicted in Figure 20. The numbers and types of bugs detected in and after the first testing phase by each type of transition matrix were identified. As a conclusion:

- No single type of transition matrix was able to detect all the bugs, and each type of transition matrix found at least one bug that could only be detected via this type. This asserts the dynamic nature of software and its consequential need for more than one transition matrix (different transition probabilities and time intervals).
- The Nominal transition matrix detected the largest number of bugs, but not all of the bugs. The fact that it detected the largest number of bugs may be explained by the fact that 60000 test steps were generated from this transition matrix and that the software specification was covered at 90%. However, the fact that not all the bugs were detected confirms the relevance of the proposed approach for combining diverse testing techniques (random, user-oriented, and fault-oriented).
- The Test Case transition matrix detected about 80% of the bugs that the End-user Profile matrix detected. As Test Case “transition matrices” transition matrices are designed from reused test cases, it is possible that the reused test cases were designed from an end-user point of view.

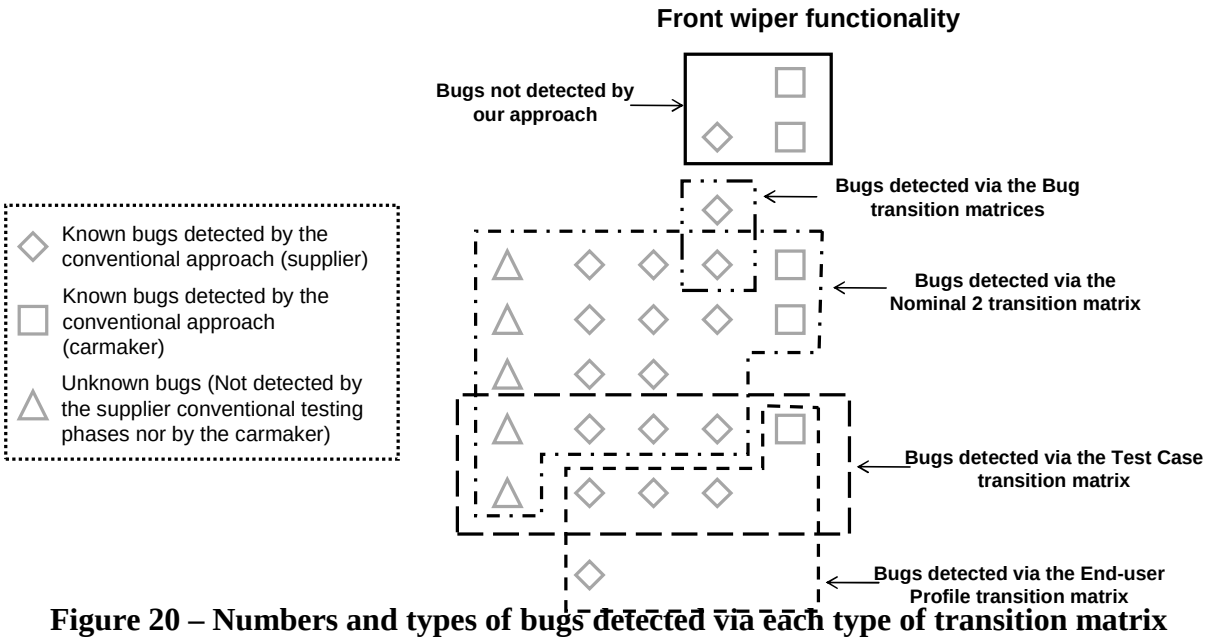


Figure 20 – Numbers and types of bugs detected via each type of transition matrix

8.3.2 Decrease the time spent in testing software

Besides detecting bugs earlier in the software development process, the time spent in testing software was decreased. The total time spent in conventionally testing the two functionalities is illustrated in Figure 13, based on historical data: 53.75 pd for the front wiper functionality and 50 pd for the fuel gauge functionality. The total time spent in testing these functionalities using the proposed approach is presented in Figure 21. The time spent in testing the front wiper and fuel gauge functionalities was decreased by 27% (39 instead of 53.75 pd) and 17% (41.5 instead of 50 pd), respectively. These numbers account for the time spent in analyzing the carmaker requirements; designing, verifying, and validating the test oracle; designing the transition matrices; generating and executing the test cases; and finally, detecting the bugs not detected by the proposed approach. In this experiment, there were 3 known bugs left undetected in the first case study and 5 in the second case study. Based on the assumption that a complementary test generation algorithm would be developed (to be able to reach 100% functional coverage), the time required to detect these remaining bugs in the two case studies was estimated, taking into account the time to generate and execute the test cases in a simulated environment (host PC) and analyze the results. For the first case study, 90% of the software specification had already been covered, and 3 bugs were remaining. For the second case study, 70% of the software specification had already been covered, and 5 bugs were remaining. Therefore, based on the experimental results, it was estimated that it would require 2 pd to generate and execute additional test cases and then 3 pd to analyze the test execution results. These estimations may be explained by the fact that:

- The software specification of the first case study is bigger than that of the second case study.
- Analyzing the execution results of the second case study takes more time that of the first case study, because the test oracle of the second case study (natural language) is less reliable that of the first case study.

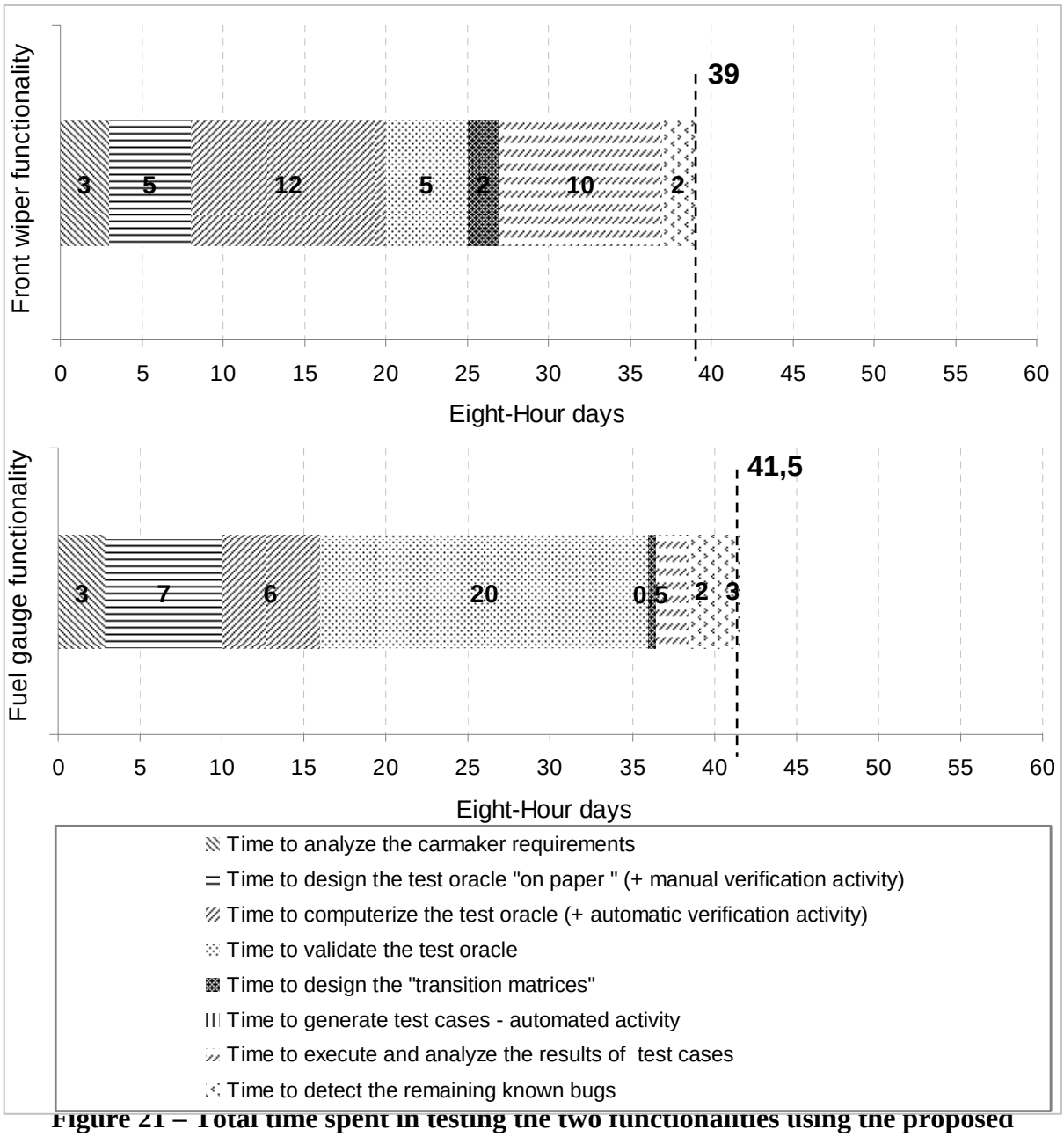


Figure 21 – Total time spent in testing the two functionalities using the proposed approach

The task of manually designing the test cases is replaced by designing, verifying, and validating the test oracle, which is also considered to be a difficult task. For the front wiper and fuel gauge functionalities, the verification and validation tasks accounted for 57% (22 out of 39 pd) and 80% (33 out of 41,5 pd) of the total time spent in testing the two functionalities using the proposed approach, respectively. Moreover, as a consequence of automatically generating many test steps, more effort is necessary to analyze the results of the test case execution. Test practitioners have to understand the generated test cases in order to confirm the existence of a bug. However, as carmaker requirements are prone to evolving throughout the timeline of the different deliveries, it will be easier for test practitioners to update the test oracle and automatically generate a new set of test cases than to manually update the design of test cases.

8.3.3 Threats to validity

The main validity threats to the perfomed experiment are related to the possible non-representativity of the selected software functionalities, inaccuracy of historical data,

inadequate experience level of the team performing the experiment, non-correlation between functional and structural coverage.

- Non-representativity of the selected software functionalities

In section 8.1.1, the selection of the two software functionalities for the case studies is discussed and justified. One important criterion in choosing the functionalities was that they exemplify the diversity of carmaker software specification languages. However, it is possible to miss a relevant software specification language used by the carmakers. One such instance is the existence of many specification languages or the creation of a new one. This can have an impact on the feasibility and, if feasible, on the performance (time to design, verify and validate the test oracle) of the proposed approach.

- Inaccuracy of historical data

Inaccurate historical data can be the result of subjective and unsystematic data extraction. In this research, all the extracted data were reviewed by Johnson Controls automotive experts. All discrepancies were settled by discussion to make sure that the extraction was as objective as possible. Therefore, the remaining problem is the validity of the experts' knowledge of these historical data. The experts were chosen based on their knowledge of the software functionalities under test. Hence, this could have an impact on the estimated benefits of the proposed approach.

- Inadequate experience level of the team performing the experiment

The team performing the experiment was composed of two individuals: an automotive test practitioner who knows the conventional approach and an inexperienced engineer who is freshly graduated. During the experiment, advice from other Johnson Controls automotive experts was also taken into account. The justification for the validity of this team is the representativeness of a testing team within the automotive industry. Indeed, this is the minimum requirement for having a valid empirical study in the domain of automotive software testing. As stated in the future work (section 9), it is planned to measure the reproducibility of the results of the two case studies by choosing another team.

- Non-correlation between functional and structural coverage

In the experiment, test cases were generated offline and later executed. Therefore, objectives were only set in terms of functional coverage. No structural coverage objectives were set. This has been done assuming that covering the software specification 100% means that the source code was also 100% covered. This assumption can have a negative impact, in terms of bugs' detection, on the estimated benefits of the proposed approach.

9 Summary and perspectives

In this paper, an integrated model-based statistical approach to automatically generate functional test cases for embedded software is developed. Test cases can be generated offline and later executed, or they can be generated and executed online. The purposes of the proposed approach are 1) ensuring conformance to specification, 2) ensuring code coverage and 3) avoiding recurrent bugs. The basics of this approach are:

- A probabilistic test model based on Markov Chains (transition matrix). When testing software, test practitioners can design one or more transition matrices that enable random, user-oriented, or experience feedback-oriented generation of test inputs.
- A formal framework integrating existing and appropriate specification techniques (DT and FSM). This allows the design of executable software specifications that play the role of the test oracle in assessing the expected results of a test.
- An aggregate stop testing criterion based on test coverage objectives and cost constraints.

In other words, an integrated framework to automatically generate test cases (test inputs and expected results) from any software specification was developed. This framework focuses on

important and critical tests to be done. The test generation is automated and monitored by quality and cost objectives, in that test practitioners can generate one or more test cases that fulfill a set of objectives (in terms of functional coverage, structural coverage, and test cost).

Two typical case studies on historical data from the automotive industry were also carried out. The experiments were performed at the unit testing level in a simulated environment on a host PC (automatic test execution). Test cases were generated offline and later executed. Potential advantages of the proposed approach over the conventional approach were highlighted. In Table 5, the results of the two case studies is summarized in terms of decreasing the time spent in testing and detecting bugs earlier in the software life cycle (during and after the first testing phase of the first version of the software components).

	Front wiper functionality	**Fuel gauge functionality**
Decreasing the time spent in testing	-27% (39 instead of 53.75 pd)	-17% (41.5 instead of 50 pd)
Increasing the number of bugs detected since the first testing phase	+88% (24 out of 27)	+78% (18 out of 23)
Decreasing the number of bugs detected by the carmaker	-60% (from 5 to 2)	-80% (from 5 to 1)
Increasing the number of bugs detected by the supplier	+41% (from 17 to 24)	+22% (from 18 to 22)
Increasing the number of new bugs detected	+18% (5 out of 27)	+0% (0 out of 23)

Table 5 – A summary of the results of the two case studies

Here are some perspectives of our research:

- In addition to the selection of inputs via a Monte Carlo simulation on the transition matrix, it would be interesting to develop a new and complementary test generation algorithm that focuses on covering non-covered areas of a software specification. This will allow the deduction and creation of inputs that cover a specific area (for instance, a state of an FSM, a condition of a DT, etc.) of the test oracle with a minimum number of test steps. This algorithm would not replace the one based on the Monte Carlo simulation method. It will be used in case the Monte Carlo simulation method does not succeed to fulfill target coverages while taking into account the test constraints (number of test steps). Similar algorithms have already been developed in the past but not integrated into a global test generation approach; especially in model checkers, SMT solvers, and constraint solvers.
- It would be interesting to evaluate the capabilities of the proposed test oracle's verification and validation methods against detecting the anomalies of a test oracle. For both the front wiper and fuel gauge functionalities, 25% (5 out of (15 + 5) and 17 out of (50 + 17), respectively) of the test oracle anomalies were not detected by these methods. It is furthermore interesting if a corresponding fault model could be derived from these results.
- It would be interesting to develop a new strategy to help test practitioners parameterize the generation of test cases, as the main purpose of a test practitioner in software testing is to detect the maximum number of bugs possible in the minimum amount of time possible. Therefore, the correlations between the optimization algorithm parameters, the functional coverage, the execution time of the generated test cases, and the numbers and types of detected bugs could be identified. Based on these correlations, rules and recommendations to help test practitioners parameterize the generation of test cases might be defined. It is also planned to develop parameterization profiles that test practitioners might adopt according to their test

objectives. Such a parameterization profile will consist of a set of predefined optimization parameters, test coverage objectives, and cost constraints.

- It would be interesting to adjust the proposed approach to integration and validation levels (all the functionalities together). In this research, the proposed approach was experimented at unit testing level (a single functionality at a time). Indeed and when integrating and validating the whole software product, all of the test oracles of the single functionalities could be connected together and transition matrices could take all the input signals of the product into account. In that case, integration and validation test cases could be automatically generated and the multilevel aspect of the proposed approach confirmed.
- It would be interesting to develop the interface between the proposed approach and the unit test execution platform on which the generated test cases of the experiments have been executed. Therefore, each generated test step will be automatically transcribed into a computer-readable language and then automatically executed on the software under test. After each test step, the test generation platform acquires the structural coverage of the software under test. In that case, experimenting the online generation and execution of test cases on both case studies would be possible.
- Finally, it would be interesting to measure the reproducibility of the results of the two case studies, in order to better control practices in industrial software testing processes. In the proposed MBST approach, the two main activities depend on the operator (i.e., human intervention). The first one is the design of the test oracle and the second is the definition of a set of targets and weights for the test case generation. Two operators may perform slightly differently and have slightly different results; parallel experiments of test design can therefore be conducted based on the same carmaker requirements. Best practices may consequently be derived in order to reduce subjective areas of the modeling activity.

10 References

[1] Sangiovanni-Vincentelli A. Electronic-System Design in the Automobile Industry. *IEEE Micro* 2003; 23 (3): 8-18.

[2] Awedikian R. Quality of the design of test cases for automotive software: design platform and testing process. *PhD Disseration*, Ecole Centrale Paris, February 2009. http://tel.archives-ouvertes.fr/tel-00393847/en/ [June 6, 2012].

[3] Beizer B. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[4] Seroussi G, Bshouty NH. Vector sets for exhaustive testing of digital circuits, *IEEE Transactions on Information Theory* 1998; 34(3): 513-522.

[5] Yang O, Jenny Li J, Weiss D. A Survey of Coverage Based Testing Tools. *International workshop on Automation of Software Test*, Shanghai, China, 2006, 99-103.

[6] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing: final report. *Planning report 02-3* 2002.

[7] McDonald M, Musson R, Smith R. The Practical Guide to Defect Prevention. *Microsoft Press* 2007, 480.

[8] Agrawal K, Whittaker JA. Experiences in applying statistical testing to a real-time, embedded software system. *Proceedings of the Pacific Northwest Software Quality Conference* 1993.

[9] Bauer T, Bohr F, Landmann D, Beletski T, Eschbach R, Poore J. From Requirements to Statistical Testing of Embedded Systems. *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems* 2007, 4.

[10] Bernard E, Legeard B, Luck X, Peureux F. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *International Journal of Software Practice and Experience* 2004; 34(10): 915-948.

[11] Rosaria S, Robinson H. Applying models in your testing process. *Information and Software Technology* 2000; 42(12): 815-824.

[12] Avritzer A, Larson B. Load testing software using deterministic state testing. *Proceedings of the International Symposium on Software Testing and Analysis* 1993, Cambridge, MA, USA, 82-88.

[13] Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM. Model-based testing of a highly programmable system. *Proceedings of the 1998 International Symposium on Software Reliability Engineering* 1998, 174-178, *Computer Society Press*, November 1998.

[14] Siegl S, Hielscher K.S, German R, Berger C. Formal specification and systematic model-driven testing of embedded automotive systems. *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* 2011, 1-6, Genoble, France.

[15] Sarode S, Radhakrishnan S, Sampath V, Jiang Z, Pajic M, Mangharam R. Demo Abstract: Model-Based Testing of Implantable Cardiac Devices. *Proceedings of the Third International Conference on Cyber-Physical Systems* 2012, 221, Beijing, China.

[16] Carter JM, Poore J.H. Sequence-based specification of feedback control systems in Simulink. *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research* 2007, Ontario, Canada, 332-345, Richmond Hill.

[17] Bohr F. Model Based Statistical Testing of Embedded Systems. *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops* 2012, 18-25, Berlin, Germany.

[18] Ozekici S, Altinel IK, Angun E. A general software testing model involving operational profiles. *Probability in the Engineering and Informational Sciences* 2001; 15(4): 519-533.

[19] Wohlin C, Runeson P. Certification of software components, *IEEE Transactions on Software Engineering* 1994; 20(6): 494-499.

[20] Poore JH, Whittaker JA. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology* 1993; 2(1): 93-106.

[21] Whittaker JA, Thomason MG. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering* 1994; 20(10): 812-824.

[22] Hessel A, Larsen KG, Mikucionis M, Nielsen B, Pettersson P, Skou A. Testing real-time systems using UPPAAL. *Formal methods and testing: an outcome of the FORTEST network*, Springer-Verlag, Berlin, Heidelberg, 2008.

[23] Legeard B, Peureux F. Generating of functional test sequences from B formal specifications presention and industrial case study. *Proceedings of the 16th IEEE international conference on automated software engineering* 2001, San Diego, USA 377-381.

[24] Hartman A, Nagin K. The agedis tools for model based testing. *ACM SIGSOFT Software Engineering Notes* 2004; 29(4): 129-132

[25] Jard C, Jéron T. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer* 2005; 7(4): 297-315

[26] Farchi E., Hartman A. and Pinter S.S. Using a model-based test generator to test for standard conformance. *IBM Systems Journal* 2002, 41 (1): 89-110.

[27] Lugato D, Bigot C, Valot Y, Gallois JP, Gérard S, Terrier F. Validation and automatic test generation on UML models: the AGATHA approach. *International Journal on Software Tools for Technology Transfer* 2004; 5(2): 124-139.

[28] Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. *Journal of Software Testing, Verification, and Reliability* 2002; 12(4): 219-249.

[29] Jeffrey D, Gupta N. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *IEEE Transactions on Software Engineering* 2007; 33(2): 108-123.

[30] Bertolino A. Software Testing Research and Practice. *Invited presentation at 10th International Workshop on Abstract State Machines ASM* 2003, Taormina, Italy, 1-21.

[31] Awedikian R, Yannou B. Automatic generation of relevant test cases: A practical model-based statistical testing approach - Part 1: Models and concepts - Part 2: Prototype implementation - Part 3: Experiments. *Technical report* 2009, Ecole Centrale Paris, France. http://www.lgi.ecp.fr/uploads/PagesPerso [June 6, 2012].

[32] Weyuker EJ, Ostrand TJ. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering* 1980; 6(3): 236-246.

[33] Nardi PA, Delamaro ME. Test oracles associated with dynamic systems models. *Technical report* 2011, Instituto de Ciencias Matematicas e de Computaçao, Universdade de Sao Paulo, Brazil. http://www.icmc.usp.br/~biblio/BIBLIOTECA/rel_tec/RT_362.pdf [June 6, 2012].

[34] Hamlet RG. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* 1997; 3(4): 279-290.

[35] Chapman D. A Program Testing Assistant. *Communications of the ACM* 1982; 25(9): 625-634.

[36] Weyuker EJ. On testing non-testable programs. *Compuer Journal* 1982; 25(4): 465-470.

[37] Baresi L, Young M. Test oracles. *Technical report* 2001, Department of Computer and Information Science, University of Oregon. http://ix.cs.uoregon.edu/~michal/pubs/oracles.pdf [June 6, 2012].

[38] Kanstren T. Program comprehension for user-assisted test oracle generation. *Fourth International Conference on Software Engineering Advances* 2009, 118-127.

[39] Baharom S, Shukur Z. Utilizing an abstraction relation document in grey-box testing approach. *International Conference on Electrical Engineering and Informatics* 2009, 304-308.

[40] Chen J, Subramaniam Sb. Specification-based testing for gui-based applications. *Software Quality Journal* 2002; 10(3): 205-224.

[41] El-Far IK, Whittaker JA. Model-Based Software Testing. *Encyclopedia of Software Engineering* (edited by J. J. Marciniak). Wiley 2011.

[42] Davis A. A Comparison of Techniques for the Specification of External System Behavior. *Communications of the ACM* 1988; 31 (9): 1098-1115.

[43] Sargent RG. Verification and validation of simulation models. *Winter Simulation Conference* 2005, 130-143.

[44] Balci O. Verification, Validation and Accreditation of Simulation Models. *Winter Simulation Conference* 1997, 135-141.

[45] Zhu H, Hall PAV, May JHR. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys* 1997; 29 (4): 366-427.

[46] Basili VR, Selby RW. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transaction on Software Engineering* 1987; 13(2): 1278-1296.

[47] Wood M, Roper M, Brooks A, Miller J. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. *Proceedings ESEC/FSE* 1997, 262-277

[48] Bertolino A, Marchetti E. Software testing. *Guide to the Software Engineering Body of Knowledge SWEBOK* 2004 edition, chapter 5. *IEEE Computer Society* 2004.

[49] Whittaker JA. Stochastic software testing. *Annals of Software Engineering* 1997; 4: 115-131.

[50] El-Far IK. Automated Construction of Software Behavior Models. *Master's Thesis*, Florida Institute of Technology, May 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.4401 [June 6, 2012].

[51] Walton GH, Poore JH. Generating transition probabilities to support model-based software testing. *Software: Practice and Experience* 2000; 30(10): 1095-1106.

[52] Duran JW, Ntafos SC. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering* 1984; 10(4): 438-444.

[53] Hamlet D, Taylor R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering* 1990; 16(12):1402–1411.

[54] Musa JD. Operational Profiles in Software-Reliability Engineering. *IEEE Software* 1993; 10 (2): 14-32.

[55] Marre B, Thévenod-Fosse P, Waeselynck H, Le Gall P, Crouzet Y. An experimental evaluation of formal testing and statistical testing. *Predictably Dependable Computing Systems* 1995, Springer, London, 273-281.

[56] Whittaker JA, Thomason MG. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering* 1994; 20(10): 812-824.

[57] Sayre KD, Poore JH. Stopping criteria for statistical testing. *Information and Software Technology* 2000; 42(12): 851–857.

[58] Littlewood B, Wright D. Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software. *IEEE Transactions on Software Engineering* 1997; 23(11), 673-683.

[59] Dalal SR, Mallows CL. When should one stop testing software? *Journal of the American Statistical Association* 1998; 83(403): 872-879.

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

[61] Offutt J, Liu S, Abdurazik A, Ammann P. Generating Test Data From State-based Specifications. *Software Testing, Verification and Reliability* 2003; 13(1): 25-53.

[62] Mckendall AR, Jin S, Kuppusamy S. Simulated annealing heuristics for the dynamic facility layout problem. *Computers & operations research* 2006; 33(8): 2431-2444.

[63] Freimut B. Developing and Using Defect Classification Schemes, *Technical Report* 2001, IESE-Report 072.01/E, Fraunhofer Institut für Experimentelles Software Engineering.

[64] Frankl PG, Hamlet RG, Littlewood B., Strigini L. Evaluating Testing Methods by Delivered Reliability. *IEEE Transaction on Software Engineering* 1998; 24(8): 586-601.