ÉCOLE CENTRALE DES ARTS ET MANUFACTURES « ÉCOLE CENTRALE PARIS »

THÈSE

Présentée par

Roy AWEDIKIAN

Pour l'obtention du

GRADE DE DOCTEUR

Spécialité : **Génie Industriel**

Laboratoire d'accueil : **Laboratoire Génie Industriel**

SUJET

Qualité de la conception de tests logiciels: plate-forme de conception et processus de test

Quality of the design of test cases for automotive software: design platform and testing process

Soutenue le : **6 février 2009**

Devant un jury composé de :

J. -P. CALVEZ – Professeur, Polytech' Nantes	**Président**
J. -M. FAURE – Professeur, Supmeca et ENS de Cachan	**Rapporteur**
A. KOBI – Professeur, Université d'Angers	**Rapporteur**
L. BOUCLIER – Johnson Controls	**Examinatrice**
P. LEBRETON – Johnson Controls	**Examinateur**
H.B. NEMBHARD – Associate Professor, Penn State University	**Examinatrice**
B. YANNOU – Professeur, Ecole Centrale Paris	**Directeur de thèse**
M. MEKHILEF – MCF, HdR, Université d'Orléans	**Co-directeur de thèse**

2009ECAP0007

Laboratoire Génie Industriel
Ecole Centrale Paris
Grande Voie des Vignes
92925 Châtenay-Malabry Cedex

DEDICACES

A la mémoire de mon grand-père
« Jean EL-HITTI »

REMERCIEMENTS

Je remercie Jean-Marc Faure, Professeur à Supmeca et chercheur au LURPA (Ecole Normal Supérieur de Cachan) et Abdessamad Kobi, Professeur à l'Université d'Angers pour avoir accepté d'être rapporteurs de mes travaux. Merci également à Jean-Paul Calvez, Professeur à l'Ecole polytechnique de l'Université de Nantes d'avoir accepté de participer au jury de cette thèse.

Merci à Bernard Yannou, Professeur à l'Ecole Centrale Paris, qui a accepté de diriger cette thèse. J'ai apprécié le professionnalisme et l'efficacité de son encadrement pendant ces trois années. Bernard a su m'apporter son soutien aux moments décisifs et me pousser quand c'était nécessaire. J'ai beaucoup appris et progressé pendant ces trois années et c'est en grande partie grâce à toi.

Merci également à mon co-encadrant Mounib Mekhilef pour les précieux conseils qui m'ont éclairé tout au long de ces années aussi bien sur le plan méthodologique que sur le plan scientifique.

Je tiens aussi à remercier Harriet Black Nembhard, *Associate Professor* à *Penn State University* pour ses conseils scientifiques et pour avoir accepté de participer au jury en tant qu'examinatrice.

J'exprime également ma reconnaissance à Claude Mignen, Responsable Europe du service *Quality Engineering Process* de la société Johnson Controls et à Tony Jaux, Responsable des centres européens de Recherche et Développement de Johnson Controls qui m'ont accueilli parmi leurs équipes et qui ont œuvré pour que cette thèse se passe dans les meilleures conditions. Un grand merci à eux.

Je voudrais également remercier Line Bouclier (tutrice industrielle de cette thèse à partir de janvier 2007), Responsable *Quality Process* de la société Johnson Controls, pour son soutien technique et administrative auprès de la hiérarchie de la société Johnson Controls. Grâce à sa confiance et à l'autonomie qu'elle m'a accordé tout au long de cette thèse, j'ai pu planifier et mener mes activités en toute souplesse.

Toute ma gratitude à Philippe Lebreton (co-tuteur industriel de cette thèse jusqu'en avril 2007), Coordinateur logiciel de la société Johnson Controls (jusqu'à avril 2008), pour son support technique et professionnel. Philippe a beaucoup contribué à la mise en œuvre et à la valorisation de nos résultats scientifiques au sein de la société Johnson Controls.

Un remerciement tout particulier à Ludovic Augusto, initiateur et porteur de ce projet de recherche industrielle. Chef de projet de la société Johnson Controls et premier tuteur industriel de cette thèse (jusqu'en décembre 2006), j'ai pu bénéficier de ses conseils professionnels et scientifiques.

Merci aussi à Jean-Claude Bocquet, Directeur du Laboratoire Génie Industriel de l'Ecole Centrale Paris, pour m'avoir accueilli dans son laboratoire. Je remercie également l'ensemble du personnel du Laboratoire Génie Industriel pour avoir rendu agréables les moments passés au laboratoire. Tout particulièrement, merci à Sylvie, Anne, Corinne et Carole.

Je remercie également l'ensemble de mes collègues à Johnson Controls pour leur disponibilité chaque fois que je les ai sollicités et avec qui j'ai partagé de très agréables moments.

Enfin, une grand merci à ma famille et mes amis qui m'ont toujours soutenu et motivé depuis le début de cette thèse.

PREFACE

« Quand est ce qu'il faut arrêter de tester un produit logiciel ? » « Comment être sûr qu'un produit logiciel ne contient plus de défauts (*bugs*) et est prêt à être livré au client » et bien d'autres questions sur la qualité logicielle m'ont interpellé dès les premiers stages d'Ecole d'Ingénieur. En effet, diplômé de l'Ecole polytechnique de l'Université de Nantes en 2004, j'ai effectué 3 stages respectifs en 1ère, 2ème et 3ème années. Durant ces stages, j'ai participé au développement de produits logiciels destinés à des applications PC mais aussi à des applications embarquées. A chaque fois qu'on développait un nouveau module logiciel, il nous fallait le tester. Le mot « tester » en industrie est souvent associé à tout type de techniques de vérification et de validation logicielle. Ayant appris en Ecole d'Ingénieur une panoplie de langages de développement informatique (C, C++, …) et notamment comment concevoir et développer un produit logiciel, les tâches de développement informatique me paraissaient simples et maîtrisables. Mais, une fois le logiciel développé il faut le tester ; je me trouvais alors fort dépourvu méthodologiquement ! En effet, les formations actuelles d'Ingénieur logiciel se focalisent presque exclusivement sur le développement logiciel au détriment du test logiciel. Depuis les débuts du développement logiciel (Années 70), des chercheurs ont montré qu'il était illusoire de penser à effectuer un test logiciel exhaustif. Un Ingénieur doit toujours se contenter de tester un sous-ensemble de cas. Bien que certaines entreprises (les grandes) ont des processus bien définis pour tester un produit logiciel, la tâche de comment choisir les cas de test reste en grande partie basée sur l'expérience des Ingénieurs. Pour cela, et afin de tester les modules logiciels pendant mes stages, je choisissais certains cas de test en fonction de leur utilité et de leur efficacité mais aussi du temps qu'il me restait avant de devoir livrer le module.

Après avoir obtenu le diplôme d'Ingénieur en 2004, je me suis intéressé plus généralement à la question : « Comment sont définis les processus de conception et conçus les méthodes et outils de conception de produits ». Afin de répondre à cette question, j'ai effectué un Master Recherche en ingénierie de conception au sein du laboratoire Génie Industriel de l'Ecole Centrale Paris. Le moment du stage arrivé, je me suis mis à la recherche d'un stage qui porterait sur l'amélioration des processus, méthodes et outils de conception de test logiciel pour établir une jonction entre mes domaines de prédilection. Bien heureusement, un stage sur le sujet était proposé par l'équipementier électronique automobile Johnson Controls. L'automobile, un secteur où l'électronique et le logiciel représentent plus de 30% du prix d'un véhicule. Pendant ce stage de 6 mois, nous avons mis en place une nouvelle approche de conception de cas de test logiciel. Le stage a livré des résultats prometteurs tant au niveau de la qualité du test logiciel que du temps passé pour tester un produit logiciel. De plus, nous avons pu identifier plusieurs pistes de recherche prometteuses.

En se basant sur ces pistes de recherche, nous avons formulé un sujet de thèse de doctorat (avec Bernard Yannou[1] et Ludovic Augusto[2]) que nous avons proposé à la société Johnson Controls. En effet, il a fallu mettre en avance l'apport scientifique pour le laboratoire Génie Industriel et surtout l'apport industriel pour la société Johnson Controls qui a financé ce projet en partenariat avec l'ANR sur un statut CIFRE. Suite à une réunion avec des responsables de la société et du laboratoire, l'accord pour lancer ce projet de thèse de doctorat a été donné en janvier 2006. Il est important de noter que la société Johnson Controls (France) n'avait jamais participé à un projet de thèse de doctorat auparavant.

[1] Bernard Yannou, Professeur de l'Ecole Centrale Paris : encadrant de mon stage chez Johnson Controls
[2] Ludovic Augusto, Chef de projet chez Johnson Controls : tuteur industriel de mon stage chez Johnson Controls

RESUME

L'électronique dans les voitures devient de plus en plus complexe et représente plus de 30% du coût global d'une voiture. Par exemple, dans une BMW série 5 modèle 2008, on peut trouver jusqu'à 80 calculateurs électroniques communiquant ensemble et représentant aux alentours de 10 millions de lignes de code logiciel. Face à cette montée en complexité, les constructeurs et équipementiers électroniques de l'automobile s'intéressent de plus en plus à des méthodes efficaces de développement, vérification et validation de modules électroniques. Plus précisément, ils focalisent leurs efforts sur la partie logicielle de ces modules puisqu'elle est à l'origine de plus de 80% des problèmes détectés sur ces produits. Dans ce contexte, nous avons mené un travail de recherche dont l'objectif est de proposer une approche globale d'amélioration de la qualité des logiciels embarqués dans les véhicules. Notre recherche part d'un audit des processus et outils actuellement utilisés dans l'industrie électronique automobile. Cet audit a permis d'identifier des leviers potentiels d'amélioration de la qualité logicielle. En se basant sur les résultats de l'audit et en tenant compte de la littérature dans le domaine de la qualité logicielle, nous avons proposé une approche globale de conception de cas de test pour les produits logiciels. En effet, nous avons développé une plateforme de génération automatique de tests pour un produit logiciel. Cette plateforme consiste à modéliser les spécifications du produit logiciel pour le simuler lors de tests, à se focaliser sur les tests critiques (ayant une forte probabilité de détecter des défauts) et à piloter la génération automatique des tests par des critères de qualité ; telles que la couverture du code et de la spécification mais aussi le coût des tests. La génération de tests critiques est rendue possible par la définition de profils d'utilisation réelle par produit logiciel, ainsi que par la réutilisation des défauts et des tests capitalisés sur des anciens projets. En plus des aspects algorithmiques du test logiciel, notre approche prend en compte des aspects organisationnels tels que la gestion des connaissances et des compétences et la gestion de projet logiciel. Notre approche a été mise en œuvre sur deux cas d'étude réels d'un équipementier électronique automobile, disposant de données de tests historiques. Les résultats de nos expérimentations révèlent des gains de qualité significatifs : plus de défauts sont trouvés plus tôt et en moins de temps.

Mots clés: Vérification et validation logicielle, Automobile, Processus de test logiciel, Simulation fonctionnelle, Qualité logicielle, Gestion des connaissances, Prise de décision, Processus de conception.

ABSTRACT

Nowadays, car electronics become more and more complex and represents more than 30% of the total cost of a car. For instance, in a 2008 BMW 5 series model, one can find up to 80 electronic modules communicating together and representing 10 million lines of software code. Facing this growing complexity, carmakers and automotive electronic suppliers are looking for efficient methods to develop, verify and validate electronic modules. In fact, they focus on the software part of these modules since it accounts for more than 80% of the total number of problems detected on these modules. In this context, we achieved our research project with the aim of proposing a global approach able to improve the quality of automotive embedded software. We started with an audit of the software practices currently used in automotive industry and we pinpointed potential levers to improve the global software quality. Based on the results of the audit and the literature review related to software quality, we developed a global approach to improve the design of test cases for software products. In fact, we developed a test generation platform to automatically generate test cases for a software product. It is mainly based on modeling the software functional requirements in order to be simulated when testing the software, focusing on critical tests to be done (because of their higher probability to detect a bug) and monitoring the automatic generation of tests by quality indicators such as the structural and functional coverage but also the tests cost. The generation of critical tests is based on the definition of real use profiles by software product and on the reuse of bugs and test cases capitalized on previous projects. Besides the computational aspects of software testing, our approach takes into account organizational matters such as knowledge management, competency management and project management. Our approach have been implemented in a computer platform and experimented on two typical case studies of an automotive electronic supplier, with historical test data. The results of our experiments reveal significant improvement in software quality: more bugs are detected earlier and in less time.

Keywords: Software verification and validation, Automotive, Software testing process, Functional simulation, Software quality, Knowledge management, Decision making, Design process.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

- ✓ CAN: Car Area Network
- ✓ CD: Component Development
- ✓ Clk: Clock
- ✓ CLR: CLarification Request list
- ✓ CMMI: Capability Maturity Model® Integration
- ✓ CMP: Software component or module
- ✓ CON: Constraint
- ✓ CPU: Central Processing Unit
- ✓ DEV: DEVelopment
- ✓ DoE: Design of Experiments
- ✓ DT: Decision Table
- ✓ DV: Design Verification
- ✓ ED: Engineering Development
- ✓ Eng: Engineer
- ✓ E-Car: Emulated Car
- ✓ FCT: Functional
- ✓ FSM: Finite State Machine
- ✓ GD: Global Design
- ✓ GUI: Graphical User Interfaces
- ✓ HP: Hewlett-Packard
- ✓ HW: Hardware
- ✓ IEEE: Institute of Electrical and Electronics Engineers
- ✓ INT: Integration
- ✓ IP: Intellectual Property
- ✓ ISO: International Standard Organization
- ✓ JCI: Johnson Controls Inc.
- ✓ KLOC: Kilo Line Of code
- ✓ LAN: Local Area Network
- ✓ LCC: Low Cost Countries
- ✓ LOC: Line Of Code
- ✓ M&A: Merge and Acquisition

- ✓ MMI: Man Machine Interface
- ✓ NIST: National Institute of Standards and Technology
- ✓ NP: Non-deterministic Polynomial time
- ✓ NRE: Non-Recurring Engineering
- ✓ ODC: Orthogonal Defect Classification
- ✓ OP: Operation
- ✓ PM: Project Management
- ✓ PPM: Pieces Per Million
- ✓ PV: Product Validation
- ✓ RAD: Rapid Application Development
- ✓ REQ: REQuirement
- ✓ RETEX: RETurn of Experience
- ✓ RS: Requirement Specification
- ✓ R-Car: Real Car
- ✓ SC: Software Coordinator
- ✓ SD: Software Developer
- ✓ SDLC: Software Development Life Cycle
- ✓ SEPG: Software Engineering Process Group
- ✓ SOP: Start Of Production
- ✓ SPICE: Software Process Improvement and Capability dEtermination
- ✓ SRS: Software Requirement Specification
- ✓ SVE: Software Validation Engineer
- ✓ SVP: Software Validation Plan
- ✓ SW: Software
- ✓ TC: Test Case
- ✓ TS: Test Step
- ✓ UML: Unified Modeling Language
- ✓ VAL: VALidation
- ✓ V&V: Verification and Validation

LIST OF TABLES

LIST OF FIGURES

LIST OF DEFINITIONS

GENERAL INTRODUCTION

I. Context

Nowadays, electronics represents more than 30% of the global cost of a car. Car electronic architecture becomes more and more complex and carmakers outsource the design of electronic modules to automotive electronic suppliers. The software part is the added value of these modules and they account for more than 80% of the total number of defects detected on these modules. As automotive electronic products become more and more complex, the size of software embedded in these products increases drastically. As a consequence, the time spent in verifying and validating these software has increased exponentially the last 10 years. *Verification and Validation* (*V&V*) activities account now for more than 50% of an automotive electronic project time and effort. Despite the huge resources spent in verifying and validating a software product and after each delivery to the carmaker (up to 10 may be made), some bugs are still detected by the carmaker and forwarded to the supplier who must react quickly and efficiently. Once an electronic module is launched on the market (e.g. integrated into a vehicle), an average of one software bug per year is detected by the end-users, which may becomes dramatic for the electronic supplier in financial and image terms if the product has to be systematically substituted.

As the automotive market becomes more and more competing, decreasing the development time of outsourced parts and lowering the number of defects detected later in the process becomes of major importance for carmakers and, consequently, a major quality indicator for automotive suppliers. Indeed, the carmakers' process for assigning new projects to suppliers is mainly based on feedbacks from previous projects. Consequently, suppliers work on reducing the development time of their products, delivering on time the products to carmakers and detecting the maximum number of bugs as early as possible in the development process.

Through our research project (*PhD*), we were asked by Johnson Controls, one of the world's leading suppliers of automotive interior systems, electronics and batteries, to improve the performance of its software *V&V* activities. Their main purpose is to improve the quality of their products and therefore better satisfy the requirements and expectations of their clients. In our research (Awedikian 2007), we go through this problem with a *systemic approach* in order to identify levers in any domains from which we might be able to improve the *global performance* of the software *V&V* activities. The major added value of the present work is to globally solve the quality issue of the *software testing* process.

II. Research process

Our research process is based on five main stages:

Stage 1: Industrial audit

The audit of the industrial context aims to identify and determine the overall environment of our research project. This results in identifying a list of anomalies and issues in the current verification and validation practices.

Stage 2: Research topic definition

Based on the results of the industrial audit, the definition of our research topic allows to better determine the scope and focus of our research. This also leads to a better definition of the state-of-the-art focus.

Stage 3: State-of-the-art

The state-of-the-art on the research issues in the scope of our research pinpoints existing solutions; their advantages, drawbacks and adaptability to our context. This results in a list of potential proposals.

Stage 4: New concepts development

The development of new concepts is the result of the three previous stages. Based on the concepts identified in the literature and taking the requirements of our industrial context into account, new concepts are developed.

Stage 5: Concepts prototyping and validation

The prototype development aims to implement our new concepts in a computer platform. This platform gives us the opportunity to validate our concepts on typical case studies.

Figure Introduction.1 illustrates our research stages all along the three years of the *PhD* cursus.

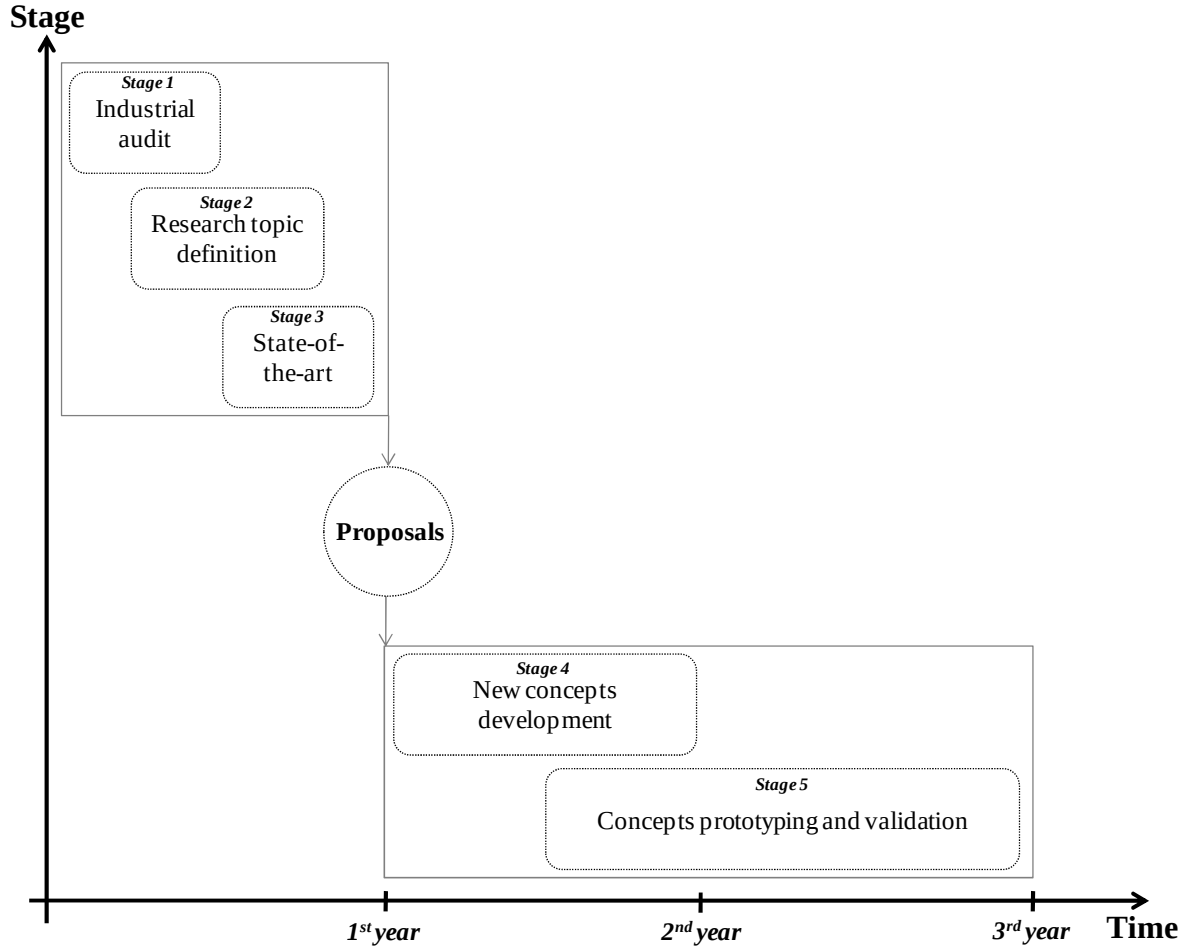


Figure Introduction.1 – Stages of our research process

III. Contributions' overview

Through our research project, we perform an audit on the current software practices in automotive industry. The result of the audit is a list of anomalies and lacks (diagnoses) in the current software *V&V* activities in automotive industry. Based on the audit results and the literature review, we propose *a new systemic approach to automate efficiently the design of*

test cases for software products. Our approach presents a much different workflow than the one presently used in automotive industry. The new workflow is based on eight activities which are *manual*, *semi-automatic* or *automatic* and managed by *different individuals*. These activities are:

1. Model the software functional requirements using a *formal* simulation model that we developed keeping in mind the automotive context and its constraints (Awedikian 2008a).
2. Verify and validate the consistency and compliance of the requirements model (Awedikian 2008a).
3. Define some behavioral characteristics of a car driver when using the software product under test (Awedikian 2008b).
4. Reuse the *test cases* developed in the past for similar software products (Awedikian 2008b).
5. Reuse the bugs detected in the past on similar software products (Awedikian 2008b).
6. Enrich the requirements model with knowledge (from activity 3 to 5) on the driver recurrent *operations* and the test engineers' experience (Awedikian 2008b).
7. Automate the design of *test cases* from the enriched model of functional requirements (Awedikian 2008c).
8. Monitor the design of *test cases* by quality objectives but also time and cost constraints (Awedikian 2008c).

Processes, *roles* and *tools* implementing these activities have been developed. The results of the experiment of our approach on two typical industrial case studies (within Johnson Controls) are very promising. We *reduce by 70%* the number of bugs detected by the carmakers and *by 9%* the ones detected by the end-users. Moreover, we *reduce by 22%* the time spent in testing a software product. In fact, we detect the bugs earlier in the software development process and closer to their origin. We also propose to deliver to the carmaker *formal quality indicators* on the delivered software. All these results contribute to an improvement of the *customer satisfaction* and as a direct impact; the number of tenders will grow. Unfortunately, estimating the cost of software bugs in an organization is a delicate, strategic and confidential question and therefore we were not allowed to communicate the numbers on the bugs' costs savings via the use of our approach.

As a consequence of these results, managers at Johnson Controls decide to *patent our approach*[3]. However and in order to patent an idea in Johnson Controls, a formal verification and validation process of the idea is required. In our case and before starting this process, Johnson Controls has submitted a *worldwide Quick Patent*[4] in order to protect our approach. A worldwide survey on *software testing* approaches has been performed by Johnson Controls patent experts. Moreover, we were formally interviewed by many managers and experts on the contributions of our approach. The final stage will be the decision of the Johnson Controls *Intellectual Property* (*IP*) committee to patent or not our approach.

IV. Reading guidelines

This dissertation is composed of 4 *parts* and each *part* is composed of two or more *chapters*. The structure of the document is illustrated in *Figure Introduction.2*:

[3] Including Bernard Yannou and Mounib Mekhilef as co-inventors.
[4] In France, we associate a *Quick Patent* to an "Enveloppe Soleau" (http://www.inpi.fr/fr/services-et-prestations/enveloppe-soleau.html, Consulted on November 2008).

- *Part I* develops the research context and the industrial audit (*Chapters 1* and *2*).
- *Part II* develops the research topic and the literature review (*Chapters 3* and *4*).
- *Part III* develops our new concepts (*Chapters 5*, *6*, *7* and *8*).
- *Part IV* develops the computer implementation and the validation of our new concepts (*Chapters 9* and *10*).

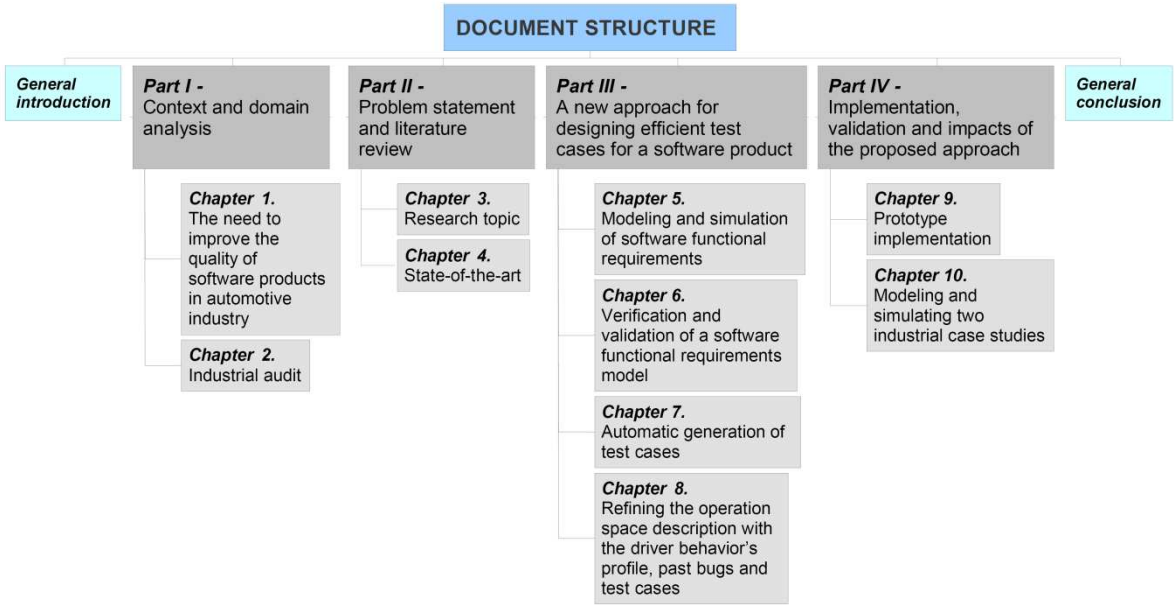


Figure Introduction.2 – Document structure

PART I - CONTEXT AND DOMAIN ANALYSIS

CHAPTER 1. THE NEED TO IMPROVE THE QUALITY OF SOFTWARE PRODUCTS IN AUTOMOTIVE INDUSTRY

I. Introduction

The world of electronics is living a revolution in the way products are imagined, designed and implemented. The ever growing importance of the internet, the advent of microprocessors of great computational power, the burden of wireless communication, the development of new generations of integrated sensors and actuators are changing the world we live and work in.

The car as a self-contained universe is experiencing a similar revolution. We need to rethink what a "car' really is and the role of embedded electronics. Electronics is now essential to control the movements of a car, of the chemical and electrical processes taking place inside, to entertain the passengers, to constantly be connected with the rest of the world and to ensure safety. However, the growth of electronics in a car might reduce its reliability. *Will electronics take the major role in car manufacturing and design? How to control the quality of electronic systems? How to manage the growth of software complexity in automotive electronic parts? What will an automobile manufacturer's core competence be in the next few years? What are the new challenges for automotive electronic suppliers?*

Our intent in this chapter is to answer some of these questions. An illustration of the present automotive industry context facing the globalization and the outsourcing issues is carried out in *Section 2*. Then, the electronic architecture of a modern vehicle is described in *Section 3*, showing the tendency toward incorporating ever more electronics. An overview of the role of software and the new challenges in automotive electronics industry is done in *Section 4* and *5*. Finally, the industrial needs and expectations as it was expressed for the first time by Johnson Controls company are summarized in *Section 6*.

II. The phenomena of globalization and outsourcing in automotive industry

As we enter the new millennium, globalization has emerged as one of the most salient and powerful forces shaping domestic and world economies.

Definition 1.1: Globalization (Wikipedia – November 2008)

Globalization in its literal sense is the process of transformation of local or regional things or phenomena into global ones. It can also be used to describe a process by which the people of the world are unified into a single society and function together. This process is a combination of economic, technological, sociocultural and political forces. Globalization is often used to refer to economic globalization, that is, integration of national economies into the international economy through trade, foreign direct investment, capital flows, migration, and the spread of technology.

The automobile industry is typically considered to be at the forefront of globalization. Evidences supporting this view have been listed by Spatz (Spatz 2002):

- the intricate network of alliances and cross-shareholdings among automobile companies, within nations and regions but also between regions,
- intensified *Mergers and Acquisitions* (*M&A*) activities in the 1990s, involving both end-producers and automotive input suppliers,
- and the trend towards technologically motivated cooperation agreements, which was caused, inter alia, by end-producers entering into new forms of partnerships for the design of principal modules and subsystems.

The new face of globalization in automotive industry is best revealed by the rise of global suppliers. Companies such as Johnson Controls, Bosch, Denso, Lear Corporation, TRW, Magna, and Valeo have become the preferred suppliers for automakers around the world. Some automakers, particularly American firms, have combined a move to "modular" final assembly with increased outsourcing, giving increased responsibility to *first-tier suppliers*[5] for module design and *second-tier sourcing*. Many first tier-suppliers started to build a *vertical integration* (through mergers, acquisitions, and joint-ventures) and to geographically spread so as to be able to provide their customers with modules on a worldwide basis. At the same time, it can be simultaneously observed a *deverticalization* in automaker companies which leads to create a new global-scale supply-base capable of supporting the activities of final assemblers on a worldwide basis.

The drivers of increased outsourcing include 1) the rising technological complexity of vehicle development, 2) rising logistics complexity as more production locations come on-stream, 3) a desire to "streamline" the final assembly process, 4) a desire to pay for parts only as they are incorporated into vehicles rather than when they are shipped from suppliers, 5) increasing competence in suppliers, and 6) a desire to lower costs by moving production to low cost suppliers.

Twenty years ago, automakers practiced low-level parts assembly within final assembly plants, purchased parts based on price, and paid minimal attention to quality. Now, automakers ask suppliers to do more design and sub-assembly work. This refers to as "modularization" in the automotive industry. For example, vehicle doors can be delivered with the glass, fabric, interior panels, handles, and mirrors pre-assembled. Dashboards can be delivered complete with polymers, wood, displays, lights, and switches. The aim of modularization is to move labor out of the final assembly process (design for manufacturability can serve the same purpose).

According to Sturgeon (Sturgeon 2000), fifteen modules represent about 75% of vehicle value. In fact, a supplier can provide groups of related modules, called "module systems". For example, seats, interior trim, and cockpit modules could be supplied as a complete "interior system". *Figure 1.1* provides a graphic representation of the apparent trend from discrete parts to modules and systems.

[5] First-tier supplier means a supplier who directly provides goods and services to the assembly plant of the product

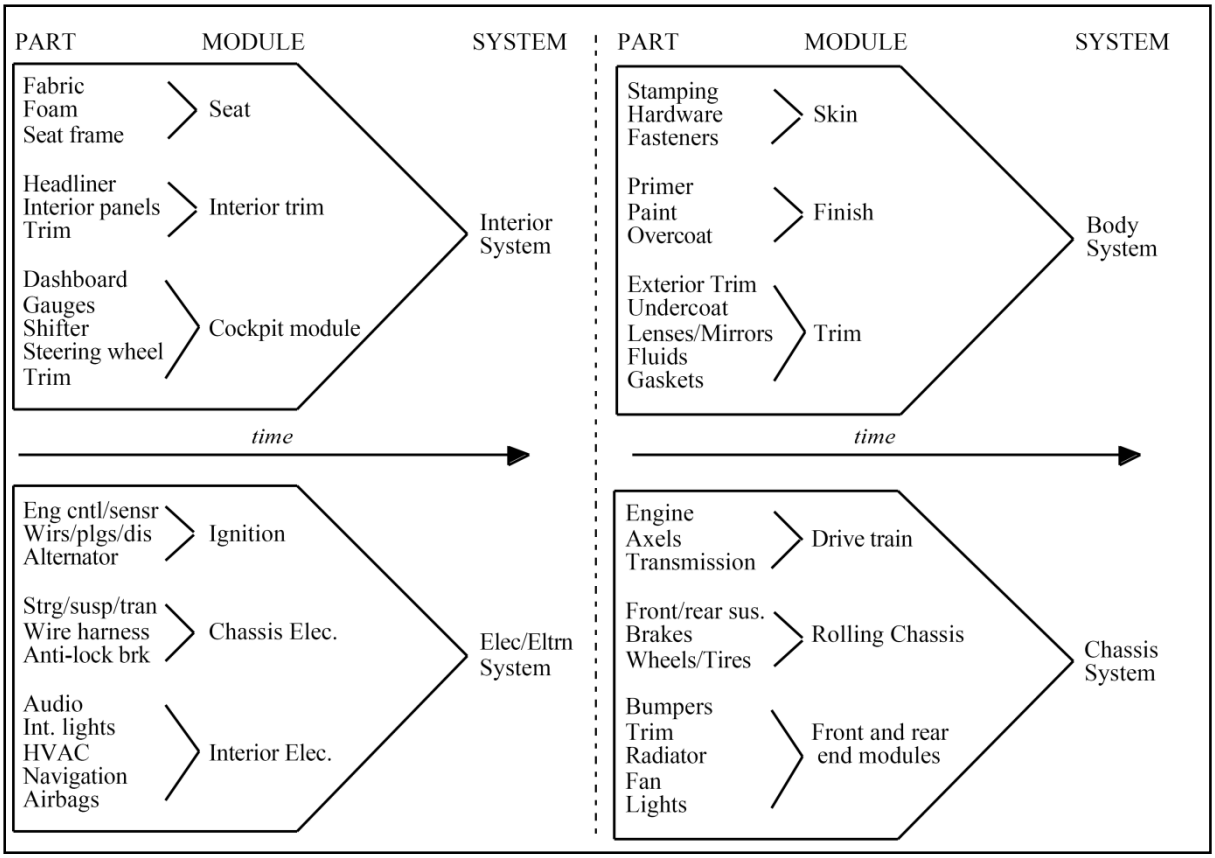


Figure 1.1 – From part to module to system (Sturgeon 2000)

III. Strong growth forecast for electronic parts in automotive

The past four decades have known an exponential increase in the number and sophistication of electronic systems in vehicles. According to Leen (Leen 2002), nowadays, the cost of electronics in luxury vehicles may amount to more than 23 percent of the total manufacturing cost. According to Moavenzadeh (Moavenzadeh 2006), a top *R&D*[6] executive from General Motors said that electronics and software content will account for 40% of the value-added in the vehicle over the new ten years. Moreover, a recent quote[7] from *Daimler* executives says that more than 80% of innovation in the automotive domain will be in electronic modules.

The growth of electronic systems has had implications for vehicle engineering. For example, today's vehicles may have more than 4 kilometers of wiring, compared to 45 meters in vehicles manufactured in 1955. In July 1969, Apollo 11 employed a little more than 150 Kbytes of onboard memory to go to the moon and back. Just 30 years later, a family car might use 500 Kbytes to keep the CD player from skipping tracks.

The resulting demands on power and design have led to innovations in electronic networks for automobiles. Researchers have focused on developing electronic systems that safely and efficiently replace entire mechanical and hydraulic applications. Just as *LANs*[8] connect computers, control networks connect a vehicle's electronic equipment. These networks facilitate the sharing of information and resources among the distributed applications. In the past, wiring was the standard means of connecting one element to another. As electronic

[6] R&D: Research and Development
[7] http://www.dsp.acm.org/view_lecture.cfm?lecture_id=86 (consulted on November 2008)
[8] LAN: Local Area Network

content increased, the use of more and more discrete wiring hit a technological wall. Fortunately, today's control and communications networks, based on serial protocols, counter the problems of large amounts of discrete wiring. Beginning in the early 1980s, centralized and then distributed networks have replaced point-to-point wiring.

Figure 1.2 shows the sheer number of systems and applications contained in a modern automobile's network architecture.

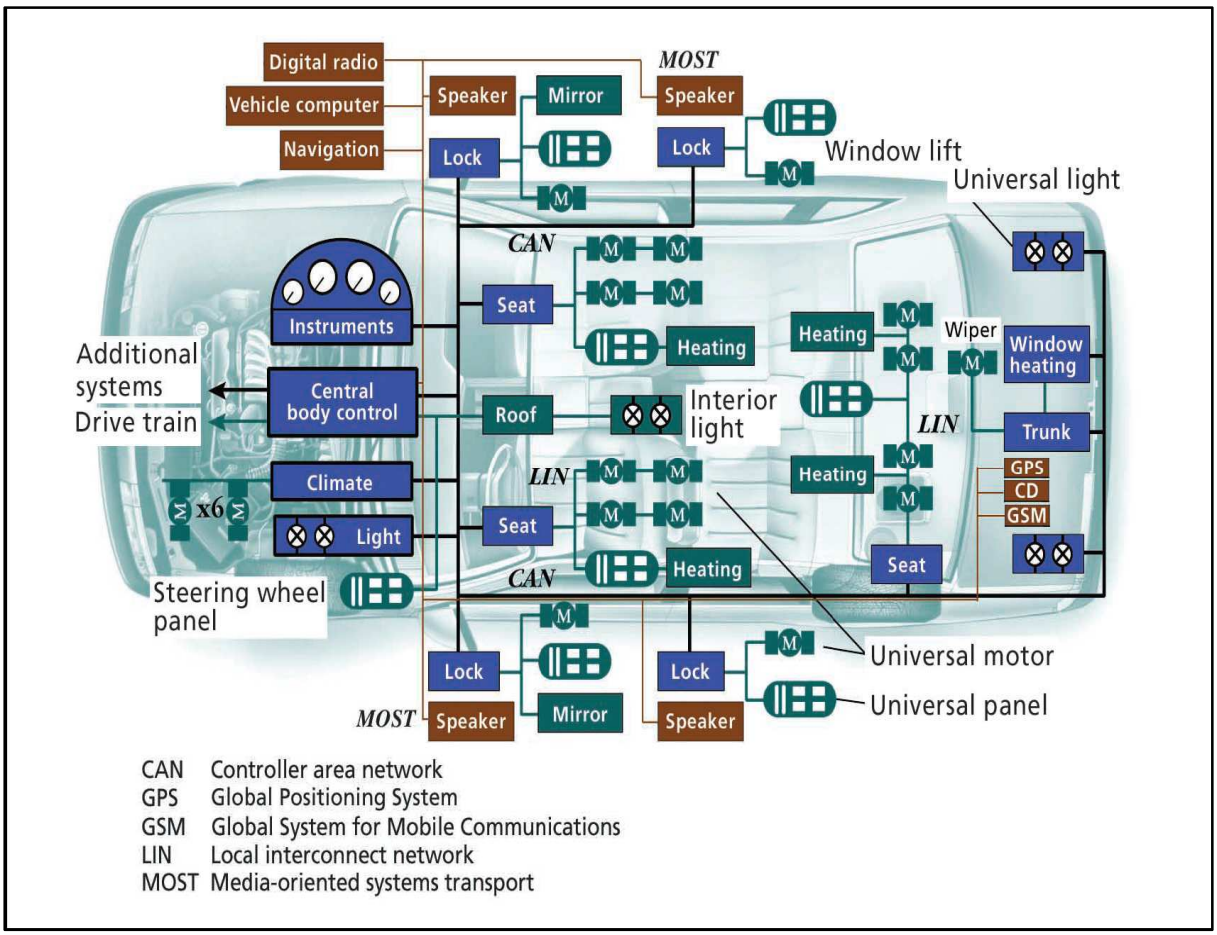


Figure 1.2 – A modern vehicle's network architecture (Leen 2002)

The electronics and software content of vehicles relies more on electrical and software engineers than on the traditional mechanical engineers associated with the automotive industry. Moavenzadeh (Moavenzadeh 2006) indicates that electronics and software engineering functions are easier to outsource or offshore than mechanical engineering functions. Software engineers across an ocean can discuss a few lines of code easier than mechanical engineers can discuss how to modify the design of a module. Software and electronic systems also tend to follow a more modular product architecture than mechanical systems; therefore, it is easier to offshore both low and high value added functions.

IV. Challenges for the automotive electronic suppliers

The overall goal of electronic embedded system design is to balance production costs with development time and cost in view of performance and functionality considerations. In other words, engineers are encouraged to shorten the overall design and validation cycle without compromising quality, reliability, and cost targets.

A. Lower the production cost

Manufacturing costs mainly depend on the hardware modules of the product. If one considers an integrated circuit implementation, the size of the chip is an important factor in determining production cost. Minimizing the size of the chip implies tailoring the hardware architecture to the functionality of the product. However, the cost of a state-of-the-art fabrication facility continues to grow up. In addition, the *Non-Recurring Engineering* (*NRE*) costs associated with the design and tooling of complex chips are rapidly growing. As a consequence, a common hardware platform to be shared across multiple applications may increase the production volume and decrease the overall cost.

B. Lower the development time and cost

Since the times of ignition car electronics in the 1970s, the complexity of automotive electronics architecture is still growing. Presently, Leen (Leen 2002) notices that a *BMW 5 series* model can have up to 80 electronic control units. However, the market dynamics for automotive electronic systems leads to shorter and shorter development times. Presently, no matter how complex the design problem is, suppliers don't have more than six months from the delivery of the customer requirements to a first and correct implementation. To meet the design time requirements and ensure a high quality of the delivered product, a design methodology that favors automation, reuse and early problem detection is essential. This implies that the design activity must be rigorously defined, so that all stages are clearly identified and appropriate checks are enforced.

V. The role of "software" in automotive electronics

Lets us start by defining what is a *software*. In this dissertation, we adopt the definition of software proposed by the *Institute of Electrical and Electronics Engineers*[9] (*IEEE*).

Definition 1.2: Software or Software product (IEEE Std. 610-1990) – Abbreviation: SW

Software is a general term used to describe a collection of computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Moreover, a software product is composed from a set of *software components* or *modules* (Cf. *Figure 1.3*).

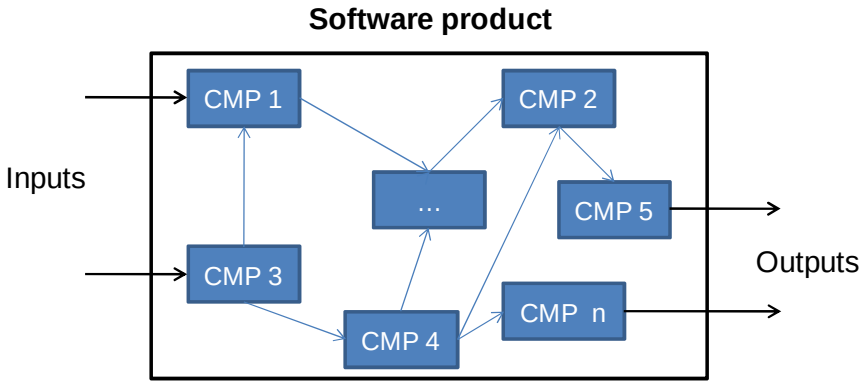


Figure 1.3 – Software product versus software component

[9] http://www.ieee.org/portal/site (Consulted on November 2008)

Definition 1.3: Software component or module (Wikipedia – November 2008)

A software component or a software module is a system element offering a predefined service or event, and able to communicate with other components or modules. It is a minimal software item that can be tested in isolation.

A software product can be dedicated for different types of electronics architecture (computer, automotive, airplane etc.). In our research, we only consider the software products written for machines that are not, first and foremost, computers. In software engineering, this type of software products is called *embedded software*. For example, it is embedded to electronics in cars, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, televisions and digital watches. This type of software products can become very sophisticated in applications like airplanes, missiles, process control systems, and so on. Embedded software is usually written for special purpose electronics architecture. For instance, the *CPUs*[10] are different from general purpose *CPUs* that we could find in our desktops or laptops. Moreover, a real-time operating system is required for managing the simulation of embedded software. In fact, tasks' scheduler and priorities are the fundamentals of a real-time operating system.

The design process of software products shares many technologies with the one of hardware products. Nevertheless, there are important differences between the two types. In the following, we identify some of these differences:

- the hardware product quality relies on design, implementation and manufacturing processes, however the software product quality relies only on design and implementation processes. The software manufacturing process is mainly based on a "simple" reproduction activity,
- a software product is able to simulate alternative commands on different inputs which lead to a high complexity of the product,
- a software product is not a physical entity and therefore, it doesn't wear out over time. In fact, since problems are detected and corrected, the quality of a software product improves over time. However, the correction and/or evolution activities can introduce new problems in the product,
- software problems cannot be prevented. In fact, specific successive commands on the inputs of the software product can reveal a problem which was not detected during the *testing* activities of the product,
- the easiness and the rapidity which with a software product can be modified lead to the fact the software development process should be very well monitored and documented,
- and historically, software modules are not frequently standardized and reused. Nowadays, there is a trend toward a reuse of software modules in order to lower the development time and cost.

One more specific concept to software products is the size. Software sizing (Wikipedia – November 2008) is an important activity in software engineering that is used to estimate the size of a software module. Size is an inherent characteristic of a software in just like weight is an inherent characteristic of any tangible material. Historically, the most common software sizing methodology was counting the *Lines Of Code* (*LOC*) written in the application source. Another famous sizing method is the Function Point analysis. New trends of software sizing have recently emerged.

[10] CPU: Central Processing Unit

A. Growth of software size in automotive electronic parts

The amount of software in many electronic products is increasing rapidly. For example, the number of lines of *source code* in a mobile phone is expected to increase from 2 million today to 20 million by 2010; a car will contain 100 million lines of code (Charrette 2005). Consequently, electronics companies no longer find it economically viable to provide all the software in their products.

In fact, the amount of software in cars grows exponentially. Within only thirty years, the amount of software in a car has evolved from zero to tens of millions lines of code. A current premium car, for instance, implements about 270 functions a user interacts with, deployed over about 70 embedded platforms. Altogether, the software amounts to about 100 megabytes of binary code. The next generation of upper class vehicles, hitting the market in about 5 years, is expected to run up to 1 gigabyte of software. This is comparable to what a typical desktop workstation runs today.

A first reason for this growing complexity in software is that software enables the implementation of functionality deemed impossible just twenty years ago. Another reason is that electronics in cars helps to reduce gas consumption and increase performance, comfort and safety, as indicated by the decreasing number of major accidents whereas traffic increases. Information processing technology cuts across all aspects of the car and is a persuasive, sophisticated and differentiating value addition to the product. Furthermore, software enables the car manufacturers and suppliers to tailor systems to particular customers' needs. In other words, software can help differentiate between cars. At least in principle, it is the software that also allows hardware to be reused across different cars. Contrarily to hardware, software has an almost negligible replication cost, which is a further incentive to bet on software as a potential tool in cost-reduction. However, the growing complexity of automotive software products leads to a dramatic increase of the software development costs. In addition, growing complexity is a driver for numerous challenges in the automotive industries like: definition of key competencies, processes, methods, tools, models, product structures, division of labor, logistics, maintenance, and long term strategies.

B. Software Development Life Cycle

The *Software Development Life Cycle* (*SDLC*) models describe activities of the software cycle and the order in which those activities are executed. A variety of *SDLC* models have been proposed in a paper (Green 1998), most of which focus exclusively on the development activities: *ad-hoc model*, *waterfall model*, *V-model*, *iterative and incremental model*, *prototyping model*, *rapid application development model*, *exploratory model* and *spiral model*.

1. Ad-hoc model

Early systems development often took place in a rather chaotic and haphazard manner, relying entirely on the skills and experience of the individual staff members performing the work (Cf. *Figure 1.4*). Today, many organizations still practice *Ad-hoc Development* either entirely or for a certain subset of their development (e.g. small projects).

Figure 1.4 – Development based on the skills and experience of the individual staff members performing the work

In the absence of an organization-wide software process, repeating results depends entirely on having the same individuals available for the next project. Success that rests solely on the availability of specific individuals provides no basis for long-term productivity and quality improvement throughout an organization.

2. The waterfall model

The *waterfall model* prescribes a sequential execution of a set of development and management activities (Cf. *Figure 1.5*). Some variants of the *waterfall model* allow revisiting the immediately preceding activity ("feedback loops") if inconsistencies or new problems are encountered during the current activity. The *waterfall model* is the earliest method of structured system development. Although it has been criticized in recent years for being too rigid and unrealistic when it comes to quickly meeting customer's needs, the *waterfall model* is still widely used. It provides the theoretical basis for other process models.

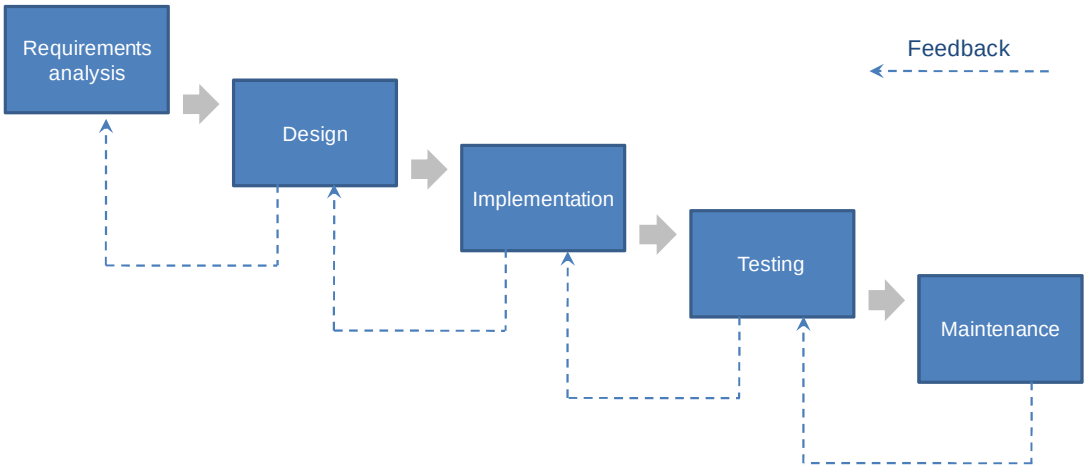


Figure 1.5 – Waterfall development model

3. The V-model

A variant of the *waterfall model* - the *V-model* - associates each development activity with a *Verification and Validation* (*V&V*) activity at the same level of abstraction (Cf. *Figure 1.6*). Each development activity builds a more detailed model of the system than the one before it, and each *V&V* activity tests a higher abstraction than its predecessor.

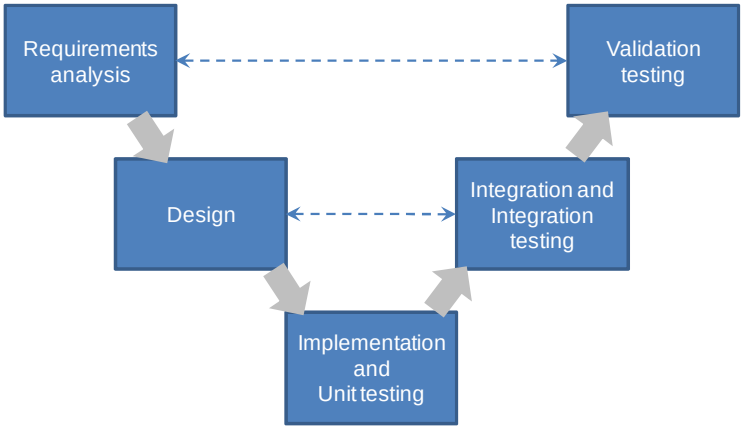


Figure 1.6 – V development model (V-model)

4. The iterative and incremental model

The problems with the *waterfall model* created a demand for a new method of developing systems which could provide faster results, require less up-front information, and offer greater flexibility (Cf. *Figure 1.7*). With iterative development, the project is divided into small parts. This allows the development team to demonstrate results earlier in the process and obtain valuable feedback from system users. Often, each iteration is actually a mini-waterfall process with the feedback from one activity providing vital information for the design of the next activity.

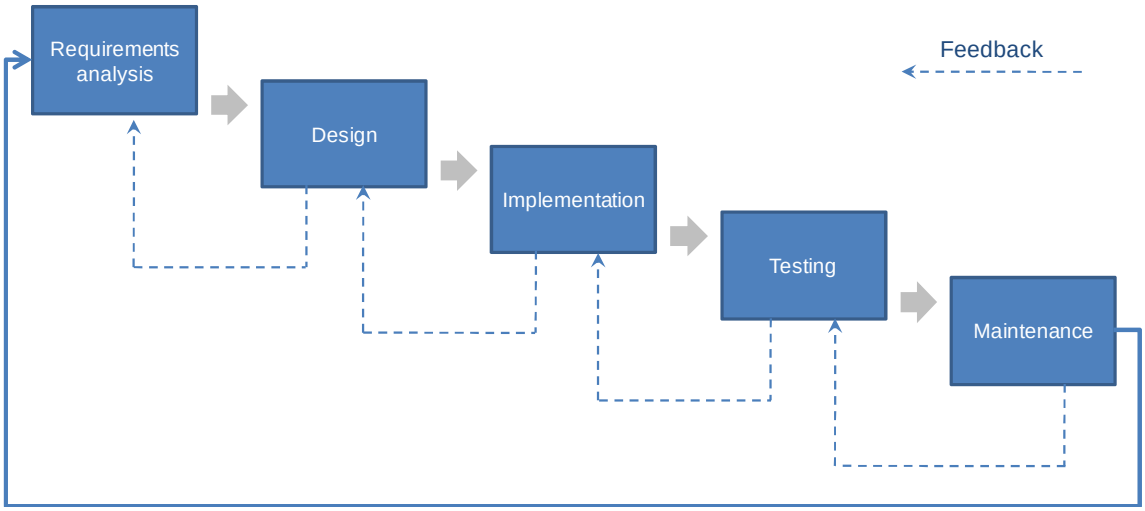


Figure 1.7 – Iterative and incremental development model

5. The prototyping model

The *prototyping model* was developed on the assumption that it is often difficult to know all of your requirements at the beginning of a project. Typically, users know many of the objectives that they wish to address with a system, but they do not know all the nuances of the data, nor do they know the details of the system functionalities and capabilities. The *prototyping model* allows for these conditions, and offers a development approach that yields results without first requiring all information up-front.

When using the *prototyping model*, the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who goes back to refine the system requirements to incorporate the additional information.

6. The rapid application development model

A popular variation of the prototyping model is called *Rapid Application Development* (*RAD*). *RAD* focuses on developing a sequence of evolutionary prototypes which are reviewed with the customer, both to ensure that the system is developing toward the user's requirements and to discover further requirements. The process is controlled by restricting the development of each integration to a well-defined period of time, called a time box. Each time box includes analysis, design, and implementation of a prototype.

7. The exploratory model

In some situations it is very difficult, if not impossible, to identify any of the requirements for a system at the beginning of the project. Theoretical areas such as *Artificial Intelligence* are candidates for using the *exploratory model*, because much of the research in these areas is based on guess-work, estimation, and hypothesis. In these cases, an assumption is made as to how the system might work and then rapid iterations are used to quickly incorporate suggested changes and build a usable system. A distinguishing characteristic of the *exploratory model* is the absence of precise specifications. *Validation* is based on the consistency of the end results and not in compliance with existing requirements.

8. The spiral model

The *spiral model* is similar to the incremental model, with more emphases placed on risk analysis. The *spiral model* has some resemblance to *Deming's "Plan, Do, Check, Act"* cycle and has four activities: *Planning*, *Risk Analysis*, *Engineering* and *Evaluation* (Cf. *Figure 1.8*). A software project repeatedly passes through these activities in iterations (called Spirals in this model). In the planning activity, requirements are gathered and risk is assessed. In the risk analysis activity, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis activity. Software is produced in the engineering activity, along with *testing* at the end of the activity. The evaluation activity allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the *spiral model*, the angular component represents progress, and the radius of the spiral represents cost.

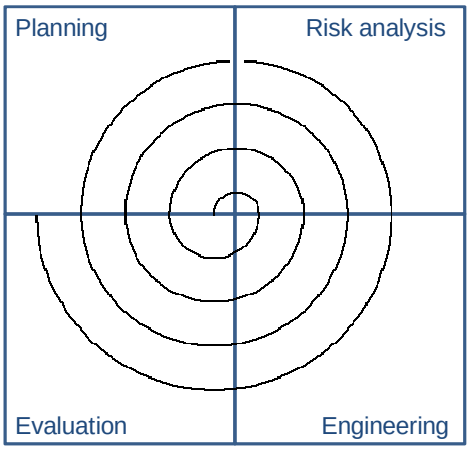


Figure 1.8 – Spiral development model

9. Common framework between software development life cycles

In conclusion, there are a lot of models and many companies adopt their own, but all have very similar patterns. The general, basic model is shown in *Figure 1.9*:

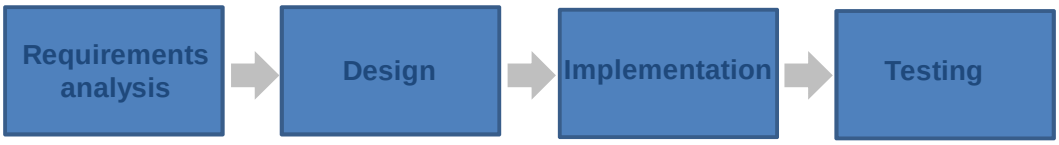


Figure 1.9 – General software development life cycle model

Each activity produces deliverables required by the next activity in the life cycle. *Requirements* are translated into *design*. Code is produced during *implementation* that is driven by the design. *Testing* verifies the deliverable of the *implementation* activity against requirements.

a. Requirements analysis

Customer requirements are gathered in this activity. Meetings are held in order to determine the requirements. Who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system? These are general questions that get answered during a requirements gathering activity. This produces a list of functionality that the software product must provide.

b. Design

The software system *design* is produced from the results of the requirements analysis activity. In this activity, the details on how the system has to work are produced. Architecture (including hardware and software), communication and software design are all part of the deliverables of a design activity.

c. Implementation

Code is produced from the deliverables of the design activity during implementation. Implementation may overlap with both the design and *testing* activities. Many tools exist to actually automate the production of code using information gathered and produced during the design activity.

d. Testing

During *testing*, the implementation is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements analysis activity. *Unit*, *integration* and *validation tests* are done during this activity. *Unit tests* act on a specific module of the system, while *integration* and *validation tests* act on the system as a whole.

C. Two complementary approaches to design “bug-free” software

Let us start this section by giving a definition for the term *bug*. In this dissertation, we adopt the definition proposed by *IEEE*.

Definition 1.4: Mistake, error, fault, failure/bug (IEEE Std. 610-1990)

"Mistake" is a human action that produces an incorrect result.

"Error" is a difference between a computed result and the specified or theoretical one.

"Fault" is a defect in a module which is the manifestation of an error.

"Failure" is the inability of a system to perform a required function within specified limits.

The relation between a mistake, an error, a fault and a failure is illustrated in Figure 1.10.

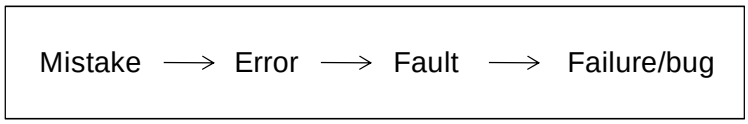


Figure 1.10 – Relation between a mistake, an error, a fault and a failure

The results of a software testing activity is a failure, therefore an analysis activity is necessary to identify the fault.

In our research, we used the term "bug" rather than "failure"

Software's complexity and accelerated development schedules make designing *"bug-free"* software difficult. In this dissertation, we also adopt the definition of software quality proposed by *IEEE*.

Definition 1.5: Software quality (IEEE Std. 610-1990)

(1) The degree to which a system, module, or process meets specified requirements.

(2) The degree to which a system, module, or process meets customer or user needs or expectations.

A widely accepted premise on software quality is that software is so complex (in *combinatorial* terms) that it is impossible to have *"bug-free"* software. One technique commonly used in industry to verify and validate a software product is the *software testing*. In this dissertation, we adopt the definition of *software testing* proposed by the *National Institute of Standards and Technology*[11] (*NIST*).

Definition 1.6: Software testing and execution (NIST 2002)

Software testing is the process of applying metrics to determine product quality. Software testing is the dynamic execution of software and the comparison of the results of that execution against a set of pre-determined criteria. ***"Execution"*** *is the process of running the software on a computer with or without any form of instrumentation or test control software being present. "Predetermined criteria" means that the software's capabilities are known prior to its execution. What the software actually does can then be compared against the anticipated results to judge whether the software behaved correctly. Software testing is a widespread V&V technique in automotive industry.*

In *Chapter 8 – Section 2*, we demonstrate that the *software testing* problem is a *NP-Complete*[12] *problem*. We often hear maxims like "there's always one more bug", and "software *V&V* techniques can reveal the existence of bugs, but never prove their absence".

[11] http://www.nist.gov/ (Consulted on November 2008)
[12] NP: Non-deterministic Polynomial time

The *Figure 1.11* released by Liggesmeyer (Liggesmeyer 1998) shows that the main part of bugs are introduced during the first life cycle of the software development (around 90% in requirements analysis, design and implementation activities) and detected in the last activities (around 80% during *unit test*, *validation test* and *serial production*). A study done in 2006 by a Johnson Controls software expert (Le Corre 2006) on about 15 projects from different types of products has confirmed the findings of Liggesmeyer.

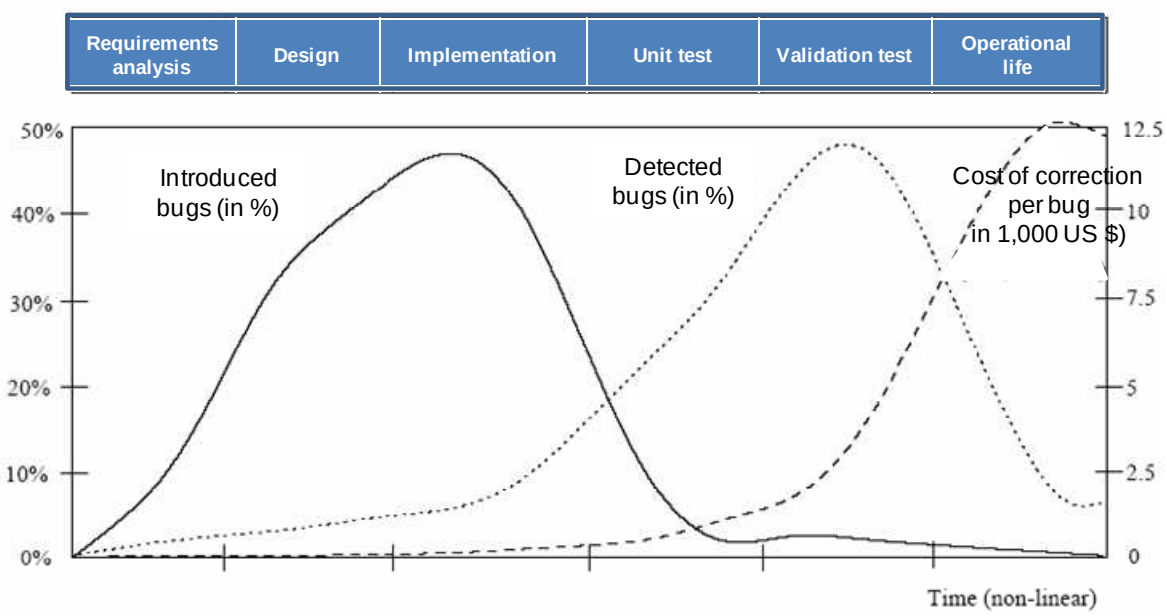


Figure 1.11 – Rate and cost of bugs introduced and detected across the software development life cycle (Liggesmeyer 1998)

Therefore, appropriate techniques, methods and procedures must be adopted in order to help engineers to:

- lower the number of bugs introduced in the software system (*prevention approach*),
- and detect all the bugs that have been introduced in the software system as soon as possible (*detection approach*).

1. Prevention approach (Mays 1990, Gale 1990, McDonald 2007)

Bugs are a consequence of the nature of human factors in the designing task. They arise from oversights made by engineers during *requirements analysis*, *design*, *implementation* and even *testing*. Complex bugs can arise from unintended interactions between different parts of the software system. This frequently occurs because software systems can be complex - millions of lines long in some cases - often having been programmed by many people over a great length of time, so that engineers are unable to mentally track every possible way in which parts can interact. The software industry has put much effort into finding methods for preventing engineers from inadvertently introducing bugs while designing a software system. These methods include:

- Engineers practices
- Standards
- Formal languages
- Prototyping
- Modeling and simulation
- Reuse

- Root cause analysis

Even with efficient bugs' prevention techniques and taking the assumption that human is not perfect, we conclude that each software system includes bugs. That is why, verifying and validating the software system before any customer delivery is a necessary activity.

2. Detection approach: V&V techniques (Beizer 1995, Myers 1978, So 2002)

Finding and fixing bugs has always been a major part of designing software systems. Maurice Wilkes[13] , an early computing pioneer, said in the late 1940s (Wilkes 1949) that much of the rest of his life would be spent finding errors in his own programs. As computer programs become more complex, bugs become more common and hard to fix. Often programmers spend more time and effort finding and fixing bugs than writing new code. Theoretical data proposed by *Brooks*[14] (Brooks 2007) but also the results of a study done in 2006 by a Johnson Controls software expert (Le Corre 2006) on about 5 projects from different types of products are presented in *Table 1.1*. The study points out that the *validation test* activity takes up to 50% of the total development duration of a project. Moreover, a recent study within Johnson Controls (2008) has shown that this ratio number has been exceeded.

Software life cycle	Johnson Controls (%)	F. Brooks (%)	Hewlett-Packard (%)
Requirements analysis	40	33	37
Design and implementation	20	17	34
Unit test		25	
Integration and validation test	40	25	29

Table 1.1 – Time and effort spent during the software development life cycle (Brooks 2007, Le Corre 2006)

Usually, the most difficult part of finding a bug is locating the erroneous part of the *source code*. Once the error is found, correcting it is usually easy. Programs known as debuggers exist to help programmers locate bugs. However, even with the aid of a debugger, locating bugs is something of an art. It is not uncommon for an error in one section of a program to cause bugs in a completely different section, thus making it especially difficult to track. Typically, the first step in locating a bug is finding a way to reproduce it easily. Once the bug is reproduced, the programmer can use a debugger or some other tool to monitor the execution of the program in the faulty region, and find the point at which the program went astray. It is not always easy to reproduce bugs. Some bugs are triggered by inputs to the program which may be difficult for the programmer to re-create.

Therefore, bugs' detection is still a tedious task requiring considerable manpower. Since the 1990s, particularly following the *Ariane 5 Flight 501* disaster, there has been a renewed interest in the development of effective automated aids to remove bugs but it's still remaining much of a work in progress. Presently, bugs' detection techniques (also called software *V&V* techniques) can be classified into two classes:

[13] http://www.cl.cam.ac.uk/~mvw1/short-biography.html (consulted on November 2008)
[14] Frederick P. Brooks is a pioneer of software engineering, http://www.cs.unc.edu/~brooks/ (consulted on November 2008)

- *Static techniques* (*Review* and *Proof*) which do not require the execution of the software under test (Ayewah 2008)
- *Dynamic techniques* (*Testing*) which require the execution of the software under test (Beizer 1990, Barezi 2006).

Each of these techniques catches different classes of bugs at different points in the development cycle.

D. Impacts of detecting bugs later in the software development life cycle

According to a newly released study commissioned by the Department of Commerce's *National Institute of Standards and Technology* (NIST 2002), software bugs cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved *V&V* infrastructure that enables earlier and more effective identification and removal of software bugs. These are the savings associated with finding an increased percentage of bugs closer to the development activities in which they are introduced. Currently (Cf. *Table 1.1*), over half of all errors are not found until the last *testing* activity in the development process (*validation test*) or during post-sale software use (*operational life*).

The impact on the software industry due to lack of robust, standardized *V&V* technology able to detect bugs closer to where they are introduced can be grouped into three general categories:

1. Poor quality perceived by the customer

The most troublesome effect of a lack of efficient *V&V* technology is the increased incidence of avoidable bugs that emerge after the product has been delivered to the customer. Poor quality often results in loss of reputation and loss of future business for the company. In addition, legal actions are undertaken against the supplier when bugs are attributable to insufficient *V&V*.

2. Increase of the software development cost

Historically, the process of identifying and correcting bugs during the software development process represents over half of development costs. Depending on the accounting methods used, *V&V* activities account for 30 to 90 percent of labor expended to produce a working program (Beizer 1990). Software engineers already spend approximately 50 percent of development costs on identifying and correcting bugs (Cf. *Table 1.1*). Early detection of bugs can greatly reduce costs. Bugs can be classified by where they were found or introduced along the activities of the software development life cycle, namely, *requirements analysis*, *design*, *implementation*, *testing*, and *operational life* activities. *Figure 1.11* illustrates that the longer a bug stays in the program, the more costly it becomes to fix it.

3. Increase of the time to market

The lack of efficient *V&V* technology also increases the time that it takes to bring a product to market. Increased time often results in lost opportunities. For instance, a late product could potentially represent a total loss of any chance to gain any revenue from that product. Lost opportunities can be just as damaging as post-release product bugs. However, they are

notoriously hard to measure. If efficient *V&V* techniques were readily available, engineers would expend less time developing custom *V&V* technology.

VI. Industrial needs and expectations

Nowadays, electronics represents more than 30% of the global cost of a car (Sangiovanni-Vincentelli 2003). Car electronic architecture becomes more and more complex and carmakers outsource the design of electronic modules to automotive electronic suppliers. The software part is the added value of these modules and they account for more than 80% of the total number of problems detected on these modules (Johnson Controls source). As automotive electronic products become more and more complex, the size of software embedded in these products increases drastically. In fact, a *body controller module* managing the interior function of a car body account for more than 200 *KLOC*[15] (Johnson Controls source). As a consequence, the time spent in verifying and validating these software has increased exponentially the last 10 years. *V&V* activities account now for more than 50% of an automotive electronic project time and effort (Cf. *Table 1.1*). Despite the huge resources spent in verifying and validating a software product and after each delivery to the carmaker, some bugs are detected by the carmaker and forwarded to the supplier who must react quickly and efficiently. Once an electronic module is launched on the market (e.g. integrated into a vehicle), an average of one software bug per year is detected by the end-users, which may becomes dramatic for the electronic supplier in financial terms if the product has to be systematically changed. In fact, in term of bug's occurrence, two types of contract engage electronics suppliers with car manufacturers:

- *Implicit contract:* during software development process, each carmaker delivery must be free of bugs.
- *Explicit contract:* on launched electronic module, carmaker tolerates a certain number of defective products expressed in terms of *PPM* (Pieces Per Million). *PPM* includes all software bugs but also electronic, mechanical and production problems.

As the automotive market becomes more and more competing, decreasing the development time of outsourced parts and decreasing the number of problems detected later in the process becomes of major importance for carmakers and consequently a major quality indicator for automotive suppliers. Indeed, the carmakers' process for assigning new projects to suppliers is mainly based on feedbacks from previous projects. Consequently, suppliers work on reducing the development time of their products, delivering on time the products to carmakers and detecting the maximum number of bugs as earlier as possible in the development process.

Through our research project, we were asked by an automotive electronic supplier namely Johnson Controls to improve the performance of its software *V&V* activities. Their main purpose is to improve the quality of their products and therefore better satisfy the requirements and expectations of their clients. In Johnson Controls, the software development life cycle follows a *V-model* (Cf. *Chapter 2 – Section 3.B*). Moreover, the validation test which is the last *V&V* activity before a carmaker delivery is considered as the ultimate activity to detect all the bugs and therefore deliver carmakers "bug-free" software. It represents up to 90% of the time spent in the *V&V* of a software product (Johnson Controls source). *Testing* a software product requires two main activities. A detailed specification of these activities is done in *Chapter 2 – Section 5*. The first one consists of designing *test cases* and the second

[15] KLOC : Kilo Lines Of Code.

one of executing these *test cases* on the software product under test. We adopt the definition of *test case* proposed by *IEEE*.

Definition 1.7: Test Case (IEEE Std. 610-1990) – Abbreviation: TC

IEEE defines test case as follows:

(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

(2) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

While the execution activity is often automated via a *test execution platform*, the *test case* design activity remains a manual task that fills in time many engineers. We propose below our definition of a *test execution platform*.

Definition 1.8: Test execution platform

A test execution platform is a platform that aims to simulate the environment of and perform test inputs on the software module or product under test. Therefore, one can verify and validate the behavior of the software under test in its real environment. For more information regarding the Johnson Controls test execution platforms, please refer to Appendix C.

Up to 50% of a software project time is dedicated to design *test cases* (Cf. *Table 1.1*). Therefore, for many of software managers and experts in Johnson Controls, *automating the design of test cases* seems to be the most adapted solution to reduce the *testing* time, cost and resources while improving the *code* and *requirement coverage*. We adopt the definition of *coverage* proposed by *IEEE*.

Definition 1.9. Coverage (IEEE Std. 610-1990)

In software engineering, the term "coverage" means the degree, expressed as a percentage, to which a specified coverage item (code or requirement) has been exercised by a test case. For the definition of code (structural) and requirement (functional) coverage, please refer to Chapter 2 – Section 5.F.

In our research, we go through this problem with a *systemic approach* in order to identify levers in any domains from which we might be able to improve the *global performance* of the software *V&V* activities. The added value of such an approach is the resolution of the problem with a *global quality* viewpoint. Consequently, in *Chapter 2*, we characterize the software design environment in automotive industry and point out issues and anomalies (diagnoses). In *Chapter 3*, based on our *industrial audit*, we clearly define the scope of our research and we formulate our research topic in accordance with the research issues in *software testing*. In *Chapter 4*, we perform a *literature review* on the existing approaches, techniques and tools in the field of the *V&V* of software products. More especially, we focus our research on finding or adapting "solutions" for the anomalies and lacks (diagnoses) that we identify via our industrial audit. We identify relevant actions for improving the global performance of the Johnson Controls *V&V* activities. In *Chapter 5*, *6*, *7* and *8*, we specify our proposed models. A prototype implementing our models has been developed in *Chapter 9*. Finally, in *Chapter 10*, we *validate* our models through two industrial case studies on historical data.

VII. Conclusion

Presently, automotive industry is facing significant difficulties in terms of selling new cars. Therefore, carmakers ask their suppliers to innovate, increase quality, reduce time to market and decrease the development cost. As electronics represents more than 30% of the global cost of a car and stands for a big amount of the problems detected on a car, electronic suppliers are the most concerned. Software technology is at the core of each electronic product; therefore, electronic suppliers focus their efforts on the improvement of their software development and *V&V* practices. Through our research project, we were asked by an automotive electronic supplier namely Johnson Controls to improve the performance of its software *V&V* activities. Their main purpose is to better satisfy the requirements and expectations of their clients in terms of quality, cost and delay. In our research, we go through this problem with a *systemic approach* in order to identify domains from which we might be able to improve the global performance of the Johnson Controls software *V&V* activities.

In the following chapter, we perform an industrial audit on the software practices and more especially on the *V&V* techniques currently used in automotive industry. We aim to identify the issues and lacks of the current practices in order to propose relevant improvement actions well adapted to the industrial context.

CHAPTER 2. INDUSTRIAL AUDIT

I. Introduction

The audit of the industrial context permits to identify and determine the overall environment in which our research project has to be performed. This must results in a better understanding of what verifying and validating a software product means and what are the necessary changes to perform.

In this chapter, we perform an *industrial audit* on the software practices currently used in automotive industry and more especially in Johnson Controls. The audit is divided into four parts:

- The process of managing the carmakers' requirements related to the software domain. In fact, delivered software products must be compliant with the carmaker's requirements.
- The processes of verifying and validating software products. Many *Verification and Validation* (*V&V*) activities are performed on a software product before the delivery to the carmaker.
- The process of managing and reusing capitalized bugs. Indeed, bugs detected on previous projects and stored in the *problems' database* must be regularly reviewed in order to avoid similar bugs on new developments.
- The process of managing and reusing capitalized *test cases*. In automotive industry, the projects related to the same type of product and car platform of one carmaker have up to 70% of common functionalities (Johnson Controls source). Therefore, reusing *test cases* from one project to another must be done frequently.

For each of these parts, we make our analysis on two stages:

1. A snapshot of the current software practices in Johnson Controls (*process*, *tool*, *people*)
2. Analysis and diagnoses of these practices.

In the conclusion of this chapter, we summarize the performed diagnoses and we locate them within the Johnson Controls software organization.

II. Frame of the audit

Our approach to audit the practices currently used in Johnson Controls when verifying and validating software products can be divided into 7 activities:

- Analyze the documents delivered by the carmakers to their electronic suppliers. Their formats and their evolutions during the software development life cycle
- Analyze the main activities of an engineer when designing *test cases* for a software product. This analysis is performed with a multi point of view: process, tool, and people
- Audit engineers when designing *test cases*
- Intervention on the design of *test cases* for four software projects
- Interview managers on the expectations of the carmakers at each stage of the software development life cycle
- Interview all types of engineers that can be involved in a software project
- Analyze data on the *software testing* practices of carmakers

In the following, the results of the audit are presented.

III. The software projects in automotive industry

A. An incremental software development process

Definition 2.1: Project (Wikipedia – November 2008)

A project is a temporary endeavour undertaken to create a unique product or service. It can also comprise an ambitious plan to define and constrain a future by limiting it to set goals and parameters. The planning, execution and monitoring of major projects sometimes involves setting up a special temporary organization, consisting of a project team and one or more work teams.

A *project* consists of a set of coordinated and controlled activities organized to achieve an objective conforming to specific requirements. Presently, at Johnson Controls, a *product project* (hardware, software and mechanical skills) typically represents 24 months of development and involves around 25 engineers (Johnson Controls source). The five main stages of a project are illustrated in *Figure 2.1* and described in *Table 2.1*. Each stage is defined by a procedure that identifies the responsibility, deliverables (inputs, processes and outputs) and applicable references. The main focus of these stages is the *design* and *testing* of components and assemblies through product launch. It is a customizable process that is to be applied for all automotive products within Johnson Controls company.

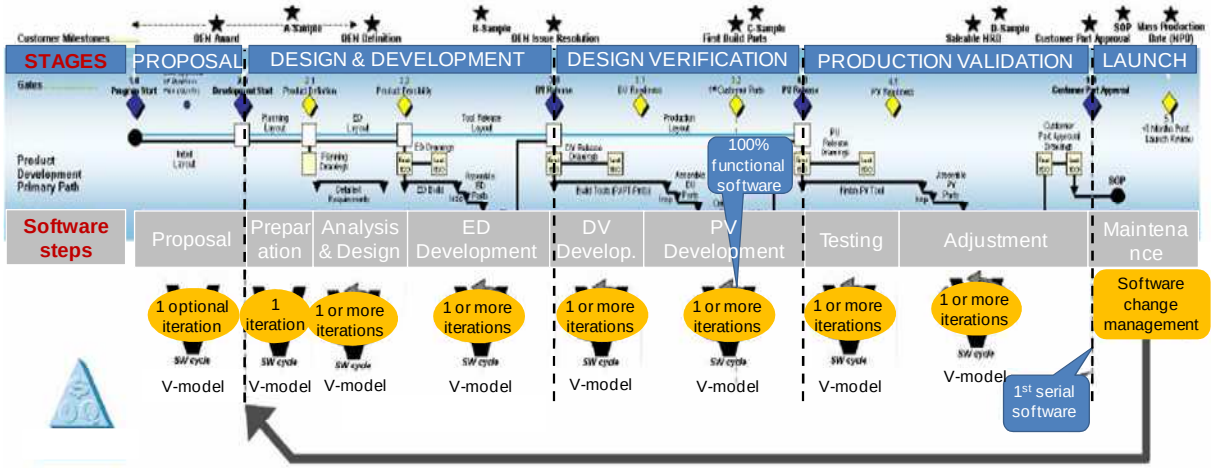


Figure 2.1 – Product development system at Johnson Controls - Some parts of this figure are voluntarily fuzzyfied for confidentiality reasons (Johnson Controls source)

PHASES	DESCRIPTION
Phase 1: Proposal	Responding to customer inquiries regarding new products. The requirements of the customer are identified and proposals are submitted for customer approval
Phase 2: Design & Development	Expanding upon the proposal through the establishment of the product definition to ensure product feasibility
Phase 3: Design Verification	Completing product design resulting in a detailed definition of both the product and process
Phase 4: Production Validation	Encompassing the activities required to ensure that the product meets all customer requirements when produced
Phase 5: Launch	Ensuring production preparation to ensure a smooth transition from production initiation to volume production

Table 2.1 – Description of the stages of a product project (Johnson Controls source)

The objective of the milestones of the product development system is to check the status and the progress of the program. *Software* skill has to take into account the *hardware* and *mechanical* releases availability and carmaker deliveries requirements of the product project to define its life cycle and its planning. In *Figure 2.1*, the steps of the *high level software life cycle* are mapped with the product development stages. A detailed description of the software steps is given in *Table 2.2*.

SOFTWARE STEPS	Description
Proposal	Analyze the customer need in order to estimate workload, schedule and resources needed to perform the software project
Preparation	Analyze in detail main customer requirements and deliver a 1st prototype (functionality level)
Analysis & Design	Analyze all customer requirements, define details of software architecture, deliver a functional release that contains main customer requirements and define the validation strategy
ED* Development	Complete the analysis of remaining requirements and deliver a partially functional release, with full functional but without diagnostic
DV* Development	Complete the global design, deliver a partially functional release without validation and manufacturing functionalities but with full functional and diagnostic and complete the test report for DV release
PV* Development	Complete the analysis of remaining requirements including validation and manufacturing requirements, specify and accept testing tools for production line
Testing	Perform a full testing of the software, improve the software reliability from customer tests feedback and complete the test report for PV Release
Adjustment	Take into account last minute customer changes and possible software problems and deliver an industrial release
Maintenance	Take into account eventual problems on serial products and customer changes

** ED: Engineering Development*
DV: Design Verification
PV: Product Validation

Table 2.2 – Description of the steps of the high level software life cycle (Mignen 2006a)

The software life cycle is initialized during the *Proposal* step and adjusted during the *Preparation* step according to carmaker deliveries planning and requirements prioritization (Cf. *Figure 2.1*). *100% functional software* means that all functionalities are implemented (carmaker and manufacturing requirements are implemented). If significant changes have to be implemented for a new release during the *Maintenance* step, a new software project is launched and the program remains in *stage #5*.

B. The elementary V-model of the software development process

Within each step of the software standard life cycle, engineering activities are performed according to the standard *V-model* of the software industry (Cf. *Figure 1.6*) and in an *incremental* way in order to take the carmaker constraints and requirements priorities into account. The number of incrementations per step is defined by the project and adjusted based on carmaker inputs and project constraints. Based on the *SPICE*[16] model, Johnson Controls has developed a process map implementing the conventional *V-model* (Cf. *Figure 2.2*).

[16] SPICE: Software Process Improvement and Capability dEtermination (http://www.sqi.gu.edu.au/spice/contents.html, Consulted on November 2008).

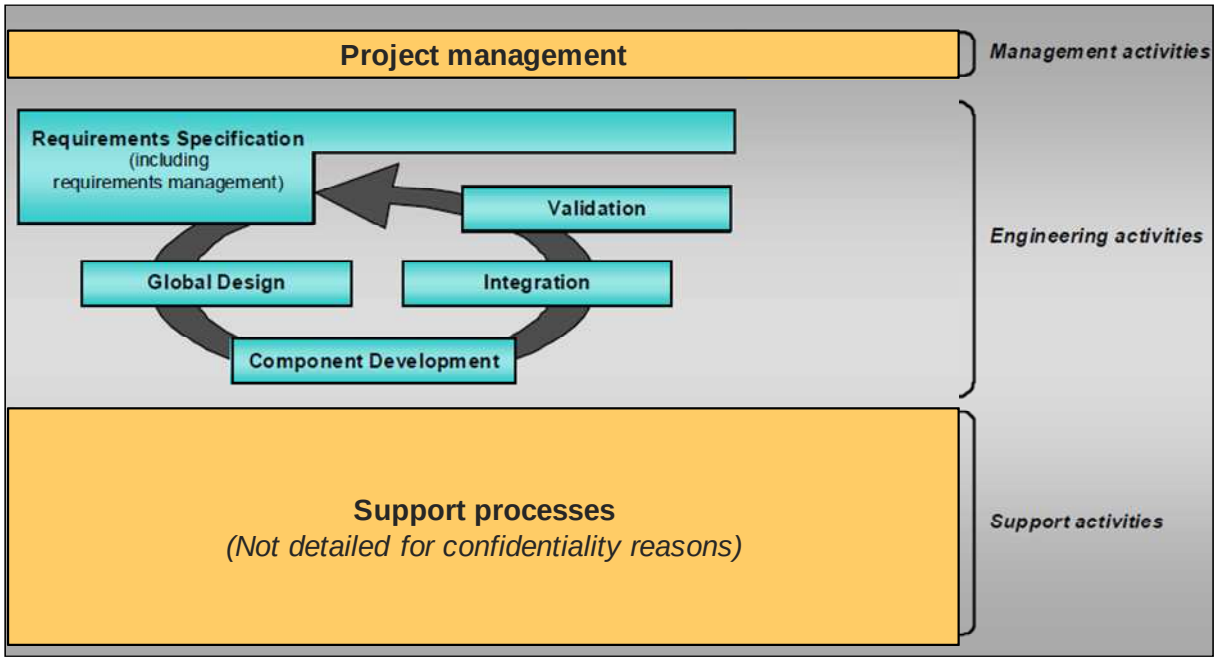


Figure 2.2 – Process map implementing the software V-model at Johnson Controls (Mignen 2006a)

Each process or group of processes (*Support processes*) of the process map is synthetically described in *Table 2.3*.

SOFTWARE PROCESS	Main activities
Project Management (PM)	Plan and monitor the software project Manage risks and documentation
Requirements Specification (RS)	Define, classify and prioritize requirements Establish traceability after validation
Global Design (GD)	Define general software architecture, components and interfaces Provide traceability and verify global design
Component Development (CD)	Develop detailed design, produce and verify components Develop, review and execute unit test procedures
Integration (INT)	Define software integration strategy Perform incremental integration and execute integration tests
Validation (VAL)	Develop software validation strategy Design, implement and perform validation procedure
Support processes *(Not detailed for confidentiality reasons)*	Control changes to configuration items Plan, track, verify and validate changes / defects Perform document and project reviews Perform Software quality and process audits

Table 2.3 – Description of the software processes within Johnson Controls (Mignen 2006a)

As illustrated in *Figure 2.1*, these software processes are carried out before each carmaker delivery of the software product. Despite the *V&V* of the software product and after each delivery to the carmaker, some bugs are detected by the carmaker. This could lead to the conclusion that carmakers have more efficient *testing* approaches than their suppliers. But, carmakers do not communicate on their practices and furthermore, they do not often transmit *test cases* to their suppliers. We analyze data on the *software testing* practices of carmakers and interview inner experts in touch with the carmakers. As a conclusion, this efficiency in *testing* software products can be related to many factors such as:

- The carmakers benefit from *real electronic platforms* where the supplier modules are installed and tested. In fact, the modules are tested in a simulated real environment with surrounding modules in a global system approach.
- The carmakers use their *experience feedback* of recurrent bugs to test a given module with the knowledge of bug probabilities and even *end-user behavior's profiles* to design the most relevant *test cases*.

Diagnosis 1

Verification and Validation practices and test cases are rarely shared between the carmakers and their electronic suppliers.

C. Functional organization of a software project

Besides the *project leader*, the *coordination* team and the *quality* team, one can identify two technical teams in a software project at Johnson Controls (Cf. *Figure 2.3*):

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

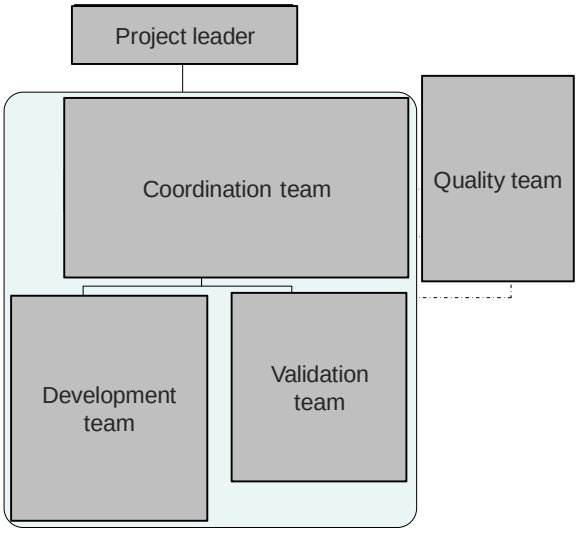


Figure 2.3 – Typical functional organization chart of one software project at Johnson Controls - Not detailed for confidentiality reasons (Mignen 2006b)

The coordination team is located in so called "front office" sites, close to the carmakers. Development and validation teams are in general located in *Low Cost Countries* (LCC). However, in some cases, they can be spread across several locations. Globally, we have *Software Developer* (*SD* or *developers*) who develop the software product, *Software Validation Engineer* (*SVE* or *validator*) who validate the software product before a carmaker delivery, *Software Coordinators* (*SC*) who are responsible for the assignment of the quality, schedule, cost goals and *quality engineers* who ensures that project quality commitments are respected. For confidentiality reasons, we are not allowed to give more details on the roles within a software project.

Ten years earlier, software *V&V* was covered only in *software engineering* courses. Nowadays, American but also European universities have responded to the importance of the *V&V* practices in industry with new independent courses and specialties in verifying and validating software products (Duernberger 1996). The goal of these courses is to prepare students for *software testing* management, testing considerations, designing *test cases*, and

applying various *testing* tools and methodologies. The fact remains that industrials give value to software *verification and validation* roles. In fact, software engineers often prefer *development* activities compared with *verification and validation* activities. This observation has been confirmed after interviewing software managers within Johnson Controls.

Diagnosis 2

Now, one cannot get a degree in software V&V. Software V&V is incorporated into the software engineering degree. Moreover, software engineers often prefer development activities compared with verification and validation activities.

IV. Management of the carmaker requirements related to software

Let us start by defining a common vocabulary on carmakers' requirements related to the software domain. In this dissertation, we consider the definition of *specification* and *requirement* proposed by *IEEE*.

Definition 2.2: Specification (IEEE Std. 610-1990)

A specification is a document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system, and, often, the procedures for determining whether these provisions have been satisfied.

Definition 2.3: Requirement (IEEE Std. 610-1990)

A requirement is a condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. There are two groups of requirement:

- Functional requirement: A requirement that specifies a function that a component or system must perform.

- Non functional requirement: A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability.

We also adopt some definition proposed by Johnson Controls software experts.

Definition 2.4: (Software) Functionality (Johnson Controls)

A functionality (called also client or software functionality) is described by some features that are described by some requirements. For instance, a speedometer is a functionality of a cluster

The term "Function" is not used in requirement management to avoid misunderstandings with the coding language.

Definition 2.5: Feature (Johnson Controls)

A feature is a "property" or "behavior" of a software. It describes the particularity of a device. Each feature is composed from one or more requirements. For instance, a "Speedometer" is a feature of a cluster. It can be broken down into 3 features:

- Speed display

- Speed computation

- Conversion from Km to miles

The breakdown granularity has to be adjusted according to the project needs.

Definition 2.6: Requirement (Johnson Controls)

A requirement is something to be done to design the device (it is required). For instance, the value of the speed shall be calculated using the X data. Each requirement must contain a subject / object and a predicate:

- Subject = system, user

- Verb / Predicate = action

The whole Requirement needs to define a result. A performance or measurable indication needs to be included

Example:

The display backlight has to be switched on in less than 1 second after ignition on.

object action result performance/measurable

There are 5 types of requirements:

- FCT: functional requirement (behavior of the software)

- CON: constraint requirement (reliability, safety, quality, process, rules, guidelines …)

- INT: interface requirement (specification of internal and external interfaces)

- DEV: development requirement for internal development support (interface, parameter …)

- MMI: man-machine interface requirement (Menus, buttons …)

Within the customer requirements, the software functional requirements account for more than 90% (Johnson Controls source). We also validate this proposal by analyzing the carmaker requirements for 5 different software projects (different products) in Johnson Controls. Therefore, through our research project, we focus on the software functional requirements and how one could verify the compliance of a software product with its functional requirements.

A. The carmakers specification of software functional requirements: diversity, typology, evolution

At the beginning of a project, automotive suppliers officially receive the electronic product requirements from the carmakers. In fact, there is no standardization between carmakers and electronics suppliers on the way requirements must be expressed. Based on this deliverable, suppliers analyze and identify skill requirements (*software*, *hardware* and *mechanical*). Afterwards, software requirements are sorted by the software department according to the typology proposed in *Definition 2.5*. In our research, we focus on the software functional requirements on which we identify two main characteristics:

1. Carmakers consider different standards to express the software functional requirements of a given electronic module. Some carmakers use semi-formal methods,

such as *Statechart* or *UML*[17] illustrated respectively in (Harel 1987) and (OMG 2005), others use *natural language*. Even, one carmaker could use two or more different standards according to each department policy. In 2007, we carried out a study on the evolution of the formalisms used by carmakers to specify software functional requirements. In *Chapter 4 – Section 4.D.1*, we do a survey on the formalisms used to specify the functional requirements of a software product (Dart 1987, Brinkkemper 1990). Three levels of formalism have been identified: *informal*, *semi-formal* and *formal*. The study was done on eight editions of carmaker requirements documents spanning from 1997 till 2006. We also considered three different carmakers: two Europeans and one Japanese but only one type of electronic product. The results of the study are illustrated in *Figure 2.4*. We underline the increase of formal methods based on the requirements simulation and the decrease of informal and semi-formal methods. Since this conclusion is fully true for the considered type of product, it is partially true for other types of product where *natural language* and *semi-formal* methods are still widely used. However and according to automotive experts, the trend is toward *formal* methods. Through this study, we also noted that software functional requirements are often expressed in many documents, emails, and even some phone calls.

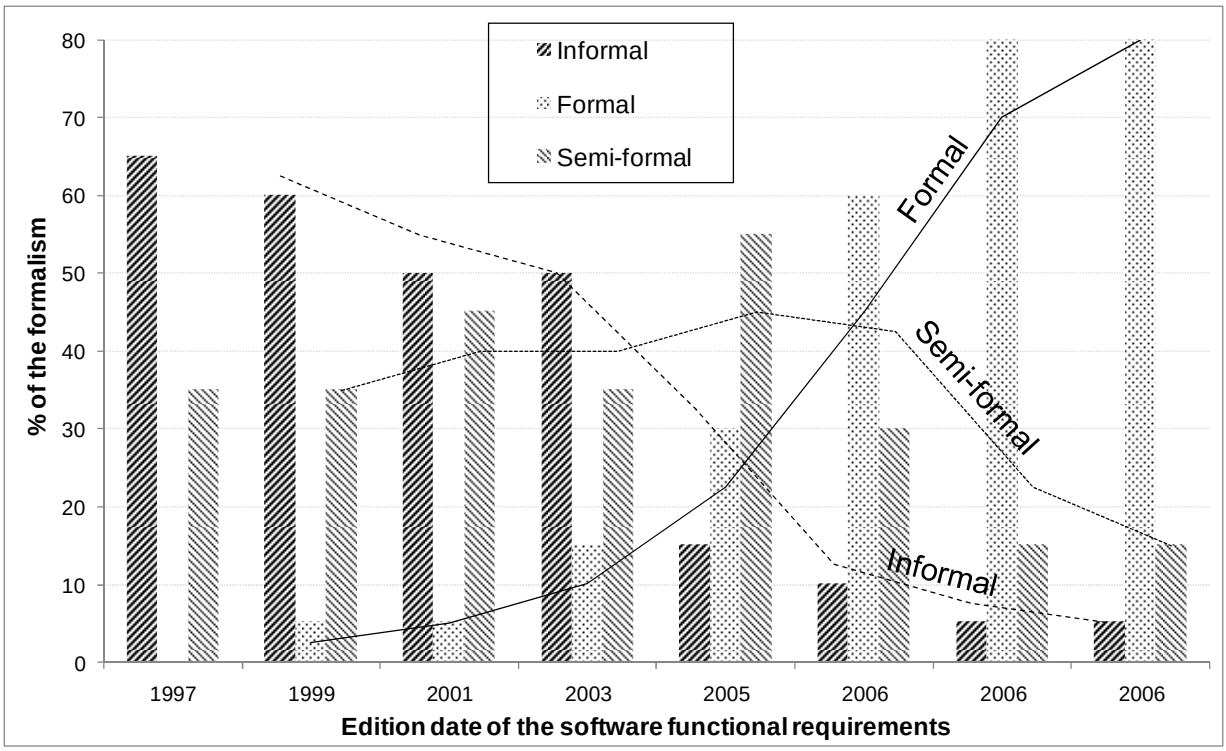


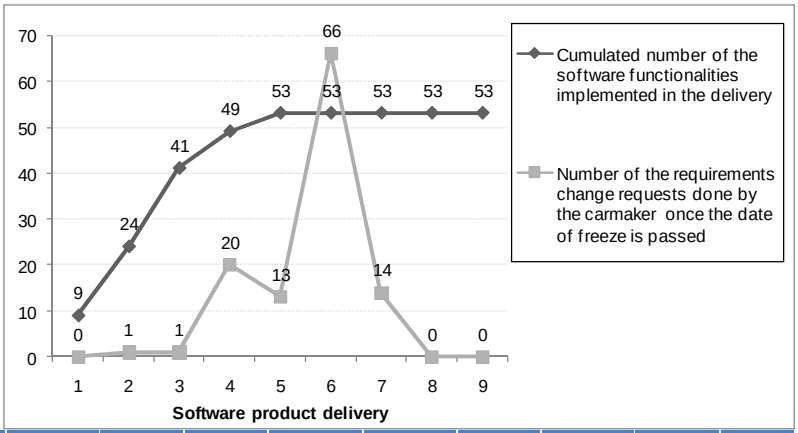
Figure 2.4 – Evolution of the formalisms used by carmakers to specify the functional requirements related software

[17] UML: Unified Modeling Language (http://www.uml.org/, consulted on November 2008).

Diagnosis 3

In automotive industry, semi-formal and formal methods are more and more used to specify software functional requirements. However, there is a lack of a standard* formalism *shared between carmakers and suppliers. In fact, for each project, the supplier has to adapt its processes to the* formalism *used by the carmaker.

2. Software functional requirements continuously evolve during development phases and also during *operational life* of the product. In 2007, we analyze the evolution on one project of the carmaker requirements related to software. The studied project started in 2005 and it is considered (by experts) as a typical project in automotive electronics industry. The *Figure 2.5* illustrates the results of the study. We note the growth number of changes asked by the carmaker after the date of software requirements freeze. We interview inner software experts and managers, often in contact with the carmakers, in order to understand this phenomenon. In fact, at the beginning of a project, the carmaker does not have a *100%-clear view* of what each functionality should perform. It is through the project and after each delivery that the expected behaviors become clearer. Moreover, suppliers are often more experimented in the development of automotive electronic products than some carmakers (system integrator). This leads to the fact that some carmakers lean on the suppliers by letting them identify inconsistencies and ambiguities in the product specifications.



Software product delivery	1	2	3	4	5	6	7	8	9
Date of deliveries	jan-05	mar-05	juin-05	sept-05	jan-06	juin-06	sept-06	jan-07	ma-07
Carmaker software requirements freeze	jul-04	oct-04	jan-05	juin-05	sept-05	-	-	-	-
Cumulated number of the software functionalities implemented in the delivery	**9**	**24**	**41**	**49**	**53**	**53**	**53**	**53**	**53**
Number of the requirements change requests done by the carmaker once the date of freeze is passed	**0**	**1**	**1**	**20**	**13**	**66**	**14**	**0**	**0**

Figure 2.5 – Growth of the number of changes asked by the carmaker all along a project

Diagnosis 4

Deadlines for carmaker requirements freeze are specified in the carmaker-supplier contract. Nevertheless, the carmaker's requirements evolve continuously along the software development life cycle without complying with these deadlines. Moreover, suppliers must react quickly by integrating (without regression) the changes in the product.

B. Commitment contract between carmakers and electronic suppliers

As noticed before, a *loop-type design process* is initiated between the carmaker and the supplier. About ten intermediary client deliveries are carried out. After each delivery, some "bugs" are detected by the carmaker and forwarded to the supplier who must react quickly and efficiently. Once an electronic module is launched on the market (e.g. integrated into a vehicle), an average of one "bug" per year is detected by the end-users, which may becomes dramatic for the electronic supplier in financial terms if the product has to be systematically changed.

We analyze typical contractual documents between carmakers and electronic suppliers. Moreover, we interview Johnson Controls managers in charge of establishing these contracts. In fact, in term of bugs' occurrence, two types of contract engage electronics suppliers with car manufacturers:

- *Explicit contract:* on launched electronic module, carmakers tolerate a certain number of defective products expressed in terms of *PPM* (Pieces Per Million). *PPM* includes all software bugs but also electronic, mechanical and production defects. For instance, in *Table 2.4*, when starting production (*SOP*) the carmaker tolerates X *PPM* on 0 km cars. It is logical that the required number of *PPM* on 0 km cars decreases (Y<X) 4 months after the production has started. The number of *PPM* is negotiated at the beginning of a project. The electronic supplier estimation of their capability in term of *PPM* number is mainly based on the *experience feedback* but also on the product complexity and novelty.

	SOP*	SOP + 4 months	SOP + 1 year
0 km	X ppm**	Y < X ppm	Z < Y ppm
3 months			
1 year			
3 years			

**SOP : Start Of Production*
***ppm: piece per million*

Table 2.4 – Explicit contract, in terms of bugs' occurrence, between a carmaker and an electronic supplier

- *Implicit contract:* implicit aspects are usually disclosed later in the development life cycle and are generally based on semantic problems. For instance, during the software development process and even if it was not stated in the contract, the carmaker expected that each intermediate delivery must be free of bugs. Many other examples can be cited. In fact, the requirements specifications delivered by the carmakers are usually and purposely unclear and incomplete in order to be able to add, modify or remove one or more requirements.

C. Sensitive criteria for carmakers

Carmakers are sensitive to different criteria depending on whether the project is in its *proposal*, *design* and *development* or *operational life* phase. In order to identify these criteria for each phase, we interview 3 *project leaders* for 3 projects in each of these phases.

1. Proposal phase

In proposal phase, carmakers choose their suppliers basically on *economic criteria*. An additional cost regarding other suppliers can exist but must be justified on quality and/or delay levels. Moreover, they strongly used their *experience feedback*s on other or previous projects with each supplier.

Presently, all the automotive electronic suppliers have almost the same knowledge and know how in the product design, development and maintenance. Therefore, competing with other suppliers on *technical criteria* remains very hard.

Finally, the *process improvement aspect* becomes a major quality criterion for the carmakers. For instance, the carmakers require now that their suppliers have reached a specific maturity level within the *SPICE* or *CMMI*[18] models. These two models are process improvement approaches that provide organizations with the essential elements of effective processes.

2. Product design and development phase

During the product design and development phase, intermediate deliveries of the product (including all or part of the product functionalities) are planned. Carmakers are sensitive to:

- *Time to delivery*: the supplier must respect the planning established at the beginning of the project.
- *Product quality*: the supplier must test the product and validate its conformance with the carmaker requirements before the delivery. *"Zero bug"* is required by the carmaker.
- *Additional cost*: sometimes, the supplier tries to invoice the modification or evolution requests asked by the carmaker.

3. Operational life phase

We identify three criteria to which carmakers are sensitive during the *operational life* of a product. We classify these criteria by priority order:

- *Regression risk* while modifying or correcting the product: the financial impact on the supplier can be severe especially when the car product lines are stopped because of its product. In order to better illustrate this issue, let us consider the following example excerpted from a real situation. Once, a carmaker required a modification on a product in *operational life* phase. The modification as it was expressed by the carmaker was to "remove" a piece of software code from the product in order to avoid the hacking of the product and therefore the stealing of the car. The supplier has implemented this modification by erasing the piece of code, full validated and delivered the new product version. The carmaker has also made a full validation of the new version of the product. Unfortunately, when starting the serial car production and when integrating the product in the cars, a bug related to this modification has occurred and thus blocked all the car production lines. A deep analysis of the bug has revealed that the removed piece of code must not be removed from the product but hidden. One more example on the implicit requirements of the carmakers since the carmaker declared that when he asked for "removing the code", he indirectly asked for "hide the code".

[18] CMMI: Capability Maturity Model® Integration (http://www.sei.cmu.edu/cmmi/, Consulted on November 2008).

- *Economic criteria* regarding the modification and correction costs.
- *Time to deliver* the new product version with the modifications and/or corrections requested.

D. Snapshot of the Requirements Specification process at Johnson Controls

1. Introduction

The purpose of the current *Requirements Specification* process is to ensure that all software requirements reflect allocation of carmaker and/or system requirement to software are identified, documented, maintained, committed and validated to serve as a basis for software design, implementation and validation. As a result of the *Requirement Specification* process:

- the software requirements to be allocated to the software components of the system and their interfaces are defined,
- software requirements are classified and analyzed for correctness and testability,
- the impact of software requirements on the operating environment is evaluated,
- prioritization for implementing the software requirements is defined,
- the software requirements are approved and updated as needed,
- consistency and bilateral traceability are established between system requirements and software requirements; and consistency and bilateral traceability are established between system architectural design and software requirements,
- and the software requirements are baselined and communicated to all concerned people.

2. Interfaces with other software processes

The current *Requirements Specification* process is considered as the main important process within the software processes. In fact and as shown in *Figure 2.6*, this process strongly interacts with all other processes. Especially, it delivers the software requirements to allocate them to components (*Global Design*), to develop these components (*Component Development*) and to design associated *test cases* (*Validation*).

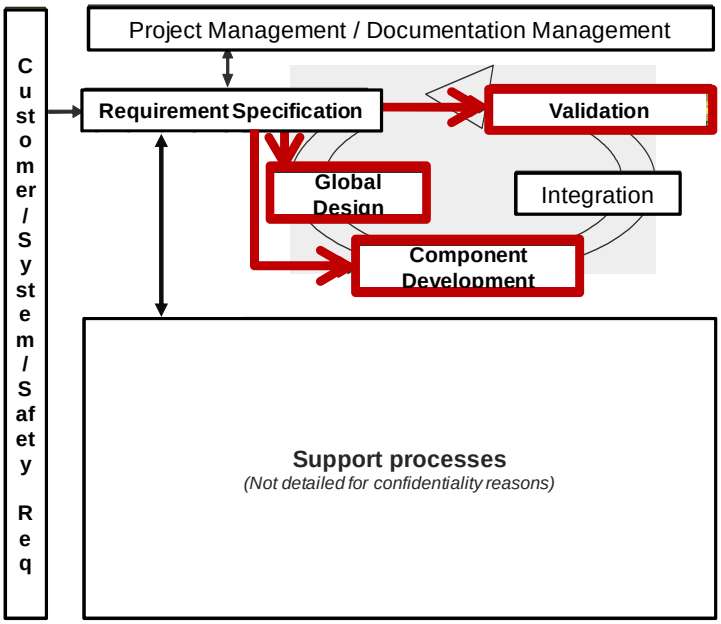


Figure 2.6 – Interaction of the Requirements Specification process with the other software processes (Mignen 2008)

3. Process flow

Within each step of the software standard life cycle (Cf. *Figure 2.1*), engineering activities are performed according to the process map illustrated in *Figure 2.2* and in an *incremental* way in order to take the carmaker constraints and requirements priorities into account. After analyzing internal documents related to the definition of the *Requirements Specification* process, we identify 6 main activities for the management of software requirements. In fact, for each design iteration, the *Requirements Specification* process flow follows the series of activities defined in *Table 2.5*.

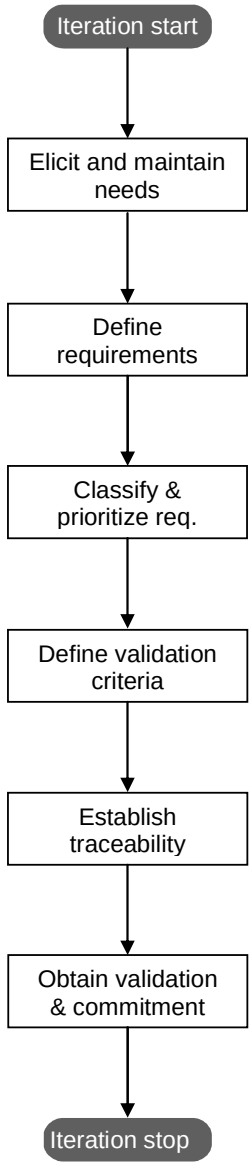
Process	Comments
	Carmaker documents of requirements are identified The activity continues until all requirements in the scope of iteration are defined). A list is used to track unclear needs until clarification with the carmaker All Software requirements are classified and prioritized Special conditions for validating requirements are defined Traceability is established with carmaker and/or system requirements All the requirements are reviewed by internal and external concerned people. Commitment on software requirements is done with carmaker and system

Table 2.5 – Process flow of the Requirements Specification process

a. Elicit and maintain needs

This activity aims to establish and maintain carmaker and system needs and expectation that will serve as a basis for specifying requirements allocated to software. The features required by the carmaker are identified and peer projects for these features are identified for use of lessons learned.

b. Define requirements

For each feature, software requirements are specified or updated using the *Software Requirement Specification* (SRS) model (see next section for the principles of this model).

Requirement Management tools such as *Reqtify*[19] or *Doors*[20] can be used for managing and storing requirements. All items that need to be clarified are filled in a *CLarification Request* list (CLR) and discussed with concerned people.

c. Classify and prioritize requirements

For each software requirement identified in the *SRS*, one has to define:

- the type of the requirement (*FCT*, *CON*, *INT*, *DEV*, *MMI*),
- the status (*new*, *accepted*, *confirmed*, *dropped*),
- the priority (*high*, *normal*, *low*),
- the associated feature,
- and the required verification and validation technique (*code analysis*, *code review*, *unit test*, *integration test*, *validation test*).

Other criteria can be added such as safety, revision and so on. A *Requirement Management* tool can be used for this classification.

d. Define validation criteria

In order to support the validation team and facilitate *test case* definition, *validation criteria* need to be specified. These criteria consist of defining when the test can be considered as passed correctly (*test case* results acceptance). A *validation criterion* can be applicable on several requirements or group of requirements. *Validation criteria* can be based on a lesson learned. By default, standard *validation criteria* are the successfully passing of *test cases* (in this case, it is not necessary to define specific validation criteria). The *validation criteria* shall define special conditions for validating some requirements if their validation deviates from the standard use of tests cases. For instance:

- Specific criteria to validate (respect of standard, performance criteria, different situations to validate …).
- Condition for validating the requirement (normal and specific conditions for the test, stress situations, tools, or negative tests).
- Criteria defining when validation tests can be considered as passed correctly (including thresholds of performance deviation).

e. Establish traceability

Purpose of this activity is to establish and maintain upward bilateral traceability between user requirements (carmaker requirements, system requirements) allocated to software and software requirements in order to verify that all carmaker and system requirements that have been allocated to software are taken into account in the *SRS*. The result of this activity is a matrix called *traceability matrix*.

f. Obtain requirements validation and commitments

Each time a step of the *SRS* elaboration is achieved in order to start a part of software development, the version of *SRS* is reviewed to make sure the understanding and commitment

[19] http://www.geensys.com/?Outils/Reqtify (Consulted on November 2008).
[20] http://www.telelogic.com/Products/doors/doors/index.cfm (Consulted on November 2008).

by the implementation and validation teams. Once the version of *SRS* is released, it is base lined and serves as a basis for implementation and validation activities.

E. The Software Requirement Specification model currently used in Johnson Controls

Automotive electronics suppliers like many *software engineering* organizations adopt the *Software Requirement Specification* (*SRS*) model to express the various expectations of a software product. A *SRS* model is a comprehensive description of the intended purpose and environment for software under development. The *SRS* fully describes what the software will do and how it will be expected to perform. A good *SRS* defines how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations. Parameters such as operating speed, response time, availability, maintainability, security and speed of recovery from adverse events are evaluated. Methods of defining an *SRS* are described by *IEEE* (IEEE Std. 830-1998).

Johnson Controls has adapted the *SRS* model to its organization, needs and types of products. In *Figure 2.7*, the data model of the *SRS* currently used is described. The *SRS* document serves as a basis for software design and validation plan.

The engineer responsible of managing the carmaker requirements shall update the *SRS* document according to the *CLarification Request* (*CLR*) answers and input specification updates as well as change requests. Once the SRS document has been released, the *CLR* shall be used to ask question or request or give clarification on *SRS* (the *CLR* is used with the carmaker and internally in the team). The *SRS* shall be updated with the content of the *CLR*. The change of the specification after the *specification freeze* milestone shall be an exception. The *specification freeze* corresponds to the date where in theory no specification change is allowed.

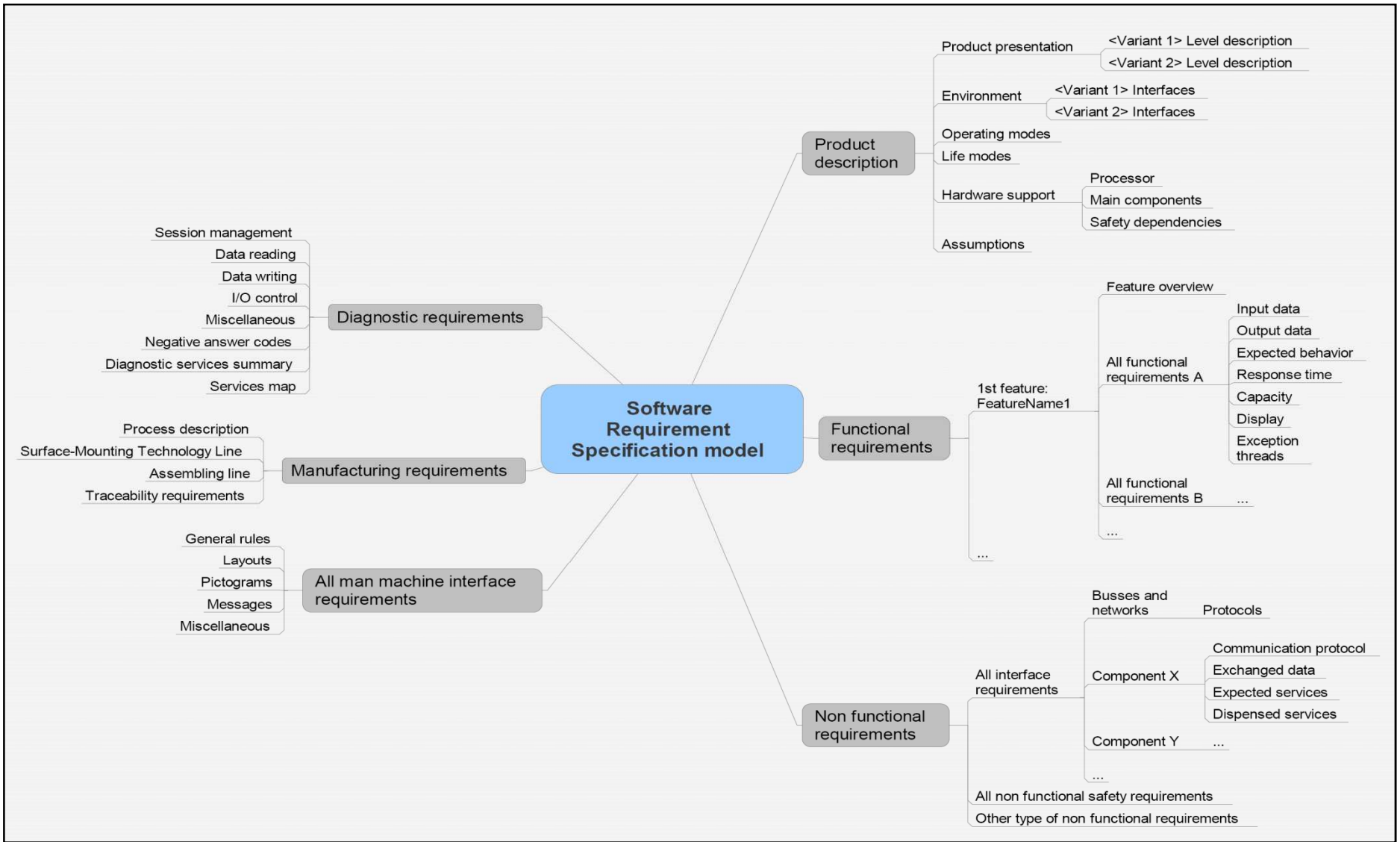


Figure 2.7 – A UML-like data model of the SRS currently used in Johnson Controls

Diagnosis 5

The Software Requirement Specification (SRS) document is often a large document (about hundreds of pages), difficult to manage, incomplete and not regularly updated.

Diagnosis 6

Sometimes, the SRS document is the official and contractual document between carmaker and supplier. It is also the main document used by the development and V&V teams in their activities. It has a standard structure but there are no standards to specify carmakers' requirements (more especially functional requirements).

F. Quality criteria of a requirement

In order to support reviews and improve the quality of software requirements the following criteria are defined by Johnson Controls software experts. They must be checked during reviews and respected during the set up of the *SRS* document. The review of the *SRS* document is supported by a *Checklist*. It consists of verifying explicitly the criteria listed hereafter.

1. General criteria

- Use a simple language style to define the requirements.
- Short sentences, not interlocked, need.
- Use present tense.
- Use simple, clear, vocabularies, introduced terms wherever possible
- No multiple definitions.
- Reference what is defined correctly by existing specifications (do not copy).
- Apply *SRS* template and *Requirement Management* tool template (Styles, fonts, types …).
- Do not specify design or implementation. Describe what to do not how to do it.
- In case of complicated conditions use state-, sequence- or flow chart to gain clarity and remove ambiguity.
- Describe the interface of the device with the environment not the interface of components within the device. In case of complicated conditions use state-, sequence- or flow chart to gain clarity.

2. Detailed Quality Criteria

Understandable

- The text is easy to understand and the requirement is clear for the reader.
- Needless or confusing words are not used.

Complete

- The Feature (group of requirements) will contain all the information needed to implement and test the requirement. No information needed to implement the feature will be missing.

Consistent

- There will be no contradictions within a single requirement or between two requirements.
- Usage of the same terms as used in other definitions.

Necessary

- The definition is needed for the realization of the feature. Removing the requirement will change the behavior related to the feature and/or will render the feature incomplete.
- Unambiguous.
- The definition is clear and has only one single interpretation.

Atomic

- A further breakdown of the definition is not possible.

Feasible

- It is possible to implement, fulfill and test the definition (time, budget, *know-how*?).

Maintainable

- There will be no redundancy. Redundancy is allowed only when removing the redundant phrase, sentence or requirement will cause ambiguity.

Testable

- It is possible to test the definition / to develop a test case.

V. Software verification and validation activities in automotive industry

A. Overview on software verification and validation techniques at Johnson Controls

As we shown in *Section 3*, within each step of the software standard life cycle, engineering activities are performed in an iterative way according to the standard *V-model* of the software industry. The *Component Development* process is the process where the *source code* is developed. Following the *code implementation* and before any carmaker delivery, a series of verification and validation techniques have to be applied on the *source code* in order to check its correctness and its compliance with the carmaker expectations (software requirement specification). At Johnson Controls, we identify 3 *software inspection techniques* (*Code review*, *static analysis*, *dynamic analysis*) and 3 *software test techniques* (*unit*, *integration* and *validation test*). In this dissertation, we adopt the definitions proposed by *IEEE* for each of these techniques.

Definition 2.7: Software code review (IEEE Std. 610 1990)

The software code review is a visual examination of a software work product to detect defects, e.g. violations of development standards and non-conformance to higher level documentation.

Definition 2.8: Static code analysis (IEEE Std. 610 1990)

The static code analysis is an analysis of source code without execution of that software. A static code analyzer is tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies.

Definition 2.9: Dynamic code analysis (IEEE Std. 610 1990)

The dynamic code analysis is the process of evaluating behavior, e.g. memory performance, CPU usage, of a system or component during execution. The dynamic analysis tool provides run-time information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic and to monitor the allocation, use and de-allocation of memory and to flag memory leaks.

Definition 2.10: Software Unit, Integration and Validation test (IEEE Std. 610 1990)

Unit test is the test of individual software components.

Integration test is the test performed to expose defects in the interfaces and interaction between integrated components.

Validation test is the process of testing an integrated software product to verify that it meets specified requirements.

The interactions between these testing techniques are illustrated in Figure 2.8.

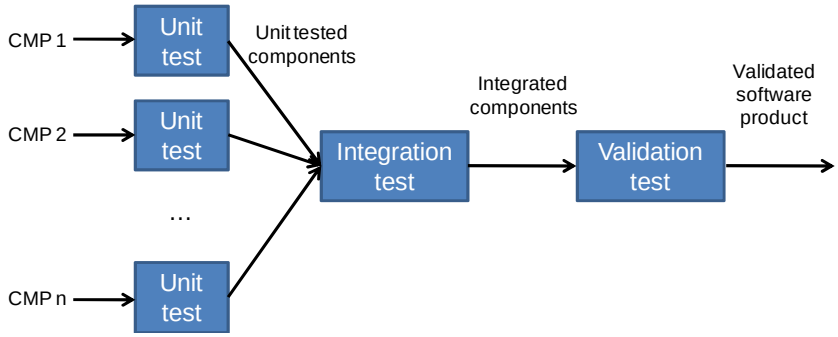


Figure 2.8 – Interactions between unit, integration and validation tests

The location of these techniques within the Johnson Controls software process map is shown in *Figure 2.9.*

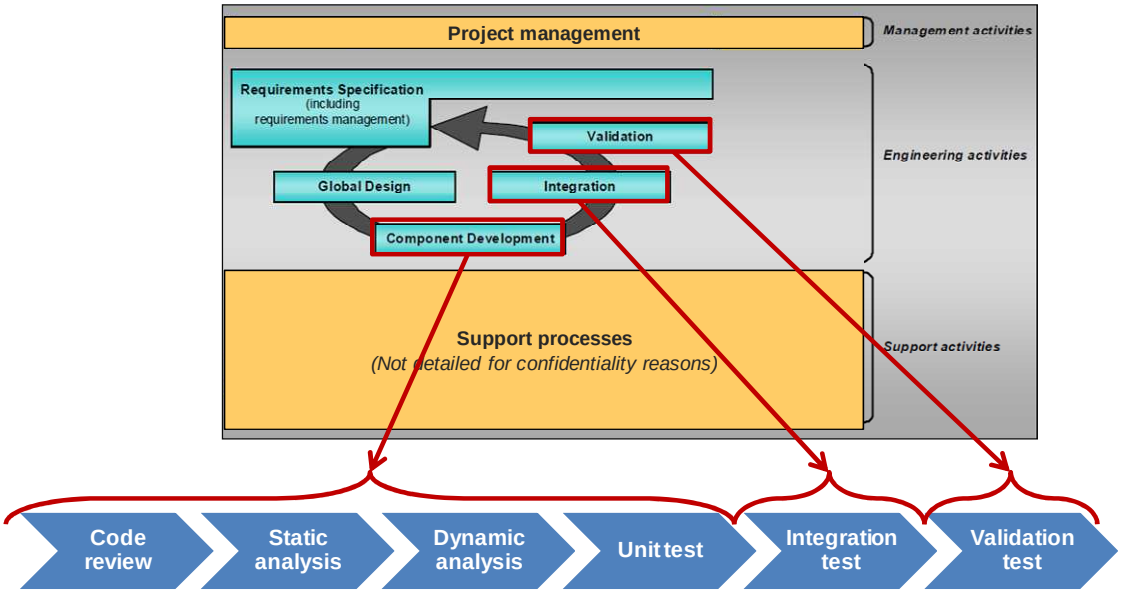


Figure 2.9 – Software verification and validation techniques within the Johnson Controls process map

Each of these techniques must catch different classes of bugs at different points in the development cycle.

B. Software V&V techniques in Component Development process

The purpose of the *Component Development* process, as it is defined in Johnson Controls, is to produce executable software components that properly reflect the global design and software requirements. Moreover, a strategy has been defined in order to verify and validate each software component, after it is produced. This strategy is applicable for all the software components in the projects. For each component in the scope of the *V-model*, a *Component Development* process is used. After analyzing internal documents related to the definition of the *Component Development* process, we identify 3 main activities when developing a new software component (Cf. *Table 2.6).*

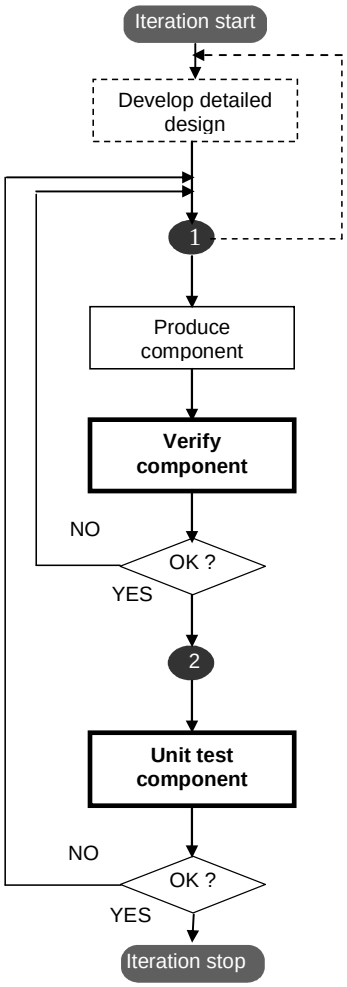
Process	Comments
 *All these activities are performed by Software Developers (SD)*	Bugs detected on component can lead to correct detailed design and/or code The "Verify component" activity consists to: - Review the code - Analyze statically the code - Analyze dynamically the code "Unit test" of the software component If the results of the unit test done on the component are OK, the component is promoted to integration

Table 2.6 – Process flow of the Component Development process

First activity: *Produce component*

Produce component activity aims to produce components (*source code*, code generator data …) and/or to fix bug(s) detected in next steps of development (*verification* and *unit test* activities). This *coding* activity is based on global and detailed design and implements the *component design* made previously. *Coding* has to be done in accordance with defined *coding standards*, *rules* and *guidelines* and with embedded system constraints (*memory size*, *hardware dependency* …).

Second activity: *Verify component*

In this activity, three *software inspection techniques* (static *V&V* techniques) are performed:

- *Code review*, based on the *Review & Verification* process.
- *Static analysis* based on a commercial tool (*QAC*[21]).
- *Dynamic analysis* based on a commercial tool (*PolySpace*[22]).

The v*erify component* activity follows a process flow described in *Table 2.7.*

[21] http://www.programmingresearch.com/QAC_MAIN.html (Consulted on November 2008).
[22] http://www.mathworks.com/products/polyspace/index.html (Consulted on November 2008).

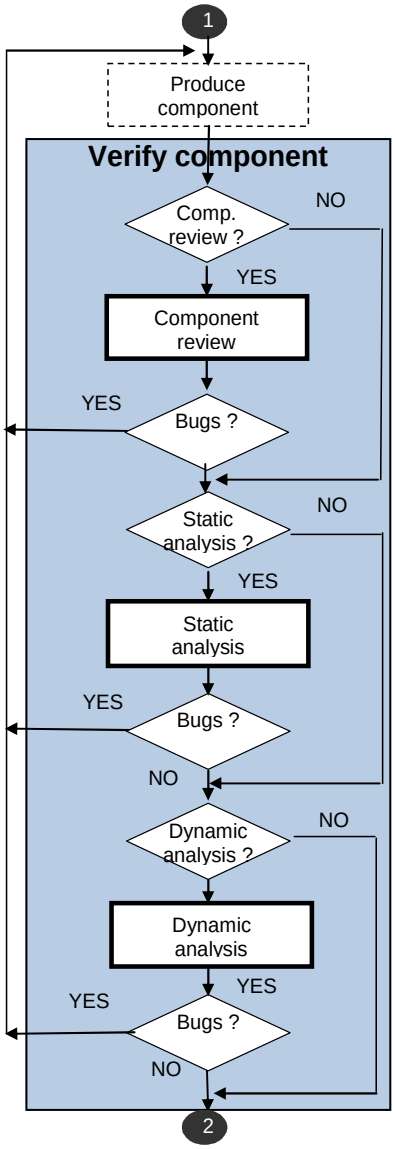
Process	Comments
	Do we need to review the software component? **Code review** is performed in filling the nonconformities in an *Issue Log*. Do we need to statically analyze the software component? **Static analysis** is performed with a *static analysis tool*: *QAC*. Do we need to dynamically analyze the software component? **Dynamic analysis** is performed with a *dynamic analysis tool*: *PolySpace*. It is possible to perform dynamic analysis only for the whole software

Table 2.7 – Process flow of the verification of a software component

Code reviews are mainly intended for checking respect of coding standard and rules / quality of comments in a software component. *Static analysis* is intended to check the compliance of the *source code* with the international automotive software coding rules (*MISRA-C*[23]). *Dynamic analysis* is intended for detecting problems, with a *dynamic* point of view, early in the life cycle. This type of problems could be detected by *testing* activities. Static and dynamic analysis can be done on the whole software (and not only for each component).

Third activity: *Unit test component*

This activity consists of *testing* unitarily each software component. In other words, this software test technique intends to verify the correctness of all *functions* / *conditions* / *decisions* / *component inputs* and *outputs* / *boundaries* and *limits* in a component *source code*.

[23] MISRA-C is a software development standard for the C programming language developed by MISRA (Motor Industry Software Reliability Association, http://www.misra.org.uk/, Consulted on November 2008).

To summarize, the software *Component Development* process within Johnson Controls performs four *V&V* techniques on each software component: three *inspection techniques* (*code review*, *static* and *dynamic analysis*) and one test technique (*unit test*). In the following, we develop each of these techniques as it is practiced at Johnson Controls.

1. Technique 1: Review of a software code

According to the Johnson Controls process, the purpose of the software component review is to:

- Check whether the *source code* of the produced component respects the design or not.
- Check whether all remaining problems after automatic static/dynamic analysis are properly justified in the *source code*.
- Verify the quality of the comments written in the *source code*.
- Verify the traceability of the component to software requirements.
- Check whether the *source code* respects the coding guidelines, especially those rules that cannot be tested automatically by a tool. Software engineers have to check the compliance of the inspected code to the rules and recommendations. The modifications of these rules can be made by a committee, whose members are appointed by the *Software Engineering Process Group*[24] (SEPG) of the company. The committee includes representatives of all Johnson Controls sites on which this document is deployed. In fact, there is a document which defines coding rules and recommendations for using the *C language*[25] in the development of embedded software for the automotive industry. The document is organized as a collection of rules and recommendations illustrated in *Figure 2.10*:
    - A *rule* is a prescription that has mandatory character. It must be always followed.
    - A *recommendation* is a prescription that has advisory character. It must be followed as much as possible.

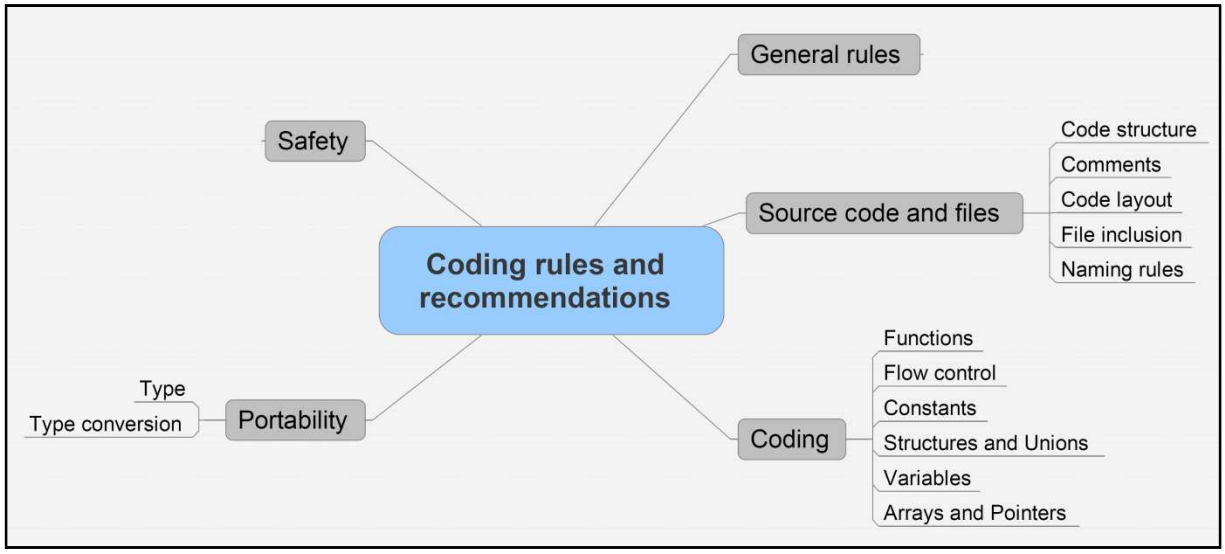


Figure 2.10 – Classification of programming rules and recommendations

[24] The SEPG is a group of software experts.
[25] Computer language.

An excerpt of these rules and recommendations is illustrated in *Appendix A*. Absolute and unconditional adherence to these rules and recommendations may not be possible at all times. However, it must be noted that some general rules is of an outstanding importance – under no circumstances shall this rule be broken.

In order to perform the *code review* of a software component, a group of Johnson Controls software engineers must read the *source code* and simultaneously fill an *Issue Log* with the identified issues. An *Issue Log* capitalizes the reviewers' names, the date of reviewing, the name of the reviewed component, the review load, the reviewed number of *Lines Of Code* (LOC) and a list of identified issues. For each issue, reviewers give an *ID*, the number of the line where the issue was found, a description of the issue and the status of the decision and correction.

In 2006, we carried out a study within Johnson Controls on four projects related to two different electronic products. The aim of this study was to audit the practical implementation of the *code review* technique. The main result of this study is that number of code reviewers is not aware of the coding rules and recommendations. Moreover, we note that *code review* is not systematically performed on each new software component. These conclusions were also validated by inner software experts and managers. In fact, in automotive industry, *software testing* is considered to be the main *V&V* activity which has to detect all the bugs. Unfortunately and as shown in *Figure 1.11*, detecting bugs later in the process costs more than detecting them as soon as they are introduced.

Diagnosis 7

Sometimes, the review of software code is badly done or even ignored. In fact, number of code reviewers is not aware of the coding rules and recommendations to be checked. Moreover, the code review is not systematically performed on each new software component. In consequence, the code review does not often detect all the bugs that must be detected through this activity.

2. Technique 2: Automatic static analysis of a software code

According to the Johnson Controls process, the goals of the static analysis are to:

- Improve the quality of the *source code*.
- Improve the robustness of the software.
- Make the *source code* as much as possible portable.
- Be compliant with *MISRA-C*.

There are two phases of this analysis, executed separately:

- During the verification activity of a single component. This must be done by the developers who create/modify the components.
- Overall project static analysis. Done after the integration of all the components. This task could be delegated to an experienced developer.

The *static analysis* is performed automatically using a computer tool such as *QAC*, the most used in automotive industry. It is recommended to apply this *V&V* technique in the beginning of the project in order to be able to detect and fix the issues as early as possible. The criterion to stop the *static analysis* of a *source code* is that all *QAC errors* and *warnings* are either fixed or justified. A screenshot of the *QAC* tool is illustrated in *Appendix A*.

3. Technique 3: Automatic dynamic analysis of a software code

A *dynamic analysis* of the software code is performed in order to find and fix as early as possible the software bugs that could possibly occur one executing the software product and cannot be detected by *code reviews* and *static analysis*. The *dynamic analysis* is performed automatically thanks to a commercial tool *PolySpace*. The intention is to get a clear view about the dynamic behavior of the software early in the process. The earlier detection of problems reduces the risk of having serious bugs at late project phases. However, the project has to plan enough time (depending on the project's size and warnings reported) for warning analysis. The criterion to stop the dynamic analysis of a *source code* is that all *Polyspace errors* and *warnings* are fixed or justified. A screenshot of the *Polyspace* tool is illustrated in *Appendix A.*

4. Technique 4: Unit test of a software component

The *unit test* of a software component is described by the process flow of *Table 2.8*. After analyzing many Johnson Controls documents related to the *unit test* activity and interviewing inner engineers practicing *unit test*, we identify three main activities: design the *test cases*, review the *test cases* and finally execute the *test cases* and analyze the results. We illustrate below the definition of a *test case* as it is adopted in Johnson Controls.

Definition 2.11: Test Case, Test Step and Operation (Johnson Controls)

*Let us consider a functionality with two input signals: I1(with domain D(I1)={0,1}) and I2 (D(I2)={1,2,3}) and three output signals: O1(D(O1)={0,7,14}), O2 (D(O2)={1,2,3}) and O3 (D(O3)={0,1}). We first call "**Operation**", the fact that an input signal is set to a value. For example, I2=3 is an operation. A "**Test Step**" is composed from an operation, an inter-operation time and expected results on the output signals. A "**Test Case**" is a succession of "test steps".*

An excerpt from a test case designed is given in the Figure 2.11:

- In test step 96, test engineers wait for 500 ms without carrying out any actions on the product and check that the outputs of the product haven't changed.

*- In test step 97, test engineers activate a switch (Input_1=1), wait for 200 ms and **check that the concerned outputs** are activated according to the expected behavior.*

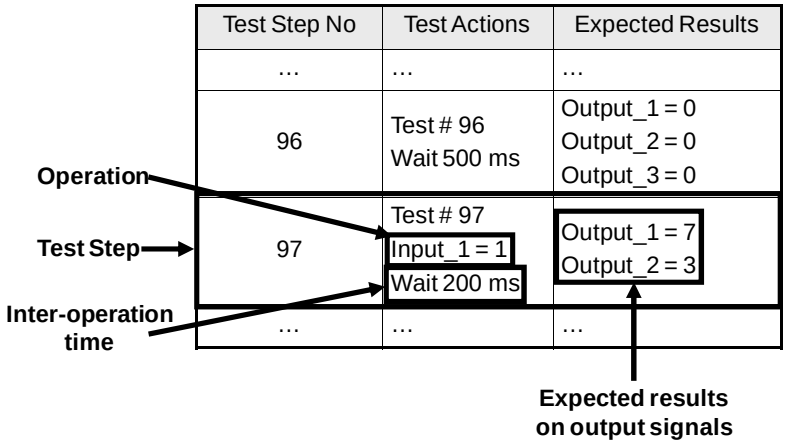


Figure 2.11 – An excerpt from a test case (two test steps) as designed by Johnson Controls tester engineers

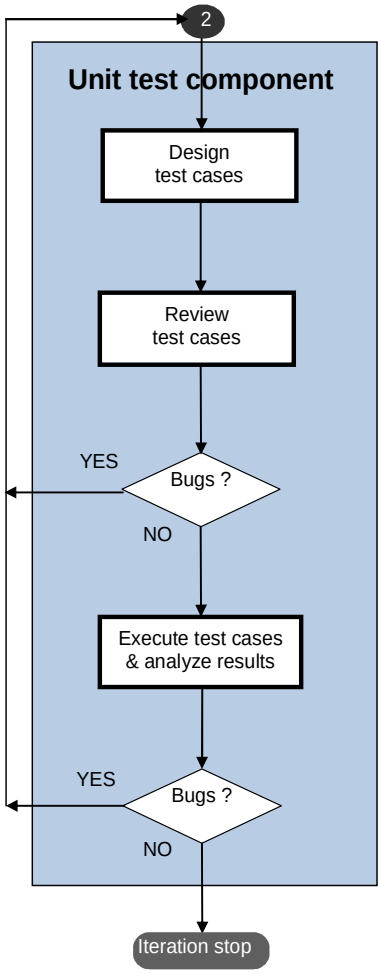
Process	Comments
	*Test cases* for *unit test* are manually designed. The quality of these test cases is based on the experience of the developer. Designed *test cases* are reviewed Designed *test cases* are executed on the software component under test If the results of the *unit test* done on the component are OK, the component is promoted to integration

Table 2.8 – Process flow of the unit test of a software component

After developing a software component, *software developers* *design manually test cases* for the *unit test* of this component. The main purpose of the *unit test* is to cover at 100% the *source code*. It is the main criterion to stop *testing* unitarily a software component. The principles of *code coverage* are developed in *Section 6.A*. In fact, developers analyze the structure of the software component and design *test cases* that must cover all the source code of the component under test. The *test case design process* presently used by the engineers at Johnson Controls is deeply described in *Section 6*. It is important to note that a software component is about 2000 *LOC* (without *blanks* and *comments*), a reasonable number of *LOC* to be analyzed (according to experts). The technique of designing *test cases* while having access to the code of the software under test is called *structural* or *white-box* or *program-based test*. A survey on *software testing* techniques (Bernot 1991, Beizer 1995) is provided in *Chapter 4 – Section 3.B.*

Presently, in Johnson Controls, the *unit test* is not responsible to verify the compliance of a software component with the carmaker requirements. In fact, one software component can be tested unitarily (100% of *code coverage*) without fulfilling the behavior required by the carmaker.

Diagnosis 8

According to the "To Be" process, the unit test of a software component must ensure a 100% source code coverage. However, this V&V technique is not responsible to verify the compliance of the component's behavior with the carmaker requirements. One software component can be tested unitarily (100% of code coverage) without fulfilling the behavior required by the customer.

The language used to design *test cases* for the *unit test* of a software component is the *C language*. A standard *unit test* structure providing predefined *C functions* in order to help the test engineer writing *test cases* is developed in *Appendix B*.

The *designed test cases are reviewed* in order to check:

- the relevance of designed *test cases* in regard with the tests objectives,
- the reached *code coverage*,
- and the usefulness of the dummy *test cases* dedicated only to reach the expected *code coverage*.

Finally, all *the designed test cases are executed* on the software component under test. All the dependencies and connections between the components are simulated to isolate the tested component from the project. Test results are analyzed to decide if *Component Development* activities have to be restarted, in case of failed tests. The *code coverage* is recorded and used as criteria to stop the design of *test cases*. The *unit test execution platform* is developed in *Appendix C*.

C. Software verification and validation techniques in Integration process

Once a set of components are produced and verified through the *Component Development* process, they are integrated together and an *integration test* (*V&V* technique) is performed on the overall software product. In Johnson Controls, the purpose of the *Integration* process is to assemble the software product from the software components, ensure that the software product, as integrated, functions properly, and deliver the tested software product to *Validation* process.

1. Technique 5: Integration test of a set of software components

The purposes of the *integration test* are to ensure that *global design* requirements work as expected and the quality of the software allows the execution of the *Validation* process. To do this, three steps have been defined by Johnson Controls software experts:

First step: *Interface review*

The engineer responsible of integrating the software components reviews each component in order to verify the conformity of interfaces to predefined architecture. According to the relevance of the review, she/he can decide to setup additional verification by adding test steps in the *functional* and/or *change test*.

Second step: *Change test*

Change test cases are defined and executed only once, when change is integrated. The first objective of *change test* is to verify the good implementation of the requirements involved in the change. The second objective of *change test* is to verify the good implementation of the architecture expectations involved in the change. Change can be either evolution implementation or bug fixing. For bug fixing, *change test cases* have to check that bug is not

reproduced when sequence used for bug detection is re-executed. For evolution, *change test cases* have to check main impacts of the change on software requirements.

Third step: *Functional test*

Functional test cases improve confidence on the verification. The purpose of this test is to detect *regression* on a new integration. It has to check a limited list of software requirements. In this perspective, a set of *test cases* per software functionality has to be defined.

In conclusion, the criteria to stop the *integration test* of a software product are rather subjective. Indeed, the *engineer* must verify (according to her/his point of view) that a change does not impact the whole software product and that the main requirements of each functionality are satisfied.

D. Software verification and validation techniques in Validation process

Once a set of components are integrated together, a *validation test* (*V&V* technique) is performed on the overall software product.

1. Technique 6: Validation test of a set of software components

In Johnson Controls, the purpose of the *validation test* is to confirm that the integrated software meets the carmaker requirements related to software. The *validation test* is activated at each new iteration when the software product has been successfully integrated. Now, the *validation test* is completed when:

- The planned *validation procedure* is executed. A *validation procedure* is composed from a set of *test cases*.
- All of the requirements (defined in the *SRS*) in the scope of the delivery are covered.
- And if any testable requirement could not be covered, the reason about it must be justified. The *requirement coverage*, as it is currently practiced in Johnson Controls, is developed in *Section 6.B*.

For each iteration, the *validation test* of a software product is described by the process flow of *Table 2.9*. After analyzing many Johnson Controls documents related to the *validation test* activity and interviewing inner engineers practicing *validation test*, we identify two main stages: *Preparation of Validation* and *Execution of Validation*.

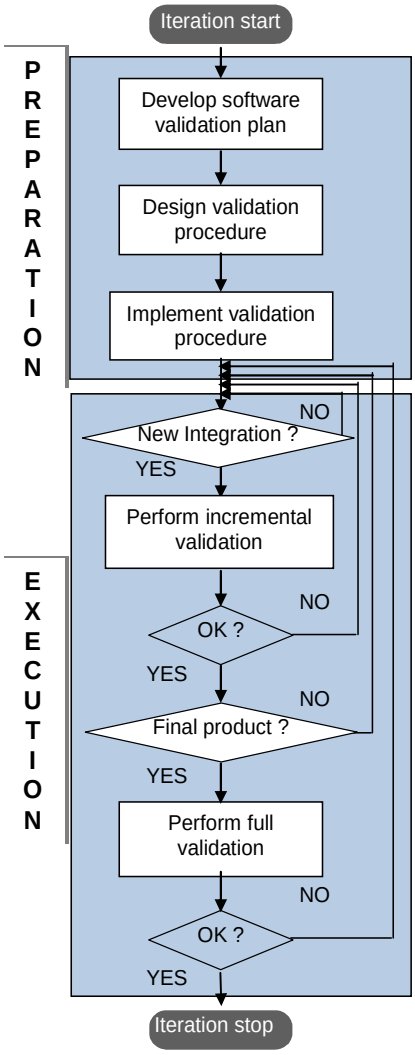
Process	Comments
	The strategy for *Software Validation* is defined *Test cases* are identified *Test cases are designed* Is there any new software integration? Execute the selected *test cases* on the new software integration Is it the final software product? Execute the whole *test cases* on the final software product.

Table 2.9 – Process flow of the validation test of a software product

a. Preparation of validation

Develop the software validation plan

The *Software Validation Plan* (*SVP*) describes the *validation strategy* for a project. It serves as a guideline for executing validation tasks by project and by scope of delivery. The strategy of validation may be adjusted for each iteration (according to the delivery content). The *SVP* supports the following objectives:

1. *Define the validation test execution platform*, the necessary equipment and the common and reused validation components. A detailed description of the *validation test execution platform* is performed in *Appendix C.*
2. *Recommend and describe the strategy for validation test application.* The *SVP* indicates, for each software functionality in the scope of the delivery, the *types of validation tests* to be performed and if the execution of the corresponding tests is manual or automatic. In *Table 2.10*, a description of the test types is provided.

Type of test	Description
Functional	Ensure the proper functionality of the whole software
Data integrity	Ensure the proper functionality of the whole software during Data treatment
Failover and recovery	Ensure the proper functionality of the whole software during restoration process
Configuration	Ensure the proper functionality of the whole software when different system configurations are set
User interface	Ensure the user's interaction with the software
Performance	Ensure time-sensitive requirements: response times, transaction rates, etc.
Load	Ensure time-sensitive requirements: response times, transaction rates, etc.
Stress	Ensure time-sensitive requirements: response times, transaction rates, etc.
Long term	Ensure time-sensitive requirements over a long period of time
Security and data access control	Ensure the proper access to specific functionalities
Installation	Ensure the proper software installation process

Table 2.10 – Description of the type of tests used in validation test at Johnson Controls (Apostolov 2007)

3. *Establish the regression strategy***.** Two kinds of *regression strategy* are defined: *change oriented* and *priority oriented*. The purpose of the *change oriented* strategy is to define how to test a software product after new functionalities or changes are implemented / applied, in order to ensure that the implementation of not changed requirements is not impacted by the changes. The aim of the *priority oriented* strategy is to ensure that the quality of implementation of requirements having highest priority has not regressed while adding new capabilities.

Design the validation procedure

The purpose of this activity is to identify the structure of the whole *validation procedure* and to establish a link between *test cases* and requirements. In other words, this activity aims to identify the number of required *test cases* and describe the scope of each one. The software requirements (from the *SRS*) defined as scope of the following delivery and the *SVP* are used as inputs for the design of the *validation procedure*. In Johnson Controls, a list of good practices when designing the validation procedure has been established:

- It is recommended that, for each software requirement, at least one *test case* has to be defined and one *test case* could cover more than one requirement.
- One *test case* can cover only one type of test.
- A *test case* must cover all the aspects and combinations of a requirement.

Implement the validation procedure

The aim of this activity is to design, in a step by step manner, the test cases of the *validation procedure*. Based on each *test case* scope, *validators* analyze the carmaker requirements and design the *test case* that must verify the compliance of the software product with the corresponding requirement. The *test case design process* presently used by the engineers at Johnson Controls is deeply described in *Section 6*. It is important to note that *validators* do not have access to the *source code* of the software product under test. It is a considered as a *black-box* with a set of inputs and outputs. The technique of designing *test cases* without

having access to the source code of the software under test is called *functional* or *black-box* or *specification-based test*. A survey on *software testing* techniques (Bernot 1991, Beizer 1995) is provided in *Chapter 4 – Section 3.B*.

The language used to design *test cases* for the *validation test* of a software product depends on the *validation test execution platform*. In case of an automatic execution of the *test cases*, one uses a *script language*. It is a Johnson Controls property language very similar to the well-known *Visual Basic*[26] language. A detailed description of this language is provided in *Appendix B*. In case of a manual execution of the *test cases*, *test cases* are written in *natural language*.

b. Execution of validation

The validation is executed in the following sequence:

1. Configuration and initialization of the *validation test execution platform*.
2. Execution of the *test cases* in sequence defined by the *regression strategy* defined in the *SVP* (*incremental* or *full* validation).

During the execution of the *test cases*, "OK" and "NOK" results, which are prepared by observing and comparing the expected and the observed results, are set for each *test step*. The execution of the *validation procedure* could performed either automatically with the help of a tool or manually (Cf. *Appendix C* on the *validation test execution platform*). In the case of a "NOK" result, the comment describing the observed situation must be added and a bug has to be issued in the *problems' tracking tool*. A detailed description of the Johnson Controls *problems' tracking tool* is performed in *Section 7*.

VI. The test case design process presently used in automotive industry

Our audit (Cf. *Section 5*) on the software *V&V* activities within Johnson Controls has confirmed the proposal of the *National Institute of Standards and Technology* (Cf. *Definition 1.5*): "*Software testing* is a widespread *V&V* technique in automotive industry". In fact, we notice that each of the *Development* (*unit test*) and *Validation* (*validation test*) processes perform *software testing* in order to verify and validate the correctness of the software delivered at the end of the process.

Presently, most of automotive suppliers have a manual *test design process*. As the software products become more and more complex (Cf. *Chapter 1*), it is illusory to be able to check that the software product responds correctly to all possible *operations*. In *Chapter 8 – Section 2*, we further demonstrate that *software testing* is a *NP-Complete* problem and therefore it is impossible to be able to cover all the *operation space*. In fact, for each software component or product under test, we can associate a *potential operation space* (Cf. *Figure 2.12*). Each engineer has a different perception of the possible and critical *operations* (based on her/his experience). Therefore, based on a common *test objective*, two engineers could choose different *test cases* according to their perception. In Johnson Controls, a software component or product is always tested against predefined objectives.

[26] Computer language (http://msdn.microsoft.com/en-us/library/sh9ywfdk(vs.80).aspx, Consulted on November 2008).

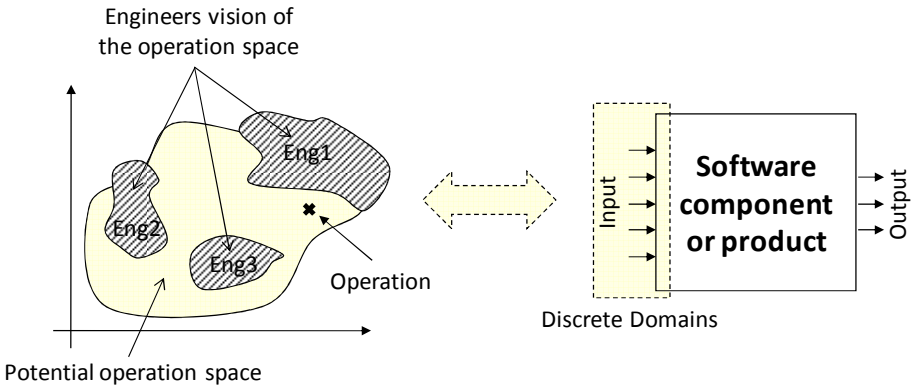


Figure 2.12 – Potential operation space of a software

A. Design test cases for the unit test

In Johnson Controls, the main purpose when *testing* unitarily a software component is to cover at 100% the *source code* of the component under test. *Developers* analyze the structure of the software component under test (*White-box test*), select one *operation* within the *potential operation space* and choose a *time to wait* before a next *operation* (Cf. *Figure 2.13*). Afterwards, by analyzing the *source code* of the component, they assess the expected values to be checked on some *output signals*. We note that *developers* do not check the behavior of all the *output signals* of a software after each *operation*. In fact *developers* decide to check only some *output signals* in relation with the performed *operation*. In fact, they verify the expected behavior according to their understanding of the program behavior. If the designed *test steps* allow to cover all the *source code*, *developers* stop designing *test steps*. If not, *developers* analyze deeply the uncovered pieces of code with the goal of designing one or more *test steps* that cover these pieces of code. Sometimes, for *time and budget reasons*, managers could decide to stop *testing* unitary a software component even if the 100% *code coverage* is not reached. The principles of *code coverage* are developed in the next section.

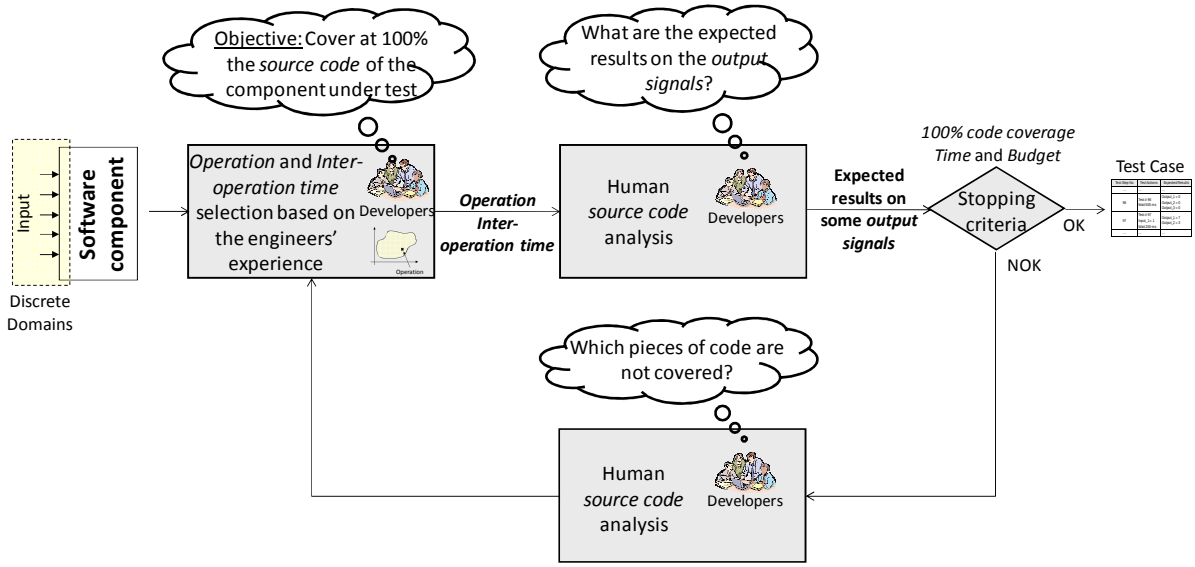


Figure 2.13 – Johnson Controls present approach to design a test case for unit test

1. Code (structural) coverage

A survey on *code coverage* based *testing* tools is done in (Yang 2006). In Johnson Controls, the *code coverage* is measured by a commercial tool (*C-Cover*[27]). *Code coverage* is a way to measure how thoroughly a set of *test cases* covers a code (Cf. *Figure 2.14*):

- The *coverage rate* of *statements*: This metric reports whether each executable *line of code* is encountered.
- The *coverage rate* of *procedures*: This metric reports whether the *test case* invokes each *procedure* (or *function*) of the software. It is useful during preliminary *testing* to assure at least some *coverage* in all areas of the software.
- The *coverage rate* of *decisions*: This metric reports whether *boolean* expressions tested in control structures (such as the *if-statement* and *while-statement*) evaluated to both *true* and *false*. The entire *boolean* expression is considered one *true-or-false* predicate regardless of whether it contains *logical-and* or *logical-or* operators. Additionally, this metric includes *coverage* of *switch-statement cases*, *exception* handlers, and *interrupt* handlers.
- The *coverage rate* of *conditions*: *Condition coverage* reports the *true* or *false* outcome of each *boolean* sub-expression, separated by *logical-and* and *logical-or* if they occur. *Condition coverage* measures the sub-expressions independently of each other. This metric is similar to *decision coverage* but has better sensitivity to the *control flow*. However, full *condition coverage* does not guarantee full *decision coverage*.

For instance, the piece of code of the *Figure 2.14* has: 1 *procedure*, 1 *condition*, 2 *decisions* and 8 *statements*.

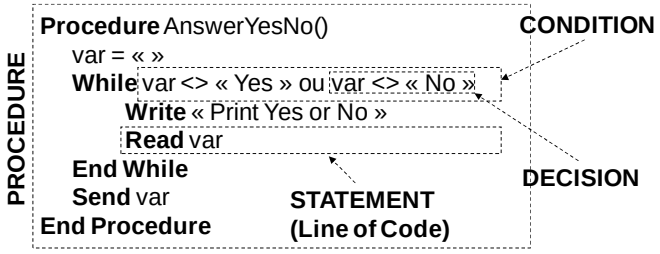


Figure 2.14 – Code (structural) coverage indicators

These criteria are apparently relevant since the goal of the *testing* activity is to check if all the pieces of the software have been visited. But it is not that simple (according to experts)!

In 2006, we analyze the unitary test reports on more than 5 projects related to different type of products. We also discuss these reports with inner software experts. In fact, even if the 100% *code coverage* is not reached, managers can decide to stop *testing* unitary each software component for *time and budget reasons*.

Diagnosis 9

Sometimes, the unit test of a software component is incomplete or even inexistent. In other words, the source code of the component under test is not covered at 100%. As a consequence, the uncovered pieces of code could hide critical bugs.

[27] http://www.bullseye.com/productInfo.html (Consulted on November 2008).

B. Design test cases for the validation test

Presently, in Johnson Controls, the *unit test* is not responsible to verify the compliance of a software component with the carmaker requirements. In fact, once a set of unitarily tested components are integrated together, *validators* have the responsibility of verifying the compliance of the whole software product with the carmaker requirements. To do this, *validators* analyze one or more software requirements (*Black-box test*) and select one *operation* within the *potential operation space* (Cf. *Figure 2.15*). Afterwards, by analyzing the carmaker requirements, they assess the expected values to be checked on some *output signals*. Idem to the design of a *test case* for the *unit test*, *validators* decide to check only some *output signals* in relation with the performed *operation*. In fact, they verify the expected behavior according to their understanding of the carmaker requirements. If the designed *test steps* allow to cover the *carmaker requirements* under test, *validators* stop designing *test steps*. If not, *validators* analyze deeply the considered requirements with the goal of designing one or more *test steps* that cover at 100% the requirements under test. Sometimes, for *time and budget reasons*, managers could decide to stop validating a software product even if the 100% *requirement coverage* is not reached. However, the carmaker must be notified on the uncovered requirements. The *requirement coverage*, as it is currently practiced in Johnson Controls, is developed in the next section.

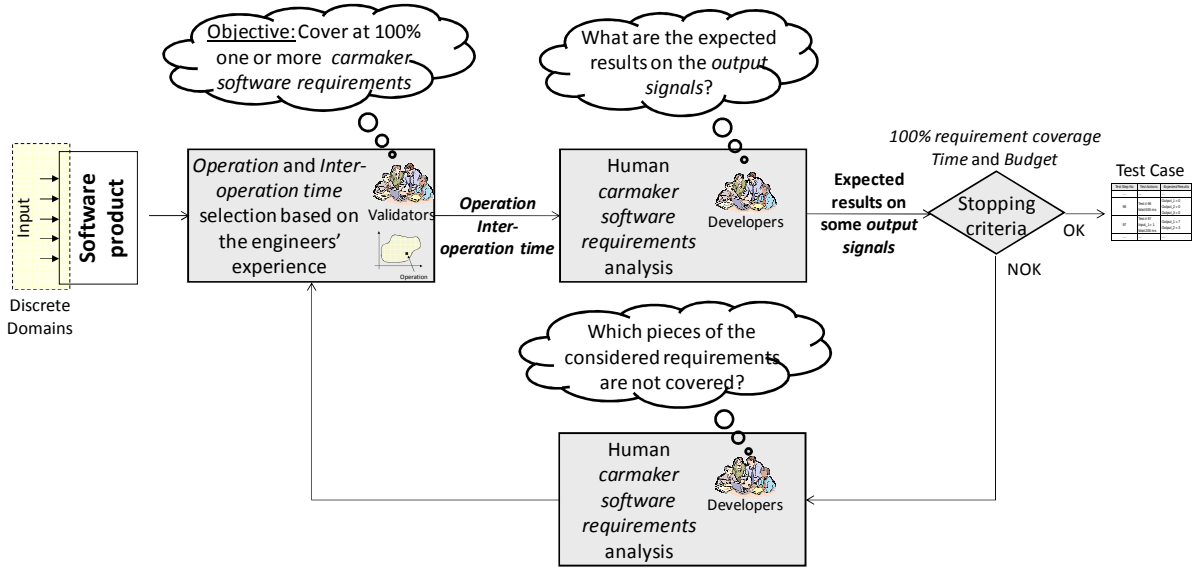


Figure 2.15 – Johnson Controls present approach to design a test case for validation test

In 2007, we analyze the bugs of two different projects related to two different electronic products. It is important to note that, in Johnson Controls, bugs detected during *review* and *unit test* activities are often not capitalized in the *problems' database* (Cf. *Section 7*). Once a bug is detected during these activities, it is corrected immediately by the person who detects it. Therefore, most of the capitalized bugs are detected in *validation test*. Through our study, we note that up to 30% of the stored bugs are due to *errors* in the design of the *test cases* in *validation test*. In fact, *validators* do not assess correctly the expected values to be checked on the *output signals*. This could be explained by the fact that a human assessment of a program behavior could be inaccuracy since carmaker requirements related to the software domain become more and more complex.

Diagnosis 10

When testing a software component or product and after an operation on the input signals, test engineers do not check the behavior of all the output signals of the component or product under test. Based on their understanding of the program behavior and/or the carmaker requirements, test engineers decide to check only some output signals in relation with the performed operation. In fact, they verify the explicit expected behavior but not the implicit one.

1. Requirement (functional) coverage

The criterion of *code coverage* does not directly assess the compliance of the software component or product with the carmaker requirements; this is a biased indicator. In fact, the *requirement coverage* is related to the *coverage* of the functional requirements of the software under test. Through a literature review (Dalal 1988, Bontron 2005, Yang 2006), several stop *testing* criteria based on covering software requirements have been identified in *Chapter 4 – Section 4.B.1*. They primarily deal with the *transitions coverage* of a *graph-based* specification. At Johnson Controls, the carmaker requirements related to the software domain are referenced and managed using the *SRS* model and the *coverage rate* of these requirements is mainly used as the criterion to stop *validation test*. Moreover, the *requirement coverage* is measured subjectively by the *validator*. Paradoxically and even a 100% *coverage* of the functional requirements has been reached during the *testing* of a software product, the carmaker is able to detect a nonconformity between the code and their requirements. In fact, presently, one requirement can hide two or more other requirements. Let us consider in *Figure 2.16* an excerpt of software functional requirements as they were defined by a Johnson Controls *engineer*. These requirements have two inputs and one output: *I1* (with domain D(I*1*)={0,1}), *I2* (D(*I2*)={0,1}), *O1* (D(*O1*{0,1}).

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

Figure 2.16 – An excerpt of software functional requirements as defined by a Johnson Controls engineer

During the *validation test*, one inexperienced *validator* designs one *test case* (composed from two *test steps*) in order to cover the previous requirements:

- *Test step* 1: set *I1* to 1, *I2* to 0 and check if *O1* is equal to 0
- *Test step* 2: set *I1*, *I2* to 1 and check if *O1* is equal to 1

Therefore, she/he decides to stop *testing* these requirements and to set them as covered. In fact, through *test step #1*, the *validator* covers at 100% the first requirement but *test step #2* does not cover at 100% the second requirement. Indeed, the second requirement can be split into three "implicit" requirements to be tested:

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

This could lead to the conclusion that the present Johnson Controls definition of a requirement is not enough refined.

Diagnosis 11

The present definition of a software requirement is not enough refined. In fact, one requirement can hide two or more implicit requirements. Therefore, inexperienced validators could miss testing some of the carmaker implicit requirements.

Based on our analysis of the present Johnson Controls approaches to design *test cases* for *unit* and *validation test*, we do the four diagnoses listed below.

Diagnosis 12

In validation test and after selecting an operation to be performed on the software under test, test engineers analyze the carmaker requirements in order to assess the expected values to be checked on some output signals of the software. In fact, this assessment is based on the engineers' understanding of the requirements and may lead to errors.

Diagnosis 13

For each software component or product under test, a large potential operation space is associated. Each engineer has a different perception of the possible and critical operations based on her/his experience. Therefore, the present strategy to select operations in order to test a software is irrelevant.

Diagnosis 14

The test cases designed by engineers do not always simulate the real use of the software product under test. The main purpose of testing activities is to cover the software code and requirements. As a direct consequence, basic user operations on the product could be not tested by the supplier before a carmaker delivery.

Diagnosis 15

Presently, the test cases for a software are manually designed by engineers. As the size of automotive software growth, this task becomes a laborious task and accounts for more than 50% of the total time and budget of a project.

VII. Capitalizing bugs in Johnson Controls

A. Snapshot on the Johnson Controls problems' tracking tool

Johnson Controls as many other electronic suppliers uses a *problems' tracking tool* (*TeamTrack*[28]) in order to manage and store problems detected during a project. A snapshot of this tool is illustrated in *Figure 2.17*.

[28] http://www.serena.com/products/teamtrack/index.html (Consulted on November 2008).

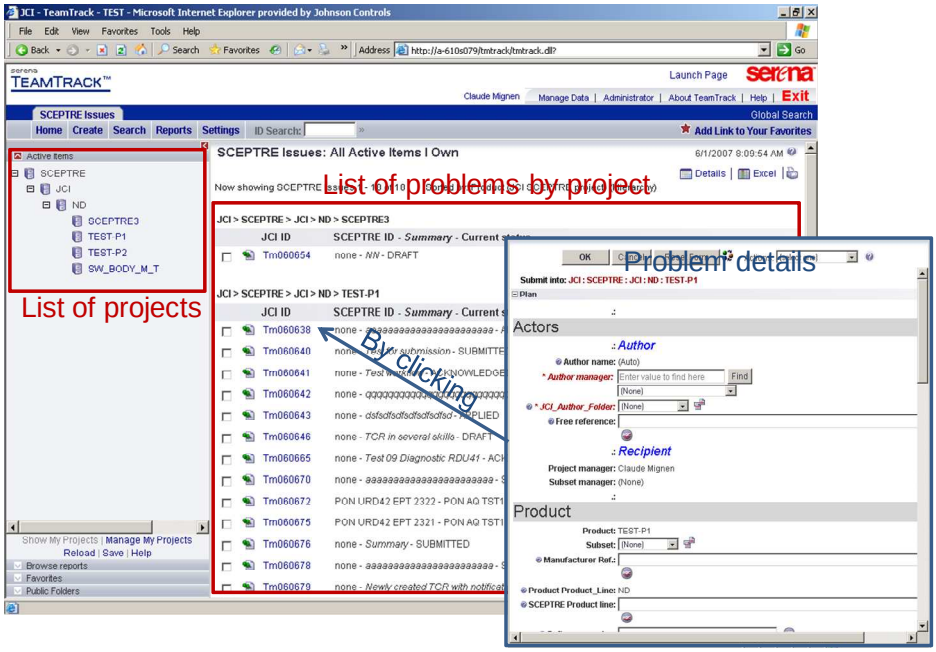


Figure 2.17 – Screenshot of the problems' tracking tool

Problems are classified according to four categories: *software*, *hardware*, *mechanical* and others. In the following, we focus on software problems, called bugs. The *tracking tool* has a database where all the problems are stored by project. In fact, a project is the combination of a customer (for instance, *Renault*), a type of product (for instance, a *body controller module*) and a car platform (for instance, *Laguna platform*). The *problems' database* has been created in the late 90's and now we estimate to tens of thousands the number of capitalized software bugs. According to experts, about 60% of these bugs are "true" bugs. The remaining 40% are duplications or without continuation. Moreover, in 2006, we perform a study on the capitalized software bugs and we come up to the conclusion that up to 90% of these bugs were detected during the *validation test* activity. Software experts and managers confirm that the bugs detected during the other *V&V* activities (*review* and *unit test*) are often not capitalized in the *problems' database*. Once a bug is detected during these activities, it is corrected immediately by the person who detects it. Most of the capitalized software bugs are detected in *validation test*.

B. The bug's model currently used in Johnson Controls

One of the *support* processes (Cf. *Figure 2.2*) has the responsibility of ensuring that all found software bugs and all changes on the product are identified, analyzed, managed and controlled to resolution and implementation. In fact, once an engineer has recorded a bug in the *problems' database*, a workflow process is initiated between the team members in order to:

- assess the impacts of the bug,
- make decisions,
- plan and implement the corrections,
- and finally verify and validate the *non-regression* of the software product.

Apart the evolution of the bug status (*dynamic view*) since its creation and till its resolution, we focus on the bug's model (*static view*) currently used in Johnson Controls. Fifteen years

ago, Sagem[29] software experts developed a bug's model with the aims of 1) managing the life cycle of a problem, 2) having traceability of the problems detected internally and by the carmaker, 3) monitoring a project by the number of detected, corrected and uncorrected problems and finally 4) reusing (by experts analysis) critical stored problems to avoid similar problems on future developments. In fact, a total of 111 attributes should be filled in by the engineers for each capitalized problem. We analyze about 2000 bugs from two different projects and products and we come up to the conclusion that 75% of these attributes are filled in; the remaining 25% are systematically unfilled. On the 75% filled attributes, 25% of these attributes are free fields. In *Figure 2.18*, we classify the 111 attributes according to the major aspects of a software bug (Mellor 1992, Fenton 1996): location (9 attributes), timing (28 attributes), symptom (2 attributes), impact (2 attributes), cause (7 attributes), type (14 attributes), severity (7 attributes) and cost (2 attributes). The remaining 71 attributes are related to Johnson Controls administrative data necessary for the management of the bug.

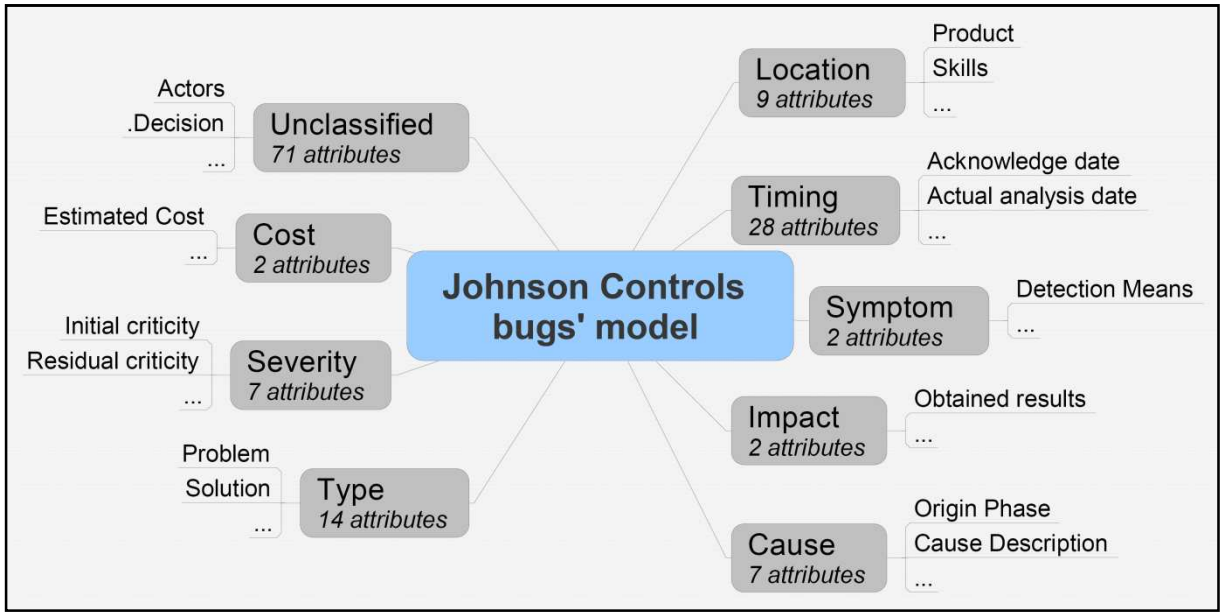


Figure 2.18 – Bug's model currently used in Johnson Controls (this figure is voluntarily uncompleted for confidentiality reasons)

As stated before, some attributes of the bug have free fields in the *problems' tracking tool* and therefore engineers can write anything they want with the main objective of giving as many information as possible on the bug. In *Figure 2.19*, an excerpt of a bug stored in the *problems' database* is illustrated. Since the attribute *Problem Description* has a free field, each engineer has the possibility to fill in this field according to her/his reasoning approach. Technical language (*code variables*, *electronic* and *software jargon* …) is often used in such case. In fact, there is no standard format that engineers must respect when describing a bug. Moreover, attributes such as *Cause Type* and *Description* are sometimes not filled in. In fact, through these attributes, one could identify the responsibility of persons in the problem.

[29] In 2001, Johnson Controls buys the electronics business from Sagem Automotive.

ATTRIBUTES	DESCRIPTION OF THE BUG
…	**…**
Problem description **A "very" technical description of the problem**	*from int volume and vehicle speed. (cf ... specification) Test example. Initial Condition : Ignition = 1 CanData1 = 1 CanData2 = 1 Vehicle_Speed = 0 Wiper_Intermitent = 1 Obtained Result : Intermitent time = 8 s ... Then Vehicle_Speed = 20000 (200 km/h) Obtained Result : Intermitent time = 8s Expected Result : Intermitent time = 4s ... Tested on E-CAR*
…	**…**
Cause type	**Not filled in by the engineers**
Cause description	**Not filled in by the engineers**
…	**…**
Origin phase	*Development*
…	**…**
Detection Means	*Validation test*
…	**…**

Figure 2.19 – An excerpt of a bug stored in the problems' database (this figure is voluntarily uncompleted for confidentiality reasons)

Diagnosis 16

When describing a bug in the problems' tracking tool, there are too many fields to fill in (111 attributes), a lot of free fields (about 25%) and a lack of relevant predefined fields (for instance, a bug's typology). As the detection of bugs comes later in the process, engineers do not have enough time to fill in all the fields of a bug (missing information). Moreover, in case of free fields, an engineer could write anything she/he wants with the main objective of giving as many information as possible on the bug (irrelevant information). Since information is missing and/or irrelevant, it remains a difficult problem to reuse bugs in order to avoid or detect similar problems on future developments.

C. Existing techniques to reuse capitalized bugs

We analyze many internal documents related to the reuse of bugs stored in the *problems' database*. We also interview software experts and managers on the current *knowledge management* practices. We come up to the conclusion that bugs stored in the *problems' database* are rarely used to avoid similar problems on future developments and ensure that carmakers will not encounter the same problems on two similar products.

Actually, no advanced (*formal* and *automated*) techniques have been implemented in order to reuse stored bugs. Nevertheless, three traditional strategies are currently practiced:

- *Create and update "Lessons Learned Checklist" for software developments***.** The process of creating and updating *lessons learnt* is illustrated in *Figure 2.20*. On the one hand and once a bug is detected on a project, the *project leader* decides if this bug must be verified on other projects or not. The decision process is not formal and is mainly based on the experience of the decision maker. In case of a reused bug, this bug is transferred to the *Software Engineering Process Group* (*SEPG*) which confirms or not the possible re-use of this bug. The way of describing a bug in the *problems' database* has a major impact on this process. On the other hand, a *software forum* exists where engineers can submit their questions, remarks and recommendations to the *SEPG* which decides or not to generalize the submitted issue. Finally, each reused bug or group of bugs and each general issue is summarized in a *lesson learnt* (a textual sentence) to be consulted on future developments. In *Figure 2.21*, an excerpt of a *lessons learned checklist* is illustrated.

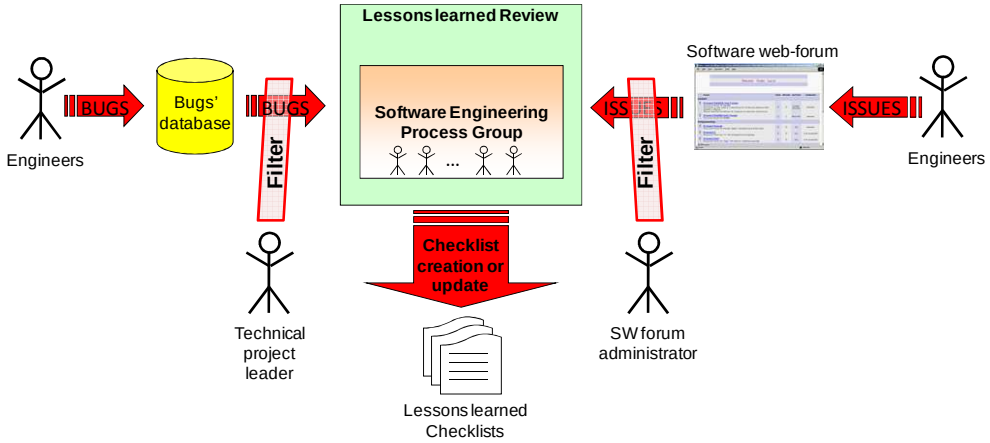


Figure 2.20 – Process of creating and updating "Lessons Learned Checklists" for software skill (Mignen 2005)

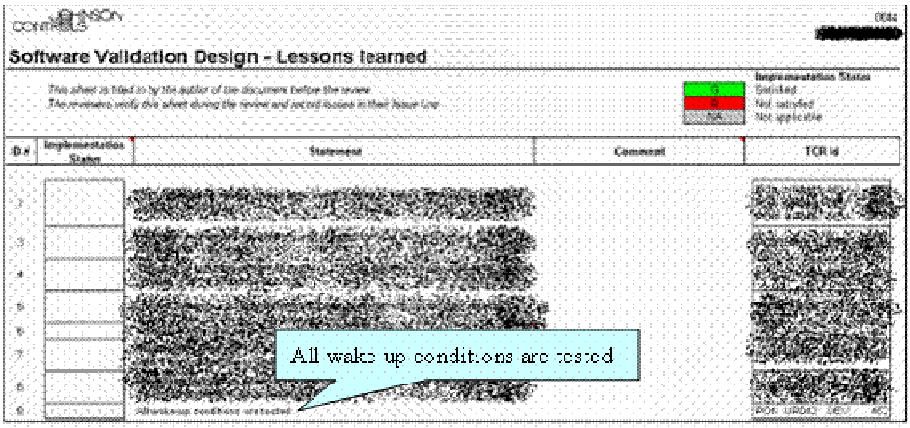


Figure 2.21 – An excerpt of lessons learnt to be checked during the design of the validation procedure - This figure is voluntarily fuzzyfied for reasons of confidentiality (Fradet 2008)

- *Use of peer project to check problems on similar products.* At the beginning of each new project, a list of "similar" previous projects (according to experts) is identified. Then, engineers have to review all the problems (including software bugs) detected on these projects and identify a list of "critical" problems to be checked on the new project.
- *Increase the reuse of software components from one project to another.* A software product is composed from a set of software components fulfilling different services. Therefore, the reuse of a component from one project to another must be a usual process. However, the challenge is to develop components with standard interfaces and configurations. The reuse of components is advocated since it increases the quality and productivity. Indeed, *lessons learnt* are already included in the reused software components.

However and according to experts, some bugs occur again from one project to another. In order to confirm this citation, we perform a study on the bugs detected on one software functionality, the *front wiper management* functionality, implemented in five different projects since 1997 and till 2007. In fact and according to experts, all the projects related to the same type of product and car platform of one carmaker have up to 70% of common functionalities. In *Table 2.11*, for each project, we identify the release year of the project, the number of *Lines Of Code* (*LOC*) implementing the *front wiper* functionality and the number of bugs detected on this functionality.

Projects	Year	Number of Lines Of Code implementing the front wiper functionality (without comments and blanks)	Number of bugs detected on the front wiper functionality
Project 1	1997	3909	30
Project 2	2001	1457	4
Project 3	2003	889	5
Project 4	2003	1255	16
Project 5	2007	1229	22

Table 2.11 – Characteristics of the front wiper functionality implemented in five different projects since 1997 and till 2007

Through the analysis of these five projects, we note that the *front wiper* functionality has, in common, 7 features. On some projects, there are one or more additional features that we ignored in our study. We make two classifications of the bugs detected on this functionality. The first one according to the 7 features (Cf. *Figure 2.22*) and the second one according to a typology of bugs (Cf. *Figure 2.23*) borrowed from the literature (Beizer 1990, Chillarege 1992, Grady 1992, IEEE Std. 1044-1993). Then, we make the arithmetic mean by feature and by type of bugs of the number of bugs detected on *projects 1*, *2*, *3* and *4*. In fact, we try to demonstrate that before developing the *front wiper* functionality on the *project 5*, we were able to predict which feature is the most critical in terms of bugs' occurrence (*feature 3*) and which types of bugs engineers are vulnerable to (*Control flow* and *sequencing*).

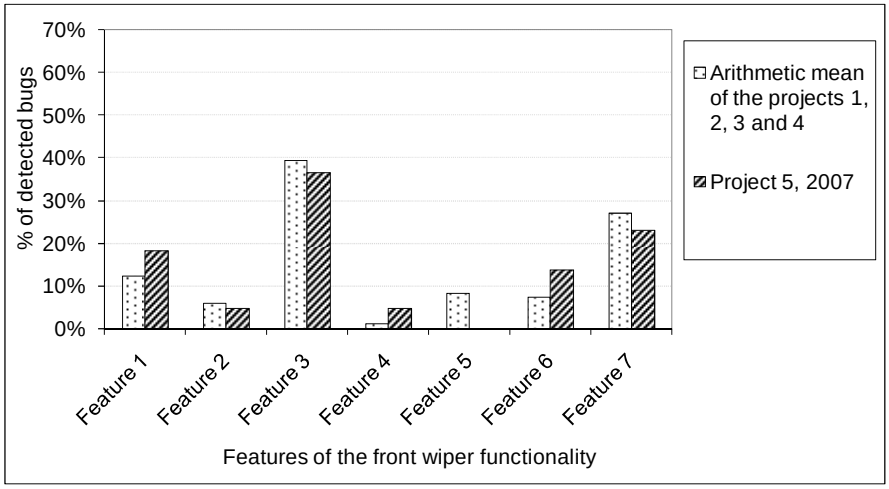


Figure 2.22 – Classification of the bugs according to the front wiper's features

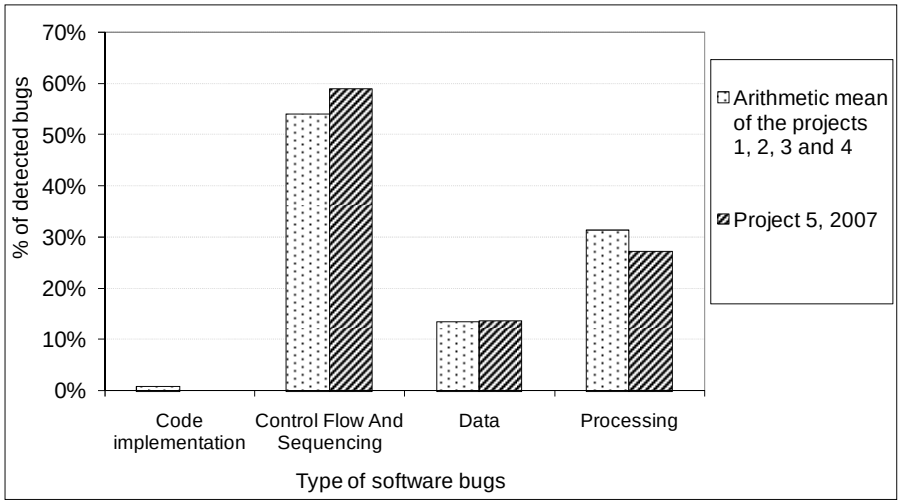


Figure 2.23 – Classification of the bugs according to a typology of software problems

As a conclusion, the thousands of bugs stored in the *problems' database* are not systematically nor efficiently reused to avoid or detect recurrent bugs. To better understand the reasons for that, we interview software managers who refers to four main issues for reusing stored bugs:

- Manual analysis by experts of the *problems' database* is impossible: thousands of bugs and 111 attributes by bug.
- Lack of information: 25% of a bug's attributes are systematically not filled in.
- Ambiguous and incomplete information: 25% of the filled attributes are free field and these attributes (for instance, problem description) are the most important to detect recurrent type of bugs.
- Lack of knowledge in *data mining* processes and tools to perform an automatic analysis of the *problems' database*.

Diagnosis 17

There are no advanced (formal and automated) techniques to reuse bugs stored in the problems' database in order to avoid or detect similar bugs on future developments. In fact, carmakers are unhappy when encountering the same type of problem on two different products delivered by the same supplier.

VIII. Managing and reusing test cases in Johnson Controls

The languages used for designing *test cases* in each of the *unit* and *validation test* activity are usually *computer languages*. Indeed, *test cases* for *unit test* are developed in *C language* and *test cases* for *validation test* are developed in a *script language* specific to Johnson Controls. A detailed description of these languages is provided in *Appendix B*. Presently, the versions of the software components of a project are managed through a commercial *version manager tool* (*PVCS*[30]). In consequence, the *test cases* for *unit* and *validation tests* are also versioned using this tool and stored in the same folder as the related software component or functionality.

As stated before (Cf. *Section 7.C*), all the projects related to the same type of product and car platform of one carmaker have up to 70% of common functionalities. Therefore, using capitalized *test cases* seems to be beneficial in automotive context. In other words, when *testing* a software functionality that we already implemented in the past on another project, it is judicious to reuse existing *test cases*. Unfortunately, in Johnson Controls capitalized *test cases* are not always reused from one project to another. We interview software experts and managers on this phenomenon and we identify two main reasons. The first one is the use of different formats to specify a *test case*. Sometimes, engineers specify the *test cases* immediately in the *computer language* (*C language*, *script language*) understandable by the *test execution platform*. Others use the *test case* format presented in *Figure 2.11*. In fact, the use of *computer languages* makes the reuse of *test cases* a difficult task. One has to analyze and adapt *test cases* written in a *computer language* from one project to another. It is important to note that now, *testing* a software product of about 200 *KLOC* (*Kilo Lines Of Code*) requires about 1000 *KLOC* of tests (Johnson Controls source). The second one is the lack of an automated process to reuse *test cases*. The manual analysis and adaptation of *test cases* from one project to another seems a laborious task. It could be more time consuming to

[30] http://www.serena.com/products/pvcs/pvcs-version-manager.html (Consulted on November 2008).

adapt existing *test cases* than to design new ones. In this situation, the *textual analysis tools* could help but unfortunately such tools are absolutely not known in the company.

Diagnosis 18

Currently, test engineers use different formats to specify a test case. Sometimes, engineers specify the test cases in a computer language (C language, script language), others use a more high level test case format (independent from the technology). Moreover, there is a lack of formal process and tools to manage and reuse test cases from one project to another.

However, one initiative was launched two years ago and had the purpose to create manually *standard test cases* for software validation. An example of a *standard test case* as it was developed at Johnson Controls is illustrated in *Table 2.12*.

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

Table 2.12 – Example of a standard test case as developed at Johnson Controls

A set of *standard test cases* were developed and classified by functionality of product. In fact, potential bugs by product functionalities are identified and documented in *standard test case patterns*. These patterns must be systematically consulted (for the given product type) before beginning *testing* stages. This is a conventional *RETEX* (*RETurn of EXperience*) strategy, but which remains to be completed for any *product line*. The two main difficulties of such an approach are to 1) describe *standard test cases* with a suitable language level understandable by any *test engineer* and 2) keep the list of *these test cases* updated without exploding their number. Two years after, many issues faced this *test case* reuse strategy and coerced software managers to stop it. The four main issues are 1) the list of *standard test cases* is no more updated due to a lack of resources, 2) an exploding number of *standard test cases*, 3) all the *standard test cases* are stored in the same *Word* document which becomes unmanageable and finally 4) most of the *standard test cases* are too much detailed and therefore not understandable by a newly-graduated *test engineer*.

IX. Conclusion

Through our *industrial audit*, we analyze the current software practices at Johnson Controls and make diagnoses on the current *V&V* activities of a software. The performed diagnoses are listed in *Table 2.13*. In *Figure 2.24*, we locate each of these diagnoses within the Johnson Controls software organization.

In the following chapter, based on our industrial audit, we clearly define the scope of our research. We also formulate our research topic in accordance with the research issues in *software testing*.

Diagnosis number	Diagnosis description
1	*Verification and Validation* (*V&V*) practices and *test cases* are rarely shared between the carmakers and their electronic suppliers.
2	Now, one cannot get a degree in software *V&V*. Software *V&V* is incorporated into the *software engineering* degree. Moreover, software engineers often prefer *development* activities compared with *verification and validation* activities.
3	In automotive industry, *semi-formal* and *formal* methods are more and more used to specify software functional requirements. However, there is a lack of a standard formalism shared between carmakers and suppliers. In fact, for each project, the supplier has to adapt its processes to the formalism used by the carmaker.
4	Deadlines for carmaker requirements *freeze* are specified in the carmaker-supplier *contract*. Nevertheless, the carmaker's requirements evolve continuously along the software development life cycle without complying with these deadlines. Moreover, suppliers must react quickly by integrating (without *regression*) the changes in the product.
5	The *Software Requirement Specification* (*SRS*) document is often a large document (about hundreds of pages), difficult to manage, incomplete and not regularly updated.
6	Sometimes, the *SRS* document is the *official* and *contractual* document between carmaker and supplier. It is also the main document used by the development and *V&V* teams in their activities. It has a standard structure but there are no standards to specify carmakers' requirements (more especially *functional requirements*).
7	Sometimes, the review of software code is badly done or even ignored. In fact, number of code reviewers is not aware of the *coding rules* and *recommendations* to be checked. Moreover, the *code review* is not systematically performed on each new software component. In consequence, the *code review* does not often detect all the bugs that must be detected through this activity.
8	According to the *"To Be" process*, the *unit test* of a software component must ensure a 100% *source code coverage*. However, this *V&V* technique is not responsible to verify the compliance of the component's behavior with the carmaker requirements. One software component can be tested unitarily (100% *of code coverage*) without fulfilling the behavior required by the customer.
9	Sometimes, the *unit test* of a software component is incomplete or even inexistent. In other words, the source code of the component under test is not covered at 100%. As a consequence, the uncovered pieces of code could hide critical bugs.
10	When testing a software component or product and after an *operation* on the *input signals*, test engineers do not check the behavior of all the *output signals* of the component or product under test. Based on their understanding of the program behavior and/or the carmaker requirements, test engineers decide to check only some *output signals* in relation with the performed *operation*. In fact, they verify the *explicit* expected behavior but not the *implicit* one.
11	The present definition of a software requirement is not enough refined. In fact, one requirement can hide two or more implicit requirements. Therefore, inexperienced validators could miss testing some of the carmaker implicit requirements.
12	In *validation test* and after selecting an *operation* to be performed on the software under test, test engineers analyze the carmaker requirements in order to assess the expected values to be checked on some *output signals* of the software. In fact, this assessment is based on the engineers' understanding of the requirements and may lead to errors.
13	For each software component or product under test, a large *potential operation space* is associated. Each engineer has a different perception of the possible and critical *operations* based on his/her experience. Therefore, the present strategy to select *operations* in order to test a software is irrelevant.
14	The *test cases* designed by engineers do not always simulate the real use of the software product under test. The main purpose of testing activities is to cover the software code and requirements. As a direct consequence, basic *user operations* on the product could be not tested by the supplier before a carmaker delivery.
15	Presently, the *test cases* for a software are manually designed by engineers. As the size of automotive software growth, this task becomes a laborious task and accounts for more than 50% of the total *time and budget* of a project.
16	When describing a bug in the *problems' tracking tool*, there are too many fields to fill in (111 *attributes*), a lot of free fields (about 25%) and a lack of relevant predefined fields (for instance, a *bug's typology*). As the detection of bugs comes later in the process, engineers do not have enough time to fill in all the fields of a bug (missing information). Moreover, in case of free fields, an engineer could write anything she/he wants with the main objective of giving as many information as possible on the bug (irrelevant information). Since information is missing and/or irrelevant, it remains a difficult problem to reuse bugs in order to avoid or detect similar problems on future developments.
17	There are no advanced (formal and automated) techniques to reuse bugs stored in the *problems' database* in order to avoid or detect similar bugs on future developments. In fact, carmakers are unhappy when encountering the same type of problem on two different products delivered by the same supplier.
18	Currently, test engineers use different formats to specify a *test case*. Sometimes, engineers specify the *test cases* in a computer language (*C language*, *script language*), others use a more high level *test case* format (independent from the technology). Moreover, there is a lack of formal process and tools to manage and reuse *test cases* from one project to another.

Table 2.13 – List of diagnoses on the software V&V practices in Johnson Controls

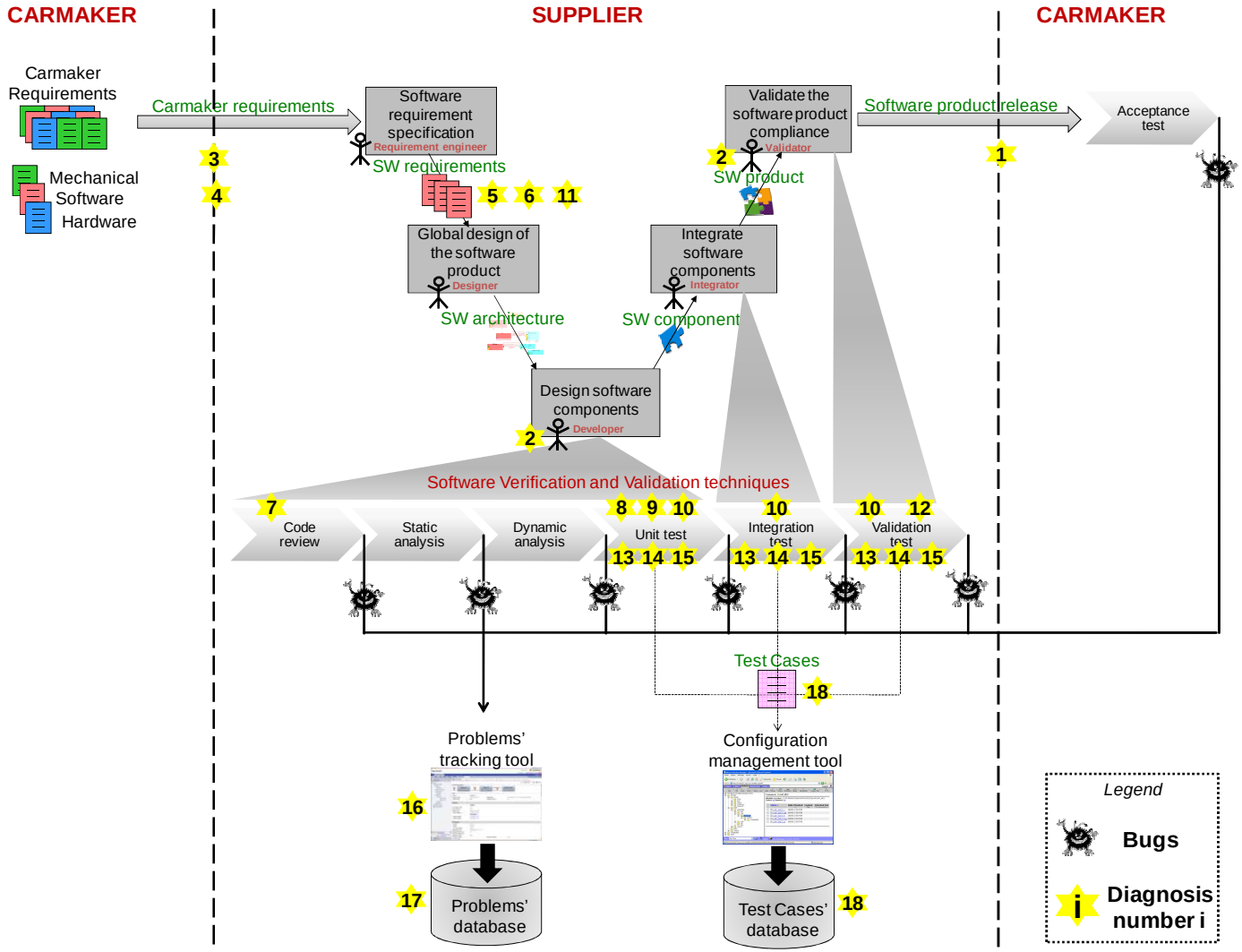


Figure 2.24 – Localization of the diagnoses within the Johnson Controls software organization

PART II –PROBLEM STATEMENT AND LITERATURE REVIEW

CHAPTER 3. RESEARCH TOPIC

I. Introduction

In our research, we go through the Johnson Controls problem with a *systemic approach* in order to identify domains from which we might be able to improve the *global performance* of the software *Verification and Validation* (*V&V*) activities. In *Chapter 2*, we perform an industrial audit and make diagnoses on the current *V&V* practices within the company. Based on the *industrial audit*, one could isolate critical anomalies and lacks in the current engineers' practices. A review of related solutions proposed in the literature could help in defining or adapting relevant solutions to our context.

In this chapter, we clearly define the *scope of our research* and we formulate our *research topic* based on the performed diagnoses and the related research *issues in software testing*. The industrial and academic needs and objectives are summarized in *Section 2*. A specification of our research topic and focus is done in *Section 3*. Finally, the main *software testing* issues is highlighted in *Section 4*. In the conclusion of this chapter, we identify the diagnoses which are in the scope of our research and associate for each of these diagnoses one or more related *software testing* issues (as stated in the *literature*).

II. Industrial and academic needs and objectives

A. Initial industrial needs

Facing the fierce competition within the automotive industry and the strong pressure that carmakers impose on their suppliers to reduce the cost and the development time, Johnson Controls is looking for new and innovative engineering solutions to increase its performances and therefore better satisfy the requirements and expectations of their clients. As said in *Chapter 1 – Section 6*, electronic parts represent now up to 30% of the global cost of a car and software bugs represent more than 80% of the problems detected on such a product. Therefore, decreasing the development cost and increasing the quality of software product have become of main interest for carmakers. Johnson Controls as an automotive electronics supplier has launched many initiatives (this *PhD* is one of these initiatives) inside its *engineering centers* around the world with the aims of:

1. Decreasing the number of bugs detected by the carmaker - *Quality*
2. Reducing the development time of electronic projects - *Delay*
3. Reducing the development cost of electronic projects - *Cost*

Through our research project, we were asked by an automotive electronic supplier namely Johnson Controls to improve the performance of its software *Verification and Validation* (*V&V*) activities. Their main purpose is to improve the quality of their products and therefore better satisfy the requirements and expectations of their clients. In Johnson Controls, the *validation test* which is the last *V&V* activity before a carmaker delivery is considered as the ultimate activity to detect all the bugs and therefore deliver carmakers "bug-free" software. It represents up to 90% of the time spent in the *V&V* of a software product (Cf. *Chapter 2 – Section 5*). While the *test cases* execution activity is often automated via a *test execution platform*, the *test case* design activity remains a manual task that fills in time many engineers. Up to 50% of a software project team is dedicated to design *test cases* (Cf. *Table 1.1*). Therefore, for many of software managers and experts in Johnson Controls, *automating the design of test cases* seems to be the most adapted solution to reduce the *testing* time, cost and resources while improving the *code* and *requirement coverage*.

B. Academic objectives

Since the 1990s, there has been a renewed interest in the development of effective *software testing* techniques. In the years, the topic has attracted increasing interest from researchers, as testified by the many specialized events and workshops, as well as by the growing percentage of *testing* papers in *software engineering conferences*. A recent paper titled *"Software Testing Research: Achievements, Challenges, Dreams"* of a software engineering pioneer named Bertolino (Bertolino 2007) organizes the many outstanding research challenges for *software testing* into a consistent roadmap. One of his conclusions was that there is a need to make the process of *software testing* more effective, predictable and effortless. In addition, the author pinpoints the many fruitful relations between *software testing* and other research areas. In fact, by focusing on the specific problems of *software testing*, we may overlook many interesting opportunities arising at the border between *software testing* and other disciplines. Unfortunately, few papers exist in which the problem of *software testing* is considered with a *systemic* approach.

Our primary scientific goal has been to go through this problem with a *systemic approach* in order to identify levers in any domains from which we might be able to improve the *global performance* of the software *V&V* activities. The added value of such an approach is the resolution of the problem with a *global quality* viewpoint. Therefore, in *Chapter 2*, we perform an *industrial audit* on the software practices currently used in automotive industry and more especially in Johnson Controls. Through the audit, we characterize the overall environment of our problem. We understand what verifying and validating a software product means and we point out the main current issues and lacks in the automotive *V&V* activities.

III. Research scope

A. Research topic formulation

Through a primary industrial audit at Johnson Controls, we first analyze the *V&V "To Be"* processes, activities and techniques. Then, we characterize the software engineers' practices *("As Is"* processes, activities and techniques) in verifying and validating a software product. As a conclusion of the audit, we perform a list of diagnoses on the current *V&V* practices within the automotive industry and more especially automotive electronic suppliers (such as Johnson Controls). **Based on these diagnoses**, our research topic may now be refined through these two questions:

1. **How to detect bugs early in the software development life cycle? In other words, How to detect bugs closer to where they were introduced?**
2. **How to detect "all" the bugs of a software product before a carmaker delivery? Or, at least, how to measure that few bugs remain to be found?**

B. Research focus

Let us define a bit more the exact contour of our research. It was defined in accordance with the Johnson Controls priorities and interests. We were not authorized to intervene within the software design process in itself to a priori lower the number of bugs. It has been considered as another issue. In other words, **we do not work on avoiding bugs while designing and developing a software product but on detecting the bugs once the product is developed**.

In *Chapter 1 – Section 5.C.2*, we identify two types of *V&V* techniques: the *static* ones and the *dynamic* ones (e.g. *software testing*). In *Chapter 4 – Section 2.B.1*, we perform a survey

on the *static techniques* and on how they are adapted or not to the automotive context. **Based on our industrial audit** (Cf. *Chapter 2 – Section 5.B*), Johnson Controls presently performs most of the *review static techniques* (*technical review*, *walkthrough*, *inspection* and *audit*). On the contrary, the *proof static technique* is still considered as a non-adapted method to the automotive and more especially Johnson Controls context. Even if s*tatic techniques* are necessary to detect errors earlier in the development process, they are not sufficient. In fact, these techniques focus on analyzing the *static* product representation and do not test the product in its real life (*dynamic*). This could explain the fact that, in Johnson Controls, *V&V dynamic* techniques are considered as the ultimate techniques to detect all the bugs. They represent up to 90% of the time spent in the *V&V* of a software product (Cf. *Chapter 2 – Section 5*). As a consequence, **we focus our research on the *V&V* dynamic techniques. The main *dynamic V&V technique* is the *software testing*.**

Testing a software product requires two main activities. A detailed specification of these activities is done in *Chapter 2 – Section 6*. The first one consists of designing *test cases* and the second one of executing these *test cases* on the software product under test. **Based on our industrial audit within Johnson Controls** (Cf. *Chapter 2 – Section 5*), the execution of *test cases* is performed thanks to Johnson Controls property *test execution platform*s. These platforms are described in *Appendix C*. On the one hand, the number of bugs related to a wrong execution of a *test case* is minor regarding the one related to an irrelevant design of a *test case* (1 over 100 – Johnson Controls source). On the other hand, the design of *test cases* is a manual task that accounts for up to 50% of a software project time. Therefore, **we focus our research on the design of efficient *test cases* for software**. In fact, **we are interested in any *organizational matter* that has a positive influence onto the quality of the *test case design process*: *simulation platform, knowledge management, competency management* and *project management***.

IV. Hot research issues in software testing

The Bertolino's definition (Bertolino 2003) of the *software testing* technique highlights the four main *testing* issues (four underlined words in the definition).

Definition 3.1: Software testing (Bertolino 2003)

Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain against the specified expected behavior.

Dynamic: *testing* implies executing the program on (valued) inputs. Since *static techniques* (*review*, *inspection* …) are useful to evaluate the internal correctness of a software product, *testing* is the only technique allowing the assessment of its behavior when executed in its real environment.

Research issue 1: How to execute test cases on a software product? (This issue is not in the scope of our research)

Finite: even for simple programs, so many *test cases* are theoretically possible that *exhaustive testing* would require years to execute. Dijkstra (Dijkstra 1972) calculated that the *exhaustive testing* of a multiplier of two *27-bit integers* taking "only" some tens of microseconds for a single multiplication would require more than 10000 years. In *Chapter 8 – Section 2*, we demonstrate that *testing* exhaustively a software product is a *NP-Complete* problem from a computational viewpoint. Generally, the whole test set can be considered infinite. In contrast, the number of executions that can realistically be observed must obviously be finite (and affordable). Clearly, "enough" *testing* to get reasonable assurance of acceptable behavior must

be performed. This basic need points to well known issues of *testing*, both *technical* in nature (criteria for deciding to stop *testing*) and *managerial* in nature (estimating the effort to put in *testing*). *Testing* always implies a *trade-off* between limited resources and schedules, and inherently unlimited test requirements.

***Research issue 2:* When to decide to stop testing a software product?**

Selected: many *operation selection techniques* differ on their strategy to select a finite number of *operations.* Test engineers must be constantly aware that different techniques may lead to quite different quality results; they also may be much dependent of context factors such as the kind of application, the maturity of the process and the organization, the expertise of test engineers, the tool platform. How to select the most suitable *operations* to be performed on the software under test is a complex issue (Vegas 2001).

***Research issue 3:* How to choose the "relevant" operations to be checked on a software product?**

Expected: it must be possible (although not always easy) to decide whether the observed outcomes of program execution are acceptable or not, otherwise, the *testing* would be useless. The observed behavior may be checked against specifications and user's expectations. The test *pass*/*fail* decision is commonly referred in the *testing* literature to as the *oracle problem*.

***Research issue 4:* How to assess the expected behavior of a software product?**

V. Conclusion: our diagnoses, the scope of our research and the software testing research issues

Based on our research focus, we identify in *Table 3.1* the diagnoses which are in the scope of our research (the design of *test cases*). One diagnosis is related to the static *V&V* techniques and three diagnoses are related to the carmakers' practices on which a supplier can absolutely not act. We also associate for each of the diagnoses in the scope of our research one or more related *software testing* issues (as stated in the *literature*).

Diagnosis number	Diagnosis description (Cf. *Table II.13*)	In the scope of our research Design of test cases	Related software testing issues Literature review
1	*Verification and Validation* (*V&V*) …	NO – related to carmakers' practices	-
2	Now, one cannot get a degree in software *V&V* …	YES	Issue 2, 3 and 4
3	In automotive industry, *semi-formal* and *formal* …	NO – related to carmakers' practices	-
4	Deadlines for carmaker requirements *freeze* are …	NO – related to carmakers' practices	-
5	The *Software Requirement Specification* (*SRS*) …	YES	Issue 2, 3 and 4
6	Sometimes, the *SRS* document is the *official* …	YES	Issue 2, 3 and 4
7	Sometimes, the review of software code is …	NO – related to static V&V techniques	-
8	According to the *"To Be" process*, the *unit test* …	YES	Issue 2
9	Sometimes, the *unit test* of a software …	YES	Issue 2
10	When testing a software component …	YES	Issue 4
11	The present definition of a software requirement …	YES	Issue 2
12	In *validation test* and after selecting an …	YES	Issue 4
13	For each software component or product under …	YES	Issue 3
14	The *test cases* designed by engineers do not …	YES	Issue 3
15	Presently, the *test cases* for a software are …	YES	Issue 3 and 4
16	When describing a bug in the *problems' tracking* …	YES	Issue 2 and 3
17	There are no advanced (formal and automated) …	YES	Issue 3
18	Currently, test engineers use different formats …	YES	Issue 3

Issue 1 is not in the scope of our research

Table 3.1 – Our diagnoses, the scope of our research and the software testing research issues

In the following chapter, we perform a literature review on the existing approaches, techniques and tools in the field of the *V&V* of software products. More especially, we focus our research on finding or adapting "solutions" for the anomalies and lacks (diagnoses) that we identify via our *industrial audit*. In fact, we mainly develop the *literature* related to the *software testing issues 2*, *3* and *4*.

CHAPTER 4. STATE-OF-THE-ART

I. Introduction

Constructing reliable products continues to be one of software development's greatest challenges. *Testing*, one of the most crucial tasks along the software development life cycle can easily exceed half of a project's total effort. A successful *testing* approach can save significant effort and increase product quality, thereby increasing customer satisfaction and lowering maintenance costs.

Despite these obvious benefits, the state of *software testing* practice isn't as advanced as software development techniques overall. In fact, *testing* practices in industry are, most of the time, neither very sophisticated nor effective. This might be due partly to the perceived higher satisfaction from developing something new as opposed to *testing* something that already exists. Also, many software engineers consider test engineers as *second-class executives*. They consider *testing* as a junior or entry position and use it merely as a springboard into development jobs. However, *academia* spends significant effort in researching new *testing* approaches. Promising approaches have started to find acceptance in industry, but the technology transfer between *testing* research and industry is still insufficient. *Academics* sometimes say that industry is immature and practitioners are clueless, whereas practitioners might argue that researchers squander their time developing cool but useless *testing* technologies. As it often happens, the truth lies somewhere in between.

In this chapter, we develop the literature related to the *software testing issues 2*, *3* and *4* and we focus our research on finding or adapting "solutions" for the anomalies and lacks (diagnoses) that we identify via our *industrial audit*. An overview on the software *verification and validation* (*V&V*) techniques is proposed in *Section 2*. A classification of the *software testing* techniques is done in *Section 3*. Finally, *software testing* issues and related solutions are developed in *Section 4*. We identify lacks in these solutions and propose improvement actions in order to fit in our context. In the conclusion of this chapter, we summarize the improvement actions that we propose all along the chapter.

II. Verification and Validation of software products

A. Principles

Verification and Validation (*V&V*) of software are defined in the present report after the *IEEE Standard Glossary of Software Engineering Terminology* (*IEEE* Std. 610-1990).

Definition 4.1: Verification and Validation (IEEE Std. 610-1990) – Abbreviation: V&V

The V&V process is the process of determining whether the requirements for a product or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final product or component complies with the specified requirements. The distinction between verification and validation has been well-framed by Barry Boehm, who memorably described verification as "building the product right" and validation as "building the right product".

Software *V&V* helps the product designers and test engineers to confirm that a right product is build right way throughout the development process and improve the quality of the software product. It makes sure that, certain rules are followed when developing a software product and also makes sure that the developed product fulfills the required specifications. This reduces the risk associated with any software project up to certain level by helping in detection and correction of faults, which are unknowingly done during the development process.

The standard definition of verification is: "Are we building the product RIGHT?" e.g. *verification* is makes sure that the software product is developed the right way. The software must confirm to its predefined specifications, as the product development goes through different stages, an analysis is performed to ensure that all required specifications are met. The *verification* part of *V&V* comes before validation and incorporates *software inspections*, *reviews*, *audits*, etc. During the *verification*, the *work product* (the ready part of the software being developed and various documentations) is reviewed / examined by one or more persons in order to find and point out the bugs in it. The *verification* helps in prevention of potential bugs.

The standard definition of validation is: "Are we building the RIGHT product?" e.g. a software product must do what the customer expects it to do. The software product must functionally do what it is supposed to, it must comply with any functional requirement set by the customer. *Validation* occurs at the end of the development process in order to determine whether the product complies with specified requirements. *Validation* starts after verification ends (after coding of the product is completed). *Testing* methods are basically carried out during the *validation*.

B. Software verification and validation techniques

Whatever the size of project, software *V&V* greatly affects software quality. People are not infallible, and software that has not been verified has little chance of working. Gibson in (Gibson 1992) stated that typically, 20 to 50 errors per 1000 *Lines Of Code* (*LOC*) are found during development and 1.5 to 4 per 1000 *LOC* remain even after *validation test*. Each of these errors could lead to an operational failure (bug) or non-compliance with a requirement. The objective of software *V&V* is to reduce software errors to an acceptable level. According to Beizer (Beizer 1990), the effort needed can range from 30% to 90% of the total project resources, depending upon the criticality and complexity of the software. The *V&V* techniques must be applied at each stage in the software process. It has two major objectives 1) the discovery of bugs in a product and 2) the assessment of whether or not the product is useful and useable in an operational situation. *V&V* must establish confidence that the software is fit for purpose. This does not mean completely fre*e o*f defects. Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed. Confidence is certainly subjective and depends on many factors such as software criticity, users and market expectations. The *V&V* consists of numerous techniques and tools, often used in combination with one another. Due to the large number of *V&V* approaches in use, we cannot address every technique. In fact, software *V&V* both use *static* and *dynamic techniques* of product checking to ensure that the resulting software product matches with its specifications and that the software product as implemented meets the expectations of the customer. In fact, *dynamic techniques* involve the execution of the software product under test, whereas *static techniques* do not:

- *Static techniques* (*Review* and *Proof*) are concerned with analysis of the static product representation to discover errors throughout all stages of the software life cycle. It may be complemented by *tool-based document* and *code analysis*.
- *Dynamic techniques* (*Testing*) are concerned with exercising and observing product behavior. The product is executed with test data and its operational behavior is observed.

1. Static techniques

a. Review

A *review* is a technique during which a *work product*, or set of *work products*, is presented to project personnel, managers, users, customers, or other stakeholders for comment or approval (*IEEE* Std. 610-1990). *Review* can be used to examine all the products of the software evolution process. In particular, they are especially applicable and necessary for those products not yet in machine processable form, such as requirements or specifications written in lateral language. *IEEE* (*IEEE* Std. 610-1990) has identified four kinds of *review* which are often used for software verification: *technical review, walkthrough, inspection and audit.* These *reviews* are all "*formal* reviews" in the sense that all have specific objectives and explicit rules of procedures. They expect to identify errors and discrepancies of the software regarding the original specifications, plans and standards.

Technical review

The objective of a *technical review* is to evaluate a specific set of review items (e.g. document, *source code*) and provide management with evidence that 1) they conform to specifications made in previous phases; 2) they have been produced according to the project standards and procedures and finally 3) any changes have been properly implemented, and affect only those products identified by the change specification. Typical conclusions of a review meeting are 1) authorization to proceed to the next phase, subject to updates and actions being completed, 2) authorization to proceed with a restricted part of the product and 3) a decision to perform additional work.

Walkthrough

Walkthrough should be used for the early evaluation of documents, models, designs and code. The objective of a *walkthrough* is to evaluate a specific review item (e.g. document, *source code*). A *walkthrough* should attempt to identify errors and consider possible solutions. In contrast with other forms of review, secondary objectives are to educate, and to solve form errors.

Inspection

Inspection can be used for the detection of errors in detailed designs before coding and during the coding stage. *Inspection* may also be used to verify *test cases*. A study done by Fagan (Fagan 1986) has shown that inspection could detect over 50% of the total number of errors introduced in development stages. *IEEE* (*IEEE* Std. 610-1990) considers that inspection is a more rigorous alternative to walkthrough, and is strongly recommended for software with stringent reliability, security and safety requirements.

Audit

Audit is an independent *review* that assesses compliance with software requirements, specifications, baselines, standards, procedures, instructions, codes and contractual and licensing requirements. To ensure their objectivity, *audit* should be carried out by people independent of the development team.

b. Proof

A *proof* attempts to logically demonstrate that software is correct. Whereas a *test* empirically demonstrates that specific inputs result in specific outputs, *proof* logically demonstrate that all

inputs meeting defined pre-conditions will result in defined post-conditions being met. Adrion (Adrion 1986) defines *proof* as is a collection of techniques that apply the *formal*ity and rigor of mathematics to the task of proving consistency between an algorithm solution and a rigorous, complete specification of the intent of the solution. This technique is also referred to as "*formal* verification". *Proof* techniques are normally presented in the context of verifying an implementation against a specification.

Prowell and Beizer (Powell 1986, Beizer 1990) have identified several limitations to *proof* techniques. *One limitation* is the dependence of each *proof* technique to a *formal* specification language. In fact, in order to use a specific *proof* technique on a project, the software requirements of this project must be written in a specific language associated with the corresponding *proof* technique. *Another limitation* has to do with the complexity of using *proof* techniques. For large programs, the amount of detail to handle, combined with the lack of powerful tools may make the *proof* technique impractical. According to inner software managers, *proof* techniques are not suitable to the automotive competitive context and more especially to Johnson Controls. The four main reasons are:

- *Proof* techniques are often used on critical software products. They often have precise and logical specifications with no loopholes and they require being highly reliable, since failures in this kind of products may lead to deathly consequences. Some areas where *proof* techniques have been successful are for the specification and verification of safe and critical products such as aircraft avionics, nuclear power plant control and patient monitoring. In Johnson Controls, the developed electronic products are related to the car interior functionalities and are not considered by the carmakers as critical.
- Automotive engineers are not familiar with *proof* techniques contrary to aeronautic or defense engineers. *Software testing* is a widespread *V&V* technique in automotive industry.
- *Proof* techniques are not widely used in automotive industry (carmakers and Johnson Controls competitors). This could lead to the conclusion that *proof* techniques are not adapted to the automotive context. In fact, the difficulty of expressing software requirements in the mathematical form necessary for *formal proof* has restricted a wider application of this technique.
- Finally, many managers highlight the complexity and the additional effort required regarding *reviewing* or *testing* techniques.

As said in the research focus (Cf. *Chapter 3 – Section 3.B*), we do not address the *static V&V techniques* but we focus our research on the *dynamic techniques* (e. g. *software testing*).

2. Dynamic techniques

Software testing, a *V&V dynamic* technique is a widespread technique in automotive industry. In Johnson Controls (Cf. *Chapter 2 – Section 5*), *software testing* represents up to 90% of the total time spent in verifying and validation a software product. Moreover, in the academic research, the traditional focus of software *V&V* techniques has been the *software testing*. In fact, *testing* approaches are widely studied in academic research and deployed in software industry. Therefore, in our *literature review*, the *software testing* category has been further refined. In the following section, we expose the major *testing* principles, techniques and issues.

III. Software testing techniques

A. What is “software testing”?

Harrold (Harrold 2000) has identified several advantages of *testing* over *static-analysis* techniques. *One advantage* of *testing* is the relative ease with which many of the testing activities can be performed. *Test cases* can be generated automatically. Software can be instrumented so that it reports information about the executions with the *test cases*. This information can be used to measure how well the *test cases* satisfy the quality objectives. Output from the executions can be compared with expected results to identify those *test cases* on which the software failed. *A second advantage* of testing is that the software being developed can be executed in its expected environment. The results of these executions with the *test cases* provide confidence that the software will behave as intended. *A third advantage* of testing is that much of the process can be automated. With this automation, the *test cases* can be reused for testing as the software evolves. Although, testing has a number of advantages, it also has a number of limitations. *Testing* cannot highlight the absence of errors; it can only stress their presence. Additionally, *testing* cannot show that the software has certain qualities. Despite these limitations, *testing* is widely used in industry to provide confidence in the quality of software. Therefore, the growth of software complexity and the increased emphasis on software quality, highlight the need for improved *testing* methodologies. In the following, we list some citations of software pioneers around the world.

> *“Quality assurance over test designs and testing is essential to a successful quality effort. [...] More than the act of testing, the act of designing tests is one of the most effective bug preventers known. [...] The ideal quality assurance activity would be so successful at this that all bugs would be eliminated during test design. Unfortunately, this ideal is unachievable. We are human and there will be bugs. To the extent that quality assurance fails to reach its primary goal of bug prevention, it must reach its secondary goal of bug detection.”*
>
> *B. Beizer, (Beizer 1984)*

> *“Reliable Object-Oriented software cannot be obtained without testing.” — R.V. Binder*
>
> *Binder, (Binder 1995)*

> *“The importance of software testing and its implications with respect to software quality cannot be overemphasized. [...] It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software (e.g. flight control, nuclear reactor monitoring) can cost three to five times as much as all other software engineering activities combined!”*
>
> *R.S. Pressman, (Pressman 1997)*

However, several different definitions have been given for the *software testing* technique. Some of them are listed below. In this dissertation, we adopt the definition proposed by the *National Institute of Standards and Technology* (*NIST*).

Definition 4.2: Software testing (NIST 2002)

Software testing is the process of applying metrics to determine product quality. Software testing is the dynamic execution of software and the comparison of the results of that execution against a set of pre-determined criteria. "Execution" is the process of running the software on a computer with or without any form of instrumentation or test control software being present. "Predetermined criteria" means that the software's capabilities are known prior to its execution. What the software actually does can then be compared against the anticipated results to judge whether the software behaved correctly. Software testing is a widespread V&V technique in automotive industry.

Definition 4.3: Software testing (Myers 1979)

Software testing is the process of executing a program or system with the intent of finding errors.

Definition 4.4: Software testing (IEEE Std. 610-1990, IEEE Std. 829-1998)

Software testing is:

(1) the process of operating a software component or product under specified conditions, observing or recording the results, and making an evaluation of some aspect of the component or product.

(2) the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

While the *NIST* definition associates the *software testing* to a quality measurement tool, the definition of Meyers insists on the fact that testing software must reveal bugs and the *IEEE* definition claims the good behavior of the software under test.

B. Classification of software testing techniques

There is an excess of testing methods and testing techniques. Classified by life-cycle phase, *software testing* can be categorized as follows: *unit test*, *integration test*, *validation test* and *regression test*. Classified by accessibility, *software testing* can be divided into: *white-box test* and *black box test*. All these test methods can be used (individually or in conjunction) at each phase of the software life-cycle. In the following, we provide some details on each of these testing techniques.

1. According to life-cycle phase

During the development lifecycle of a software product, *testing* is performed at different levels and can involve the whole product or parts of it. Depending on the process model adopted, then, *software testing* activities can be articulated in different phases, each one addressing specific needs relative to different portions of a product. Whichever the process adopted, Bernot (Bernot 1991) states that one can at least distinguish in principle between *unit*, *integration* and *validation test*. These are the three *testing* levels of a traditional phased process (such as in Johnson Controls). Pezze (Pezze 1998) considers that these levels are complementary with different goals and execution procedures. In fact, none of these levels is more relevant or important than the others. Each level must address a specific *typology of bugs* in a software product. The *unit test* must detect bugs related to the behavior of each

software component independently from its environment. The *integration test* focuses on problems of communications and interfaces that may arise during component integration. And finally the *validation test* focuses on the behavior of a software product as a whole.

a. Unit or component or module test

A component is the smallest testable piece of software (Cf. *Definition 1.3*), which may consist of hundreds sometimes thousands of *LOC*, and generally represents the result of the work of one programmer (*developer*). Bernot (Bernot 1991) defines the *unit test* as a *V&V* technique to ensure that a software component satisfies its functional specification and/or that its implemented structure matches the intended design structure. The *unit test* can also be applied to check the *local data structure* (improper typing, incorrect variable name, inconsistent data type) and the *boundary conditions*. In other words, go through all the *source code* of a software component.

b. Integration test

Generally speaking, *integration* is the process by which software components are aggregated to create a software product. Bernot (Bernot 1991) defines the *integration test* as the technique that aims at verifying that each component interacts with other components according to its specifications. In particular, it mainly focuses on the communication interfaces among integrated components. Even though the single components are individually acceptable when tested in isolation, in fact, they could still result in incorrect or inconsistent behavior when combined. For instance, there could be an improper call or return sequence between two or more components. Fenton (Fenton 2000) has identified two *integration test* approach: *non-incremental* and *incremental*. In a *non-incremental* approach the components are linked together and tested all at once (*big-bang testing*). In the *incremental* approach, we find the classical *top-down* strategy, in which the modules are integrated one at a time, from the main program down to the subordinated ones, or *bottom-up*, in which the tests are constructed starting from the modules at the lowest hierarchical level and then are progressively linked together upwards, to construct the whole product. Usually in practice (as in Jonson Controls), a mixed approach is applied, as determined by external project factors (e.g. availability of modules, release policy, availability of test engineers and so on).

c. Validation or system test

Validation test involves the whole software product and is defined by Bernot (Bernot 1991) as the technique that aims at verifying that the whole software behaves according to the customer requirements. In particular it attempts to reveal bugs that cannot be attributed to specific components, but they are due to the inconsistencies between components, or to the planned interactions of components and other objects (which are the subject of *integration test*). In (Bertolino 2002), Bertolino summarizes the primary goals of *validation test*:

- Discovering the bugs that manifest themselves only at system level and hence were not detected during *unit* or *integration test.*
- Increasing the confidence that the developed product correctly implements the required capabilities.
- Collecting information useful for deciding the release of the product.

Validation test must therefore ensure that each product function works as expected.

d. Regression test

In (Bernot 1991), the author consider that the *regression test* is not a separate level of testing, but may refer to the retesting of a component, a combination of components or a whole software product after modification, in order to ascertain that the change has not introduced new errors. As software produced today is constantly evolving, driven by market forces and technology advances, *regression test* takes by far the predominant portion of *testing* effort in industry. Since both corrective and evolutive modifications may be performed quite often, to re-run after each change all previously executed *test cases* would be prohibitively expensive. Therefore various types of techniques have been developed to reduce *regression test* costs and to make it more effective. Fernandez (Fernandez 1996) proposes a selective *regression test* techniques based on selecting a (minimized) subset of the existing *test cases* by examining the modifications. Other approaches instead prioritize the *test cases* according to some specified criterion (for instance maximizing the *code coverage*).

2. According to accessibility

Testing methods can also be divided into two families, according to the *input data* from which *test cases* are selected (Beizer 1990): *black-box test* and *white-box testing*. *Black-box test*, a term most likely borrowed from electronic engineers, involves treating software component or product as a *black-box* (like an electronic component) to which input can be supplied and from which the corresponding output can be collected and observed, but whose inner intermediate workings of the software cannot be seen. *White-box* test does allow one to observe the internal workings of the software and to make use of its structural information to adapt or drive the *testing* process.

a. Functional or black-box or specification-based test

According to Beizer (Beizer 1995), *test cases* for *functional test* are derived from the functional specification of the software product under test, apart from the code. The criterion of correctness is the functional specification of the software under test: program behaviors are compared to those required by the specification. The goal is to select *test cases* that cover each requirement described by the functional specification. *Functional test* is typically the base-line technique for designing *test cases*, for a number of reasons. *Functional test case design* can (and should) begin as part of the *requirement specification* process. Even if the *source code* of a software is not already developed, one can design functional *test cases* for this software based on the software functional requirements. Moreover, *functional test* is effective in finding some classes of bugs that typically elude *structural test* techniques. *Functional test* techniques can be applied to any description of program behavior, from an *informal* partial description to a *formal* specification and at any level of granularity, from software component to product *testing*.

Since, *functional test* aims at finding any discrepancies between what a software does and what it is intended to do, one must obviously refer to requirements as expressed by users and specified by software engineers. An important side effect of test design is highlighting weaknesses and incompleteness of software functional requirements. A survey on the *formal*ism degree of the software functional requirements is performed in *Section 4.D.1*. Designing functional *test cases* is an analytical process which decomposes requirement specifications into *test cases*. In most cases (as in Johnson Controls), *functional test* is a human intensive activity. For instance, when test engineers work from *informal* specifications written in *natural language*, much of the work is in analyzing the specification for identifying

test cases. Even expert test engineers can miss important *test cases*. Systematic processes amplify but do not substitute for skills and experience of the test engineers. In a few cases, *functional test* can be fully automated. This is possible for example when requirement specifications are expressed in a *formal* language (for instance, a grammar or an executable model). This approach is known under the name of *formal testing* or *model-based testing* and has been described by Apfelbaum and Robinson in (Apfelbaum 1997, Robinson 1999). The authors highlight the approach's advantage of guaranteeing a good and *formal coverage* of the requirements specification. In fact, the test engineers' job is limited to the choice of the *test selection criteria*, which defines the strategy for generating *test cases*. Several experiments have been performed in *testing* using *formal* specifications. A good summary of these experiments has been done by Gaudel and El-Far in (Gaudel 1995, El-Far 2001).

b. Structural or white-box or program-based test

According to Beizer (Beizer 1995), *test cases* for *structural test* are derived from the code of the software under test. In fact, the structure of the software itself is a valuable source of information for selecting *test cases* and determining whether a set of *test cases* has been sufficiently thorough. We can check whether a *test case* has covered a specific part of the program. In fact, *testing* can reveal an error only when the execution of the corresponding erroneous *items* causes a bug. For instance, if there were an error in the line *N* of the program, it could be revealed only with *test cases* that would cause this line to be executed. Based on this observation, a program has not been adequately tested if some of its *items* have not been executed. An *item* could be a *line of code*, *decision*, *condition* or *procedure* (Cf. *Chapter 2 – Section 6.A.1*). Unfortunately, a set of correct program executions in which all *structural items* are exercised does not guarantee the absence of errors. Execution of an erroneous *item* may not always result in a bug. The state may not be corrupted when the item is executed with some data values, and a corrupted state may not propagate through execution to eventually lead to a bug. Many software researchers (Jorgensen 1995, Pezze 1998, Woodward 2005) state that structural information must not be used as the primary answer to the question, "How shall I choose tests," but it is useful in combination with other *test selection criteria* such as cover the customer requirements.

Based on our industrial audit (Cf. *Diagnosis 8*), test engineers in Johnson Controls use the *structural* approach in the *unit test* stage and the *functional* approach in the *validation test* stage. The purpose of designing *test cases* using the *structural* approach is to cover at 100% the *source code* while using the *functional* approach, test engineers have to check the compliance of the software with the carmaker requirements. This leads to the fact that bugs related to the behavior (regarding the requirements) of one independent software component are detected later in the process (during the *validation test*). We propose to perform *functional test* since the earlier *testing* stages. One has to verify the compliance of each software component (independently from its environment) with the carmaker requirements.

IV. Software testing research issues and solutions

In this section, we develop the *software testing* issue identified in *Chapter 3 – Section 4*. In fact, we analyze the related solutions proposed in the *literature*, identify lacks in these solutions and propose *improvement actions* in order to fit in our context.

A. Research issue 1: How to execute test cases on a software product?

The execution of a *test case* can occur in a *manual* or *automated* way. In other words, the *test case* descriptions that are the result of the *test design* activity could be manually or automatically executed against a software product. One issue when automating the *test execution* is to transcribe the specified *test cases* into a computer language. Another issue is the ability to put the product into a state from which the specified *test cases* can be launched. This is sometimes referred to as the *test precondition*. In fact, before a specific command can be executed, several runs in sequence are required to put the product in the suitable *test precondition*. An effective way to deal with this is to arrange the selected *test cases* into suitable sequences, such that each *test case* leaves the product into a state similar to the *precondition* of the next *test case*. This problem has been early *formal*ized and tackled by Dick in (Dick 1993). Moreover, a more complex problem arises when *testing* only one or more components of a software product (the case of a software *unit test*). Indeed, the testing task itself requires a large programming effort. To be able to test one software component of a large software product we need to emulate the behavior of its peripheral software components. Fortunately, some commercial test tools exist which can facilitate these tasks. Finally, when testing reveals a bug, the task of recreating the conditions that made it occur is called *test replay*. Exact *test replay* requires mechanisms for capturing the happening of synchronization *operations*, and for forcing the same order of *operations* when a test is replayed.

As said in the research focus (Cf. *Chapter 3 – Section 3.B*), we do not address the problem of executing *test cases* on the software product. In fact, we made the assumption that the present *test execution platforms* are reliable.

B. Research issue 2: When to decide to stop testing a software product?

Determining when to stop *testing* and release a product is an important *management decision*. It is clear that there is natural *trade-off* between the decision to continue *testing* or to stop: (a) if *testing* stops too early, many bugs remain. Thus we incur the cost of later bug-fixing and losses due to customers' dissatisfaction. The cost of fixing a bug after release is a lot more than the cost of fixing while *testing* (Cf. *Figure 1.11*). (b) if *testing* continues up to the maximum permissible time, then there is the cost of *testing* effort and a loss of market initiative.

1. Criteria to stop testing a software

Several stopping criteria have been proposed for *software testing*.

a. Stochastic similarity

A *stopping criterion based on stochastic similarity* is proposed by Whittaker in (Whittaker 1994) and refined by Sayre in (Sayre 2000). This criterion is based directly on the statistical properties of a *usage* and *testing chains*. The *usage chain* is a model of ideal testing of the software; e.g. each arc probability is established with the best estimate of actual usage, and no failure states are present. The *testing chain*, on the other hand, is a model of a specific test history, including bug data. Thus, the *usage chain* represents what would occur in the statistical test in the absence of bugs, and the *testing chain* represents what has occurred. Dissimilarity between the two models is therefore a useful measure of the progress of *testing*. When the dissimilarity is small, the test history is an accurate picture of the *usage model*.

Unfortunately, Johnson Controls test engineers are often subjected to time pressure and therefore *test* and *bug data* are note often well organized. Therefore, in this context, the use of the *stochastic similarity* model to decide stop *testing* a software may lead to poor results.

b. Reliability estimation

A *stopping criterion based on estimated reliability and confidence* is proposed by Littlewood in (Littlewood 1997). This criterion relies on a target reliability. *IEEE* (*IEEE* Std. 610-1990) defines *software reliability* as the probability of *"bug-free"* software operations for a specified period of time in a specified environment. In fact, during the past 30 years, many models have been proposed assessing the reliability measurements of software products. A *software reliability model* specifies the general form of the dependence of the bug process on the principal factors that affect it: *error introduction*, *error removal*, and the *operational environment*. *Software reliability* modeling forecasts the curve of the bug rate by statistical evidences. The purpose of this measure is two-fold: 1) to predict the extra time needed to test the software to achieve a specified objective; 2) to predict the expected reliability of the software when the *testing* is finished. The success of a model is often judged by how well it fits a curve to the observed "number of bugs vs. time" function. It is important to note that all the *software reliability models* are based on some assumptions 1) The module under test remains essentially unchanged throughout *testing*, except for the removal of errors as they are found, 2) Removing an error does not affect the chance that a different error will be found, 3) "Time" is measured in such a way that *testing* effort is constant and finally 4) All errors are of equal importance. Unfortunately, none of these assumptions fit with our industrial context and therefore using such models in deciding when to stop *testing* a software in Johnson Controls may lead to poor results. Many of the current *software reliability* models, techniques and practices are detailed in the *Handbook of Software Reliability Engineering* by Lyu (Lyu 1996).

c. Cost benefit estimation

A *cost-benefit stopping criteria based on estimates of the errors remaining in the product and the cost to repair them both before and after release*, are proposed by Dalal in (Dalal 1988). A more sophisticated version which includes costs due to lost market and customer dissatisfaction is proposed by Chavez in (Chavez 2000). This model remedies all the assumptions considered by the *reliability models*. However, this leads to a complex mathematical problem. Since Johnson Controls test engineers are not familiar with mathematical theories (which are not the core of their skills), it remains difficult to apply such a model in our context.

d. Test coverage

A *stopping criteria based on test coverage* are presented by Offutt in (Offutt 1999a). The decision of when to stop *testing* is based on covering a software code or requirements in various ways. In practice, *code coverage* is used to decide when to stop *structural test*, while *requirement coverage* is used in a *functional test* context. On the one hand, researches in *code coverage* measurement have reached a high level of maturity and many automated tools were commercialized. In a survey done by Yang in (Yang 2006), the author studies and compares 17 *code coverage* measurement tools. In fact, the *code coverage* measurement helps engineers detecting *"dead code"*, piece of code that can be never covered and "non specified code", piece of code that does not implement any of the requirements. On the other hand, *requirement coverage* measurement is still immature. In fact, the accuracy of a *requirement*

coverage measurement depends on the degree of formalism used when specifying a set of requirements. In *Section 4.D.1*, we develop the three degrees of *formal*ism (*informal*, *semi-formal* and *formal*) used to specify software functional requirements. Measuring the coverage of an *informal* or *semi-formal* specification is usually done by a manual approach (Bontron 2005). In fact, all the requirements associated to the software product under test are identified and when designing a *test case*, test engineer has to identify the requirement(s) that has (have) been covered by this test. Obviously, such an approach is imprecise since it strongly depends on the engineers' degree of the specification comprehension and interpretation. Measuring the *coverage* of a *formal* specification can be considered as a simple problem that can be easily automated (Offutt 1999a). However, the *coverage* measurement criteria are specifics for each *formal*ism of *formal* specification. Now, in Johnson Controls, the *code coverage* is *formal*ly used when *testing* unitarily each software component. Moreover, in *validation test*, test engineers have to ensure a 100% *coverage* of the software functional requirements. In *Chapter 2 – Section 4.A*, we show that *semi-formal* and *formal* methods are more and more used to specify software functional requirements in automotive industry but there is not a unique standard *formal*ism shared between carmakers and suppliers. As an indirect consequence and even with *formal* specifications, test engineers still measuring the *requirement coverage* using the tradition manual approach presented in *Chapter 2 – Section 6.B.1*.

Based on our industrial audit (Cf. *Diagnosis 2, 5, 6, 8, 9, 11 and 16*), the stopping criterion used when *testing* unitarily a software component is the 100% *coverage* of the component source code. Sometimes, for time and budget reasons, test engineers stop testing a component even if the 100% *code coverage* is not reached. In validation test, the criterion to stop *testing* a software product is to cover at 100% the related carmaker requirements. These requirements are documented in the *SRS* document, a large document difficult to manage, incomplete and not regularly updated. Moreover, there are no standards to specify software requirements and test engineers have to adapt their *coverage* practices to each requirement's formalism. Finally, the present definition of a software requirement is not enough refined. In fact, one requirement can hide two or more implicit requirements. Therefore, inexperienced *validators* could miss *testing* some of the carmaker implicit requirements. Based on the literature review, we consider that ensuring a 100% *code coverage* is a necessary quality objective when *testing* a software. Nevertheless, we propose to *formal*ize the measurement of the *requirement coverage*. To do this, one has to specify the requirements using a *formal* language. Moreover, we suggest integrating *project constraints* (test time and cost) in the decision to stop *testing* a software product.

C. Research issue 3: How to choose the operations to be checked on a software product?

Effective testing requires strategies to *trade-off* between the two opposite needs of amplifying testing thoroughness on the one side (for which a large number of *test cases* would be desirable) and reducing times and costs on the other (for which the fewer the *test cases* the better). Given that test resources are limited, how the *operations* are selected becomes of crucial importance. Indeed, the problem of *operation selection* has been the major dominating topic in *software testing* research. A decision procedure for selecting the *operation* is provided by an *operation selection strategy*.

A basic strategy is *random testing*, according to which the *operations* are randomly chosen from the whole input domain according to a specified distribution, e.g. after assigning to the

inputs different "weights" (more properly probabilities). For instance the uniform distribution does not make any distinction among the inputs, and any input has the same probability of being chosen. In contrast with *random testing*, a broad class of *operation selection strategies* referred to as *partition testing*. The underlying idea is that the program input domain is divided into sub domains within which it is assumed that the program behaves the same, e.g. for every point within a sub domain the program either succeeds or fails: we also call this the *test hypothesis*. Therefore, thanks to this assumption only one or few points within each sub domain need to be checked, and this is what allows for getting a finite set of *operations* out of the infinite domain. Hence a *partition testing* strategy essentially provides a way to derive the sub domains. An *operation selection* strategy yielding the assumption that all *operations* within a sub domain either succeed or fail is only an ideal, and would guarantee that any fulfilling set of *operations* always detect the same bugs; in practice, the assumption is rarely satisfied, and different sets of *operations* fulfilling a same criterion may show varying effectiveness depending on how the *operations* are picked within each sub domain. The relative merits of these two different *operation selection* philosophies have been highly debated by Weyuker and Frankl in (Weyuker 1991, Frankl 1998). However, the most practiced *operation selection strategy* in industry is probably based on reaching an objective; in particular *code* and/or *requirement coverage* objective. In fact, the test engineer keeps selecting *operations* until the predefined objective is reached or her/his manager tells her or him to stop (for *time or budget* reasons). This strategy is clearly subjective and based on the test engineer's intuition and experience. Nevertheless, expert test engineers can perform a very good selection mechanism taking into account many factors such as the time and cost and the efficiency of the selected *operations* (Johnson Controls source). When using this *operation selection strategy*, the test engineer's skill (experienced and skilled) is the factor that mostly affects test effectiveness in finding bugs.

Many are the factors of relevance when *an operation selection strategy* has to be chosen. An important point to always keep in mind is that what makes a test a "good" one does not have a unique answer, but changes depending on the context, on the specific application, and on the goal for *testing*. The most common interpretation for "good" would be "able to detect many bugs"; but again precision would require to specify what kind of bugs, as Basanieri has shown in (Basanieri 2002) that different *operation selection strategy* detect different types of faults. Paradoxically, *operation selection* seems to be the least interesting problem for test practitioners. In 2006, we did a survey on existing commercial tools supporting the *operation selection* when *testing* a software product. We focus our survey on the tools able to select *operations* that verify the compliance of a software with its specification (*functional requirements*). We identify 6 tools:

1. **CONFORMIQ TEST GENERATOR** by *VERYSOFT* – GERMANY
2. **MATELO** by *ALL4TEC* – FRANCE
3. **PRO-TEST/PRAXIS** by *DIGITAL COMPUTATIONS, INC* – USA
4. **REACTIS** by *REACTIVE SYSTEMS, INC* – USA
5. **RHAPSODY TESTCONDUCTOR/AUTOMATIC TEST GENERATOR** by *I-LOGIX/TELELOGIC* – USA
6. **T-VEC RAVE/TESTER** for Simulink/Stateflow by *T-VEC* – USA

An overview of the characteristics of each of these tools is given in *Appendix D*. The major number of these tools is based on a *Model-Based* approach. Indeed, the software functional requirements are represented in a specific format from which operations are selected automatically. On the one hand, more than 278 tools supporting the *software testing* process (*test management*, *test execution* and so on) have been referenced in (Legeard 2007) by Legeard. On the other hand, we have shown through our *industrial audit* (Cf. *Chapter 2* –

Section 6) that the activity of designing manually *test cases* for software products becomes more and more laborious and time consuming. Therefore, one could highlight the lack of tools supporting the selection of operations when *testing* a software. 2% (6 over 278) of the commercial *software testing* tools are dedicated to the *software testing* activity that accounts more than 50% of the total software project time and budget.

Based on our industrial audit (Cf. *Diagnosis 2, 5, 6, 13, 14, 15, 16, 17 and 18*), the *operation selection strategy* currently used in Johnson Controls is a manual subjective one based the test engineers' experience and intuition. Their main purpose is to reach a code or *requirement coverage* objective. In fact, test engineers do not always select *operations* that simulate the *real use* of the software product under test. Moreover, there is no *formal* process to analyze recurrent bugs stored in the *problems' database* and select *operations* that detect these bugs on future developments. And finally, there is a lack of *formal* process and tools to manage and reuse *test cases* from one project to another. We propose to *formal*ize the process of selecting *operations* in order to be independent as much as possible from the test engineers' experience. One solution we propose is to automate this process. One could select *operations* randomly or, select *operations* based on the *end-user behavior's profile* or the *experience feedback* (from bugs and *test cases* capitalized on similar projects in the past).

Since 2005 and as the design of *test cases* for software has reached the 50% of the total time and budget of a project in Johnson Controls, the automation of the *test case design process* became a hot topic. Therefore, inner software experts and managers have evaluated some of the previous listed tools (evaluation version of the tool). None of these tools are fully adapted to the Johnson Controls context. Many issues have been identified by the experts and managers: 1) these tools propose to represent the software requirements in a *formal* language which is not adapted to the automotive software context, 2) these tools do not propose a relevant stop testing criterion based on the *test case* quality and software project constraints (test cost and time), 3) some of these tools do not manage the reuse of capitalized bugs and *test cases* from one project to another ,4) some of these tools do not propose to generate *test cases* with a *end-user behavior's profile* and finally 5) the tool licenses and trainings are expensive.

1. Advantages and drawbacks of automating the design of test cases

The activity of designing *test cases* for a software product is a major activity in a software development life cycle. To cut down cost of manual *test case* design and to increase reliability of it, researchers and practitioners have tried to automate it. Many managers today expect *test design automation* to be a silver bullet; killing the problems of test scheduling, the costs of *testing*, defect reporting, and more. However, there are many factors to consider when planning for *test design automation*. It usually has broad impacts on the organization such as the skills needed to design and implement automated tests, automation tools, and automation environments. Development and maintenance of automated tests is quite different from manual tests. The job skills change, test approaches change, and testing itself changes when automation is installed. These impacts have positive and negative components that must be considered. *Automation* is only a means to help accomplish our task – *testing* a product. It may reduce staff involvement during testing, thus saving time relatively to manually designing *test cases*. But, *automatic test design* may generate a bunch of results that can take much more staff involvement for analysis, thus costing more than manual test design. Often the information obtained from *automatic test generation* is more cryptic and takes longer to analyze and isolate when bugs are discovered. In fact, successful test automation efforts don't focus on eliminating the test team, they focus on doing a more effective and efficient job of

testing with the human resources available. *Automatic test generation* can be incredibly effective, giving more *coverage*. It also provides us with opportunities for *testing* in ways impractical or impossible for manual *testing*. Indeed, *automatic test design* can generate millions of *test cases* limited only by the machine power and time available for running the tests. However, Black (Black 2000) notice that *automated testing* is a huge investment, one of the biggest that organizations make in testing. Tool licenses are often expensive. Engineers cannot use alone most of these tools and therefore training, consulting, and expert contractors can cost more than the tools themselves. Moreover, the test engineers could resist using an automation tool since they felt that their manual process worked fine. Effort must be invested in incorporating a new automation tool into the process. As we propose to automate the design of *test cases* within Johnson Controls, we must take into account all these considerations.

2. Design test cases based on end-user behavior's profile

We propose to define an *end-user (driver) behavior's profile* for each software under test. Therefore, when *testing* the software, one could select the *operations* or *succession of operations* recurrently performed on the software in real use. In fact, there is no better way to test a product other than testing it in the way that it will be used. The main work in this field is the one of Musa in (Musa 1993). Musa has proposed a process to define an *operational profile* for a system. This process involves one or more of the following five levels: *Client type list*, *User type list*, *System modes*, *Functional profile* and *Operational profile*. By customizing this process to our context, we only consider the *operational profile* level. In fact, we consider that a software product is dedicated to only one customer (carmaker) and a *client type list* is not necessary. *T*he *end-users* (driver) of an automotive software product can be classified regarding to criteria such job, climate, sexe, age, culture … In our research, we do not deal with these criteria and we consider a nominal *end-user behavior's* profile. Most software products have more than one *mode of operation* (*normal* mode, *factory* mode and *diagnostic* mode). However, the occurrence probability of the *normal* mode is about 99% (Johnson Controls source) and consequently we decide to ignore the other modes. The next step is to break *system modes* down into the functionalities. It needs, to create a *functionality list* in determining the occurrence probability of each functionality. The best source of data to determine occurrence probabilities is usage measurements, e.g. frequency measurements of the users *operations*, taken on the last release or on similar system. In our context, we don't have this type of information (since it is considered as confidential by the carmakers). Finally, for each functionality, a set of *operations* and *sucession of operations* is possible (Cf. *Chapter 2 – Section 6*). Therefore, for each functionality, experts must identify recurrent *operations* and *succession of operations* and therefore define occurrence probabilities. Barnaghan (Branaghan 1999) has developed the fundamentals of *usability testing*. In fact, the *usability testing techniques* are widely and often used in testing *Graphical User Interfaces* (*GUI*).

3. Design test cases based on experience feedback

We also propose to reuse capitalized bugs and *test cases* from one project to another. Therefore, when testing the software, one could use the *experience feedback* on bugs and *test cases* respectively detected and designed on similar software in the past. Presently, information on stored bugs is missing and/or irrelevant and reusing these bugs in order to avoid or detect similar problems on future developments remains a difficult problem (Cf. *Diagnosis 16*). Therefore, classifying stored bugs and identifying the recurrent type of bugs detected on a specific type of software could be useful. In the next section, we perform a

survey on the *bug classification models*. We propose to define a typology of bugs adapted to our context and useful to focus the design of *test cases* on recurrent type of bugs.

a. A survey on the software bugs classification models

In this section, we present one bug classification scheme proposed by *IEEE* standard and two industrial schemes: the *Hewlett-Packard Scheme* (*HP*) and the *Orthogonal Defect Classification Scheme* (*ODC*) developed by *IBM*. Classification is performed by assigning a set of measurement variables (*attributes*) to discrete values, which are selected, based from a predefined set of values (*attribute values*). Therefore, bug classification schemes can differ in the way different *attributes* or *attributes values* relate to each other.

Orthogonal Defect Classification (Chillarege 1992)

The *ODC* scheme has been developed by *IBM*. Since its definition, this classification has been adopted by more and more organizations. In a survey performed in 1999 by Paulk in (Paulk 2000), 14 out of 37 high-maturity software organizations (according to the *CMM* maturity model) used this scheme as quantitative analysis practice. The *attributes* of this scheme are organized according to two process steps:

- *Step OPEN*: when a bug has been detected and a bug report is opened in *the bug tracking system*.
- *Step CLOSE*: when the bug has been corrected and the bug report is closed.

Two interesting *attributes* was taken into account in this scheme: 1) the *attribute defect type* which captures the fix that was made to resolve the bug and 2) the *attribute trigger* which captures the reason why an error turns into a bug. The entire *ODC* scheme with *attribute* name, meaning and values is described by Chillarege in (Chillarege 1992).

Hewlett-Packard Scheme (Grady 1992)

The *HP* scheme was developed by *HP's Software Metrics Council* in 1986. This scheme is based on three descriptors for each bug:

- The *origin* – where was the bug introduced in the product
- The *type* of the bug
- The *mode* – whether information was *missing*, *unclear*, *wrong*, *changed* or *done in a better way*

The choice of an *attribute value* for the *attribute Origin* defines the possible set of *attributes* available for the *attribute Type.* The entire *HP* scheme with *attributes* and *attribute values* is developed by Grady in (Grady 1992).

IEEE Standard Classification for Software Anomalies (IEEE Std. 1044-1993)

The *IEEE* scheme was developed by the *Institute of Electrical and Electronics Engineers* (*IEEE*), the world's leading professional association for the advancement of technology. The different *attributes* of the scheme are organized according to a general bug classification process consisting of four steps:

- *First Step: Recognition* – the bug is found
- *Second Step: Investigation* – we identify issues and propose solutions
- *Third Step: Action* – we establish a plan of action to resolve the problem
- *Last Step: Disposition* – we complete all required resolution actions and long-term corrective actions

The entire *IEEE* scheme with *attributes* and *attribute values* is developed in (*IEEE* Std. 1044-1993).

b. Major aspects of a software bug model

The previous survey reveals several valuable elements to be take into account when designing a new *bug classification scheme. Bugs are inserted due to a particular reason into a particular piece of software at a particular point in time. The bugs are detected at a specific time and occasion by noting some sort of symptom and they are corrected in specific way.* Each of these aspects might be relevant for a specific measurement and analysis purpose. Mellor and Fenton in (Mellor 1992, Fenton 1996) have proposed a framework of bug key elements that capture on high-level aspects of a bug. Each of these key elements can be refined leading to many attributes that can be captured by means of measurement:

- *Location: where in the product?* The *location* of a bug describes where in the product the bug was detected. This attribute can also contain attribute values describing different high-level entities of the entire product (*Specification*, *Design*, *Code*, *Documentation* …).
- *Timing: when in the process phases, we introduce, detect and correct the bug?* The *timing* of a bug refers to process phase when the bug was created (*origin phase*), detected (*detection phase*) and corrected.
- *Symptom: what we observe when the bug occurred? Symptom* captures what was observed when the bug occurred or the activity revealing the bug. For instance, the *ODC* attribute *Trigger* captures the mechanism that allows a bug to occur. Under *symptom* it is also possible to classify what is observed during *diagnosis* or *inspection*. For instance, in *IEEE* classification scheme, the *attribute symptom* provides a classification of the symptom.
- *End result: what are the impacts of the bug on the company itself, on the customer, on the end-user? End result* describes the failure caused by the bug. For instance, in *ODC*, the *attribute impact* captures the impact of a bug on the customer (*performance, usability, instability* …).
- *Mechanism: in which activity and how, we introduce, detect and correct the bug? Mechanism* describes how the bug was created, detected and corrected. *Creation* describes activity that inserted bug into the system. *Detection* describes activity that was performed when the bug was detected (*code review*, *unit test* …). *Correction* refers to the steps taken to remove the bug.
- *Cause: What is the mistake that leads to the bug? Cause* describes the mistake leading to the bug. For instance, in (Mays 1990) the author uses *attributes valu*es like *Education*, *Oversight*, *Communication*, *Tools* and *Transcription* for an attribute *Cause*. In (Leszak 2000), the author uses different *attributes* capturing different kind of *causes*: *Human-related Causes* (lack of knowledge, communication problems …), *Project Causes* (time pressure, management mistake) and *Inspection Causes* (no or incomplete inspection, inadequate participation …).
- *Severity: what is the severity of the bug? Severity* describes the severity of a resulting or potential failure on the whole behavior of the product.
- *Cost: How much the bug cost the company? Cost* captures the time or effort to locate, isolate and correct an error.

Bug classification scheme often have problems including incomplete, ambiguous and overlapping *attributes* and *attribute values*. To prevent such problems, a *bug classification*

scheme needs to be well defined. Freimut in (Freimut 2001) has proposed a list of quality properties of a good *bug classification scheme*:

- *Orthogonal attributes and orthogonal attributes values:* This means that for a particular bug and for each *attribute* only one *attribute value* is appropriate. If the *attribute values* are not *orthogonal*, it may happen that two or more attribute values may fit so that the engineer has arbitrarily to decide which value to assign. This leads to inconsistent and unreliable data.
- *Complete attribute values:* The set of *attributes* value must be complete so that for all bugs an appropriate *attribute value* can be selected. If the set of values are not complete, engineer may decide not to classify the bug or select the nearest possible value.
- *Small number of attribute values:* The scheme must contain a small number of *attributes values*, as too large a number can make selection of the appropriate *attribute value* difficult and therefore unreliable.
- *Clear meaning and definition of attributes and attribute values:* The *attributes* and *attribute values* of the scheme need a clear definition. This definition has to be developed with all engineers who have to use the *attributes* and need an understanding of the *attribute*.

D. Research issue 4: How to assess the expected behavior of a software product?

An important component of *testing* is the *oracle*. Indeed, a test is meaningful only if it is possible to decide about its outcome ("OK" or "Not OK"). The difficulties inherent to this task, often oversimplified, had been early articulated by Weyuker in (Weyuker 1982). In much of the research literature on *software testing*, the availability of *oracles* is either explicitly or tacitly assumed, but applicable *oracles* are not described. The research literature on *test oracles* is a relatively small part of the research literature on *software testing*. Some older proposals (Panzl 1978, Chapman 1982) base their analysis either on the availability of pre-computed input/output pairs or on a previous version of the same program, which is presumed to be correct. The former hypothesis is usually too simplistic: being able to derive a significant set of input/output pairs would imply the capability of analyzing the product outcome. In the current industrial practice of *software testing*, the *oracle* is often a human being. Relying on a human to assess program behaviors has two evident drawbacks: *accuracy* and *cost*. While the human *"eyeball oracle"* has an advantage over more technical means in interpreting incomplete, natural-language specifications, humans are prone to error when assessing complex behaviors or detailed, precise specifications, and the *accuracy* of the *eyeball oracle* drops precipitously with increases in the number of test runs to be evaluated. Even if it were more dependable, the *eyeball oracle* is prohibitively expensive for large volumes of *test cases*, and so may become a limiting factor when other parts of testing are accelerated with automation. Therefore, *automated oracles* could be a well adapted solution to this problem. Baresi's (Baresi 2001) survey proposes approaches to automate the *test oracles*. In view of these considerations, it must be evident that the *oracle* might not always judge correctly. So the notion of relevance of an *oracle* is introduced to measure its *accuracy*. Bertolino in (Bertolino 1997) proposes to measure the *oracle* accuracy by the probability that the *oracle* rejects a test, given that it must reject it.

Based on our industrial audit (Cf. *Diagnosis 2, 5, 6, 10, 12 and 15*), the *oracle* currently used in Johnson Controls is a human being. In fact, after selecting an *operation* to be performed on a software, test engineers analyze the *source code* and/or the carmaker

requirements of this software in order to assess the expected values to be checked on some *output signals*. In fact, this assessment is based on the engineers' understanding of the code and/or requirements and may lead to errors. Moreover, as automotive software becomes more and more complex, this task becomes a laborious task and accounts for more than 50% of the total *time and budget* of a project. We propose to automate the assessment process of all the expected outputs values by developing a *simulation model* of the software functional requirements. In fact, test engineers could perform the selected *operation* on the *requirements model* and assess the output values automatically by simulating the model. Moreover, once developing a *simulation model* of the software requirements, one could *formal*ly measure the *requirement coverage*.

1. Modeling and simulation of software functional requirements

a. Types of software requirements

In software domain, several standards organizations (including the *IEEE*) have identified four categories of requirements:

- *Functional requirements* are the main customer requirements. They refer to the behavior of the product. For instance, in a *body controller product*[31] , the behavior of the *front wiper management* functionality is specified by a set of functional requirements.
- *Non functional requirements* are the interface requirements between functionalities and software performances in terms of *CPU* load and *memory* capacity. An example of *non functional requirements* can be the communication protocols.
- *GUI (Graphical User Interface)* requirements are the customer requirements related to user interfaces. This category of requirements is frequent in electronic display product.
- *Non technical requirements* include all organizational customer requirements. Confidentiality, return of experience, past defects reviews capitalization is examples of these requirements.

Johnson Controls has adopted this typology of requirements (Cf. *Definition 2.6*). As demonstrated in *Chapter 2 - Section 4,* the functional requirements account for more than 90% of the carmaker requirements related to the software domain. Therefore, through our research project, we focus on the software functional requirements and how one could verify the compliance of a software product with its functional requirements.

b. Formalisms in specifying the functional requirements of a software product

Both Dart and Brinkkemper in (Dart 1987, Brinkkemper 1990) propose same definitions of *informal*, *semi-formal* and *formal* specification:

- *Informal*: These techniques do not have complete sets of rules to constrain the models that can be created. *Natural language* (written text) and *unstructured pictures* are typical instances.
- *Semi-formal*: These techniques have a defined syntax. Typical instances are *diagrammatic techniques* with precise rules that specify conditions under which

[31] A body controller module is an automotive electronic module in charge of managing all electrical currents of a car

constructs are allowed and textual and graphical descriptions with limited checking facilities.

- *Formal*: These techniques have rigorously defined syntax and semantics. There is an underlying theoretical model against which a description expressed in a *mathematical notation* can be verified. *Simulation languages* are typical instances.

In *Table 4.1*, Duphy (Duphy 2000) propose a list of advantages and drawbacks for each of the *informal*, *semi-formal* and *formal formal*ism.

	Advantages	Drawbacks
Informal	Easy to be understand by all the project actors. No training but need to understand the writing rules.	Ambiguity. Incompleteness. Inconsistency. Imprecision. Cannot be easily automated.
Semi-formal	Graphical and abstract representation. Easy to be understood by all the project actors. Synthetic, structuring and intuitive representation. Modularity and reuse. Better traceability.	Lack in precision. Sometimes, notations are ambiguous. Difficult to be interpreted. Simulation not possible. No techniques to verify and validate the model. Code generation from these models are not reliable.
Formal	Precision. Abstraction levels. Formalism. Model verification and validation. Proof. Simulation is possible. Generation of code and test cases from the specification model. Avoid imprecision, ambiguities and contradictions in the customer requirements. Long term cost reduction.	Trainings are mandatory. If no trainings, cost and delay increase. Not easy to be understood . Used for critical software products. Lack in tools supporting formal methods. Rarely used in industry. Not integrated to the software development process.

Table 4.1 – Advantages and drawbacks of *informal*, *semi-formal* and *formal* specification languages (Duphy 2000)

In *Table 4.2*, Duphy (Duphy 2000) has evaluated these three *formal*isms based on four criteria: modelling precision, use, communication facility and training cost.

	Modelling precision	Use	Communication facility	Training cost
Informal	-	+++	+++	++
Semi-formal	+	++	++	+
Formal	+++	-	---	---

+++ very positive; ++ positive; + quite positive; - quite negative; -- negative; --- very negative

Table 4.2 – Evaluation of the *informal*, *semi-formal* and *formal* specification languages (Duphy 2000)

In *Table 4.3*, a classification of the specification languages is proposed by Fraser in (Fraser 1994).

Informal	Semi-formal	Formal
Natural Language Specifications (Although often used, published studies are less numerous)	**UML** Variations on Data/Control Flow Diagrams Entity-Relationship Diagrams DeMarco Gane and Sarson PSL/PSA SADT SERM IORL CORE SDL JSD	**Finite State Machines** **Statecharts** **Markov Chain** **Decision Tables** Petri Nets Executable Specifications GIST Refine VDM Anna Z CSP GIST Predicate-Transition-Nets

Table 4.3 – Classification of the specification languages (Fraser 1994)

In fact, we cannot talk in detail about all the specification languages. In (Davis 1988), the author discussed a variety of *informal*, *semi-formal* and *formal* languages useful for testing. In the *Section 4.D*, we propose to develop a *simulation model* of the software functional requirements in order to automate the test oracle and *formal*ize the measurement of the *requirement coverage*. Only *formal* languages could be used to simulate software functional requirements (Cf. *Table 4.1*). Therefore, we focus our research on the most useful (according to the literature) *formal* languages: *Finite State Machines* (*FSM*), *Statecharts*, *Markov Chains* and *Decision Tables* (*DT*). An *FSM* is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an input, the machine generates an output and changes state. Both the output and the new state are purely functions of the current state and the input. *FSMs* are applicable to any model that can be accurately described with a finite number (usually quite small) of specific states. Chow in (Chow 1978) was one of the earliest researchers addressing the use of *FSMs* to specify the behavior of a software. Now, there is work on *FSMs* in software engineering with varied tones, purposes, and audiences (Apfelbaum 1997, Robinson 1999, Liu 2000). *Statecharts*, extensions to *FSMs*, were proposed by Harel in (Harel 1987). Statecharts make it even easier to model complex real-time system behavior with less ambiguity. The extensions provide a notation and set of conventions that facilitate the hierarchical decomposition of *FSMs* and a mechanism for communication between concurrent *FSMs*. *Statecharts* are probably easier to read than *FSMs*, but they are also nontrivial to work with and require some training upfront. A sample of software requirements expressed using *Statecharts* has been proposed by Hong in (Hong 2000). *Markov chains* are stochastic models proposed by Kemeny in (Kemeny 1976). They are structurally similar to *FSM* and can be thought of as probabilistic automata. In fact, a probability is associated for each transition. The sum of the probabilities associated to the transitions that get out of the same state must be equal to 1. Many researchers worked on using the *Markov chains* in specifying the behavior of a software (Whittaker 1994, Walton 2000). Sometimes there is a need to describe the required external behavior of some aspect of a system when the *FSM* approach makes no sense. One simple solution is the *Decision Table* proposed by Moret in (Moret 1982). A *DT* is used to lay out in tabular form all possible situations on the inputs of a system and to specify which action to take on the outputs in each of these situations.

c. How to choose a language to specify the functional requirements of a software product?

Many researchers (Davis 1988, Sommerville 1997, El-Far 2001) state that there are no software specification languages today that fit all intents and purposes. In fact, for each context decisions need to be made as to what language (or collection of languages) is most suitable. No large-scale studies have been made to verify the claims of any particular language. However, in his paper (Davis 1988), Davis has identified five criteria to help choosing the most adapted specification language for a specific context: 1) understandable to non computer-oriented customers, 2) used as the main input for the development and validation teams, 3) automated checks for ambiguity, incompleteness, and inconsistency, 4) encourage the requirements engineer to think and write in terms of external product behaviour, not internal product components, and finally 5) provide a basis for automated *source code* and *test generation*.

Based on the study that we performed on the evolution of the *formal*isms used by carmakers to specify functional requirements related to software (Cf. *Chapter 2 – Section 4.A*), we underline the increase of *formal* languages and the decrease of *informal* and *semi-formal* languages. However, within the *formal* languages, there not a unique standard *formal*ism shared between carmakers and suppliers (70% of *FSMs* and *Statecharts* and 20% of *DTs*). In fact, for each project, the supplier has to adapt its processes to the *formal*ism used by the carmaker. Duphy (Duphy 2000) highlights many works that try to complete or combine *semi-formal* and *formal* languages in order to have the fully, consistent and reliable view of a software. Three categories of combination can be identified: 1) create a new *formal*ism based on the existing techniques concepts, 2) complete an existing technique with the aim of reducing its weakness and finally 3) use simultaneously several existing techniques and thus cumulate their advantages. In our context, we propose to develop a new formal (simulation) specification language better adapted to the Johnson Controls context. In fact, for each project, we propose to represent the software functional requirements of the carmaker into this language. In order to be able to represent all the carmaker requirements (now and in the future), it could be judicious to base our specification language on a combination between the *FSM* (*Statechart*) and the *DT* languages; the two languages that carmakers tend to use.

V. Conclusion

We believe that the importance placed on testing will increase as software's pervasiveness in everyday life increases. Our dependence on software, from driving cars to shopping on the Internet, will decrease users' tolerance of defective software. Although testing isn't the only *software engineering* practice to ensure quality software, it remains an essential component of the software development's life cycle. We focus our research on the design of efficient *test cases* for improving the quality of software products. In fact, we are interested in any organizational matter that has a positive influence onto the quality of the *test case design process*: *simulation platform*, *knowledge management*, *competency management* and *project management*.

In this chapter, we pinpointed the main progress in each of these fields when designing *test cases*. Many techniques and approaches have been developed and for each one, we identify the advantages and drawbacks to be used or adjusted to our context. As a conclusion, we propose a list of actions that could improve significantly the global performance of the Johnson Controls company:

- Perform *functional test* since the earlier testing stages. One has to verify the compliance of each software component (independently from its environment) with the carmaker requirements.
- *Formal*ize the measurement of the *requirement coverage*. To do this, one has to specify the requirements using a *formal* language. Moreover, we suggest integrating *project constraints* (test time and cost) in the decision to stop testing a software product.
- *Formal*ize the process of selecting *operations* in order to be independent as much as possible from the test engineers' experience. One solution is to automate this process. One could select operations randomly or, select operations based on the *end-user behavior's profile* or the *experience feedback* (from bugs and *test cases* capitalized on similar projects in the past).
- Automate the assessment process of all the expected outputs values by developing a *simulation model* of the software functional requirements. In fact, test engineers could perform the selected *operation* on the *requirements model* and assess the output values automatically by simulating the model. Moreover, once developing a *simulation model* of the software requirements, one could *formally* measure the *requirement coverage*.

Based on our proposals, in the following four chapters (*Chapter 5*, *6*, *7* and *8*), we start specifying our approach to improve the global performance of the Johnson Controls *V&V* activities. *Firstly*, we develop a new *simulation model* of the software functional requirements. *Secondly*, we provide methods and tools to *verify and validate* the *requirements model*. *Thirdly*, we propose to *monitor the generation of test cases* by *quality objectives* and *cost constraints*. *And finally*, we suggest *refining the operation space description* with the *driver behavior's profile*, *past bugs* and *test cases*.

PART III – A NEW APPROACH FOR DESIGNING EFFICIENT TEST CASES FOR A SOFTWARE PRODUCT

CHAPTER 5. MODELING AND SIMULATION OF SOFTWARE FUNCTIONAL REQUIREMENTS

I. Introduction

Ten years ago, *formal* methods were rarely used in automotive industry, contrarily to medical, avionics and railways industries. The main argument of automotive industry managers was the high cost of deploying and using *formal* methods. But, as automotive electronic products becomes more and more complex, automotive industry is required to start adapting existing *formal* methods to their context or developing new ones. Actually, the cost of *non-quality* (warranty and customer dissatisfaction) exceeds the cost of using *formal* methods. We still have to change the engineers' practices and even adapt the education in *software engineering* to the challenge of complex products. Now, in automotive industry, *semi-formal* and *formal* methods are more and more used to specify software functional requirements (Cf. *Diagnosis 3*). However, there is a lack of a standard *formal*ism shared between carmakers and suppliers. In fact, for each project, the supplier has to adapt its processes (test case design, requirement coverage measurement) to the *formal*ism used by the carmaker (Cf. *Chapter 2 – Section 6.B*). Most of the automotive electronic suppliers use the *SRS* (*Software Requirement Specification*) model (Cf. *Chapter 2 – Section 4.E*). This model is mainly used to organize by functionality and by type the carmakers' requirements related to software and to tag them.

In this chapter, we develop our new *formal* language to model software functional requirements (a *simulation model*). Advantages and drawbacks of using *formal* languages in modeling software functional requirements are summarized in *Section 2*. Our *formal* model to represent software functional requirements is developed in *Section 3*. We identify two types of software functional requirements in automotive industry. These types of requirements could be modeled using a *Decision Table element* or a *Finite State Machine element*. The simulation process of the requirements model is described in *Section 4*. Finally, a didactic case study is proposed in *Section 5* in order to better illustrate our requirements model.

In the following, we use the shortcut *"software specification"* to designate the *"software functional requirements specification"*.

II. Advantages and drawbacks of formal languages in modeling software functional requirements

Formal specification is a specification expressed in a language whose *vocabulary*, *syntax*, and *semantics* are *formal*ly defined, and which has a mathematical, usually *formal* logic and basis. In this dissertation, we adopt the definition of a *formal* specification language proposed by Wing in (Wing 1990).

Definition 5.1: Formal Specification Language (Wing 1990)

A formal specification language provides a formal method's mathematical basis. …. A formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification.

Carmakers consider different standards to express the software functional requirements of a given electronic module. Based on the study performed in *Chapter 2 – Section 4.A*, some carmakers still use *semi-formal* and *informal* methods, but most of them start using *formal* methods (*simulation model*s). Incompleteness and ambiguity are the main characteristics of *informal* and *semi-formal* methods (Cf. *Chapter 4 – Section 4.D.4*). More than 30% of the bugs detected on a software product are related to lacks in and incomprehension of software functional requirements (Johnson Controls source). In fact, when designing test cases for the validation test, test engineers should assess the expected values to be checked on the *output signals* of the software product under test (Cf. *Chapter 4 – Section 4.D*). Indeed, a test is

meaningful only if it is possible to decide about its outcome. In case of an *informal* or a *semi-formal* representation of functional requirements, the assessment of the expected values on *output signals* is always a human being (Cf. *Chapter 2 – Section 6D*, as in Johnson Controls). Relying on a human to assess program behaviors has two evident drawbacks: *accuracy* and *cost*. In fact, after choosing the future operation to be performed on the software product, test engineers analyze the customer requirements and identify the required behavior to be checked on the product. In case of large volumes of test cases or complex behaviors, the accuracy of the eyeball oracle drops precipitously. However, in case of a *formal* representation of functional requirements, the assessment of the expected values on *output signals* could be done automatically (by simulation). The use of *formal* specification methods is expected to lead to increased software quality and reliability.

Hall (Hall 1990) suggests that benefits of using *formal* specifications are obtainable without an increase in, and possibly in lowering, development costs. However, Sommerville (Sommerville 1997) indicates that *formal* specification methods have not been widely accepted in industrial software development. Nevertheless, a number of strategies have been proposed for incorporating *formal* specification methods into the software development process.

On the one hand, a variety of advantages has been attributed to the use of *formal* software specifications. These advantages include understanding of specifications, help in the verification of specifications and automatic generation of the source code and test cases. *Firstly*, according to Wing (Wing 1990), the *formal* specifications help crystallize the customer's vague ideas, and reveal or avoid contradictions, ambiguities, and incompleteness in the specifications. Sommerville (Sommerville 1997) highlights that, depending on the *formal* specification language used, it may be possible to animate (simulate) a *formal* system specification to provide a prototype system. The *simulation model* can be used by inner engineers and by end-users to gain further insights into the behavior of the specified system. *Secondly*, as *formal* specifications can be analyzed using mathematical operators, many researchers (Wing 1990, Kemmerer 1990, Fraser 1994) propose to use mathematical proof procedures to test (and prove) internal consistency and correctness of specifications. Furthermore, the completeness of the specifications can be checked in the sense that all enumerated options and combinations have been specified. *Thirdly*, from an implementation point of view, as the final problem solution -the implementation- will be in a *formal* language (e.g. programming language); it is easier to avoid misconceptions and ambiguities in crossing the divide from *formal* specifications to *formal* implementations. This raises the possibility of automatic code generation from *formal* specifications and therefore avoiding the manual and labor coding of the software. Moreover, *formal* specifications can be used as a guide to the test engineers of software components in identifying and generating automatically appropriate test cases. In our research project, we do not consider the code generation aspect. *In conclusion*, the use of *formal* methods can lead to higher-quality specifications, implementations and testing.

On the other hand, a number of reasons by various authors have been suggested to explain the lack of using *formal* methods in industrial contexts. *Firstly*, Leveson (Leveson 1990) pinpoints the lack of methodological and support tool in *formal* specification research which makes it difficult to develop, analyze, and process large-scale specifications using *formal* specification languages. *Secondly*, Sommerville (Sommerville 1997) highlights that the notation and the conceptual grammar of *formal* specification languages require familiarity with discrete mathematics and symbolic logic which most practicing software engineers do not currently have. *Thirdly*, the very *formal*ity which makes *formal* specifications desirable during the later phases of software specification makes them an inappropriate tool for

communicating with the end-user during the earlier requirement elicitation and confirmation stages. *Finally*, Sommerville (Sommerville 1997) suggests that management is generally conservative and unwilling to use new techniques whose benefits are not yet established. Given these difficulties in using *formal* methods, challenges remain in integrating *formal* methods with the system development effort and in scaling up *formal* method techniques to large-scale real-world development projects.

In *Table 5.1*, we summarize the main advantages and drawbacks of *formal* languages in modeling/specifying software functional requirements.

Advantages	Drawbacks
Avoid contradictions, ambiguities, and incompleteness in the software specifications	Lack of methodological and support tool
Automatically Check consistency, correctness and completeness of software specifications	Lack of familiarity with discrete mathematics and symbolic logic that most practicing software engineers do not currently have
Automatically generate code for software product	Inappropriate tool for communicating with the end user during the earlier requirements elicitation and confirmation stages
Automatically generate test cases for software functional testing	Need to verify and validate the developed model. Indeed, we have to proove the conformity between carmaker requirements and the developed model
Reduce development cost and time	
Easy maintenance	

Table 5.1 – Advantages and drawbacks of formal languages in modeling/specifying software functional requirements

Unfortunately, within the *formal* languages currently used in automotive industry, there is not a unique *formal*ism shared between carmakers and suppliers (Cf. *Diagnosis 3*). In *Chapter 4 – Section 4.D.4*, we pinpoint the benefits of a unified *formal* (simulation) language able to model all types of software functional requirements.

III. Our formal language to model software functional requirements for functional simulation

Nowadays, and according to Davis and El-Far (Davis 1988, El-Far 2001), an international unified model to specify and simulate software functional requirements doesn't exist. After studying a variety of models in literature, we came up with the fact that each model has been developed for a specific industrial or academic context. Based on the study that we performed on the evolution of the *formal*isms used by carmakers to specify functional requirements related to software (Cf. *Chapter 2 – Section 4.A*), we underline that, within the *formal* languages, there is not a unique standard *formal*ism shared between carmakers and suppliers (70% of *FSMs* and *Statecharts* and 20% of *DTs*).

In our research project, we define our own *formal* model, to represent software functional requirement, keeping in mind the automotive context and its constraints. As defined before (Cf. *Definition 2.4*, *2.5* and *2.6*), a *software functionality* is described by some *features* that are described by some *requirements*. In the following, we do not consider the non-functional requirements and we focus our research on modeling software functional requirements.

A. Typology of software functional requirements

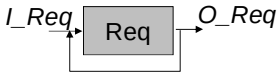
Each software functionality has a set of *configuration (Config)*, *input (I)*, *output (O)* and *intermediate (Int) signals* with discrete domains. These signals interconnect the *features (F)* of the functionality and each *feature* is composed from one or more requirements of the same type. Based on our study of the carmakers' requirements related to software (Cf. *Chapter 2 – Section 4.A*) and the literature review on modeling software specifications (Davis 1988, Apfelbaum 1997, Robinson 1999), we identify two types of software functional requirements:

- *combinatorial* (Cf. *Figure 5.1*) if the values of the requirement *output signals* at instant *t (O_Req$_t$)* depend on the sole values of the requirement *input signals* at instant *t (I_Req$_t$)*.

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

Figure 5.1 – "Combinatorial" functional requirement

- *Sequential* (Cf. *Figure 5.2*) if 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 5.2 – "Sequential" functional requirement

In *Figure 5.3*, we provide a graphical illustration of our unified functional requirements model. This example is the functional requirements model of a software functionality which has 1 *configuration signal*, 3 *input signals*, 4 *output signals*, 5 *intermediate signals* and 4 *features*. *Configuration signals* allow to parameterize the software functionality (for instance, by activating or deactivating one *feature*). *Input signals* could be switches, sensors or car environment variables (for instance, the *vehicle speed signal*). *Output signals* could be actuators or any type of command (for instance, the *wiper motor command signal*). Finally, *intermediate signals* allow to manage and share data between two or more *features*.

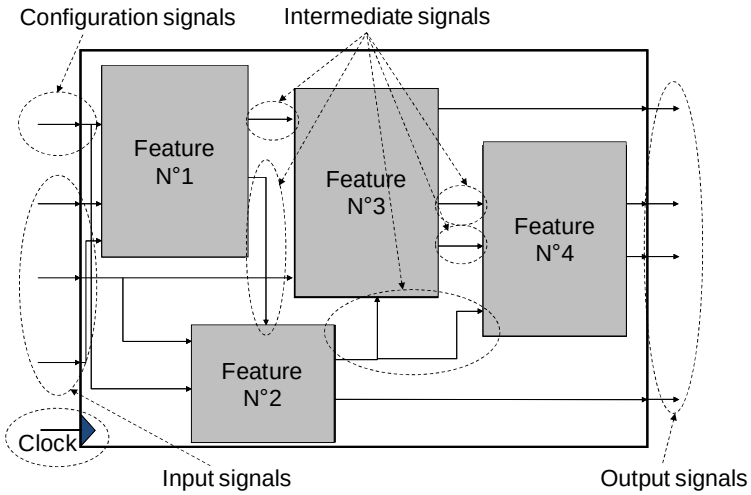


Figure 5.3 – Graphical illustration of our unified formal model to represent software functional requirements

A "*Clock*" signal (Cf. *Figure 5.4*) is required since the behavior of a software product is ruled by synchronism. In fact, a "*Clock*" is just a signal that alternates between zero and one, back and forth, at a specific *pace* (*cycle time*). It sets the "*pace*" for the functional simulation of the model. The value of this "*cycle time*" depends on some timing characteristics of the software functional requirements. It should be defined by the *modeler* once analyzing and designing the requirements model.

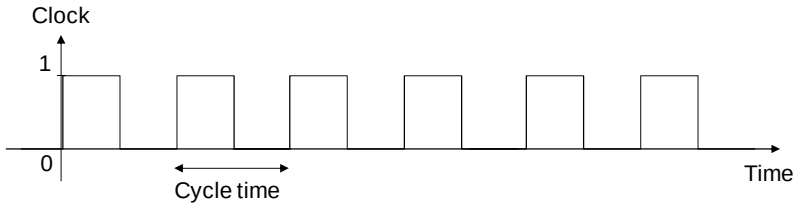


Figure 5.4 – The shape of a "Clock" signal

B. Two types of modeling elements to model the features of a software functionality

As we stated before, each *feature* is composed from one or more functional requirements of the same type (*combinatorial* or sequential). We propose to model these two types of functional requirements thanks to two types of modeling *elements*.

1. Decision table element (DT)

Moret and Chvalovsky (Moret 1982, Chvalovsky 1983) were the first to thoroughly explore the uses and capabilities of *DT*. We use a *DT element* to model a *feature* composed from one or more *combinatorial* functional requirements. A *DT* is a table (Cf. *Figure 5.7*) that presents a set of exclusive *conditions* on the *DT input signals (Cq)* and their corresponding set of *actions* on the *DT output signals (Aq)*. Each set of *conditions (Cq)* represents a requirement in a *DT element*. The characteristics of a "*condition*" and an "*action*" on a signal (S_i) are respectively illustrated in *Figure 5.5* and *Figure 5.6*.

The structure of a "Condition" is organized as following:

```
Condition(Operator Op, String Sig, float Val)
  {
     Operator = Op; // ANY, EQUAL, NEQUAL, GREATER, LESS, GREATER_EQ, LESS_EQ
     Signal = Sig;
     Value = Val;
  }
```

Rules of defining a "Condition" on a signal S_i:

1- When **Operator** is set to *ANY*, then **Signal** must be set to *" "* and **Value** must be set to *0*

2- When **Operator** is different from *ANY* and **Signal** is different from *" "*, then **Value** must be set to *0*

Examples:

S_1: Condition(ANY, " ", 0) ↔ no matter the value of the signal S_1

S_2: Condition(LESS, "S_1", 0) ↔ if the value of the signal S_2 is LESS than the value of the signal S_1

S_3: Condition(EQUAL, " ", 10) ↔ if the value of the signal S_3 is equal to 10

S_4: Condition(GREATER_EQ, " ", 5) ↔ if the value of the signal S_4 is GREATER than or EQUAL to 5

Figure 5.5 – Characteristics of a "Condition"

The structure of a "Action" is organized as following:

```
Action (GlobalOperator GOp, float Val1, float Val2, Operator Op, String Sig1, String Sig2)
  {
     GlobalOperator = GOp; // UNCHANGE, EQUAL
     Signal1 = Sig1;
     Signal2 = Sig2;
     Operator = Op; // NONE, ADD, SOUS, DIV, MULT
     Value1 = Val1;
     Value2 = Val2;
  }
```

Rules of defining an "Action" on a signal S_i:

1- When **GlobalOperator** is set to *UNCHANGE*, then **Signal1** and **Signal2** must be set to " ", **Operator** must be set to *NONE* and **Value1** and **Value2** must be set to *0*

2- When **GlobalOperator** is set to *EQUAL* and **Operator** is set to *NONE*, then **Signal2** must be set to " " and **Value2** must be set to *0*. And if **Signal1** is different from " ", then **Value1** must be set to *0*

3- When **GlobalOperator** is set to *EQUAL*, **Operator** is different from *NONE* and **Signal1** and **Signal2** are different from " ", then **Value1** and **Value2** must be set to *0*

4- When **GlobalOperator** is set to *EQUAL* and **Operator** is different from *NONE*, **Signal1** is different from " " and **Signal2** is equal to " ", then **Value2** must be set to *0*

5- When **Operator** is equal to *DIV* and **Signal2** is different from " ", then the value of the **Signal2** must be different from *0*

6- When **Operator** is equal to *DIV* and **Signal2** is equal to " ", then **Value2** must be different from *0*

Examples:

S_1: Action(UNCHANGE, " ", " ", NONE, 0, 0) ↔ no actions to do on S_1

S_2: Action(EQUAL, "S_1", " ", NONE, 0, 0) ↔ S_2 must be set to the value of the signal S_1

S_3: Action(EQUAL, " ", " ", NONE, 5, 0) ↔ S_3 must be set to 5

S_4: Action(EQUAL, "S_2", "S_3", ADD, 0, 0) ↔ S_4 must be set to the value of (S_2 + S_3)

S_5: Action(EQUAL, "S_4", " ", SOUS, 5, 0) ↔ S_5 must be set to the value of (S_4 – 5)

S_6: Action(EQUAL, " ", " ", MULT, 5, 10) ↔ S_6 must be set to the value of (5 x 10)

S_7: Action(EQUAL, "S_5", "S_6", DIV, 0, 0) ↔ S_7 must be set to the value of (S_5 / S_6), with $S_6 \neq 0$

S_8: Action(EQUAL, "S_7", " ", DIV, 3, 0) ↔ S_8 must be set to the value of (S_7 / 3)

Figure 5.6 – Characteristics of an "Action"

As said before, each software functionality has a set of *configuration (Config), input (I), output (O)* and *intermediate (Int) signals*. These signals interconnect the *features (F)* of the functionality. In fact, an input signal of a *Decision Table element* could be a *configuration, input* or *intermediate signal* of the functionality. While an output signal of a *Decision Table element* could be an *output* or *intermediate signal* of the functionality. A *Decision Table element* is illustrated in *Figure 5.7*, a. For one set of *conditions* (for example, *C1* in *Figure 5.7*), it must require that at least one input of the *DT* is set to a specific value (*I1=1*), the other inputs of the *DT* may be indifferent (*ANY*).

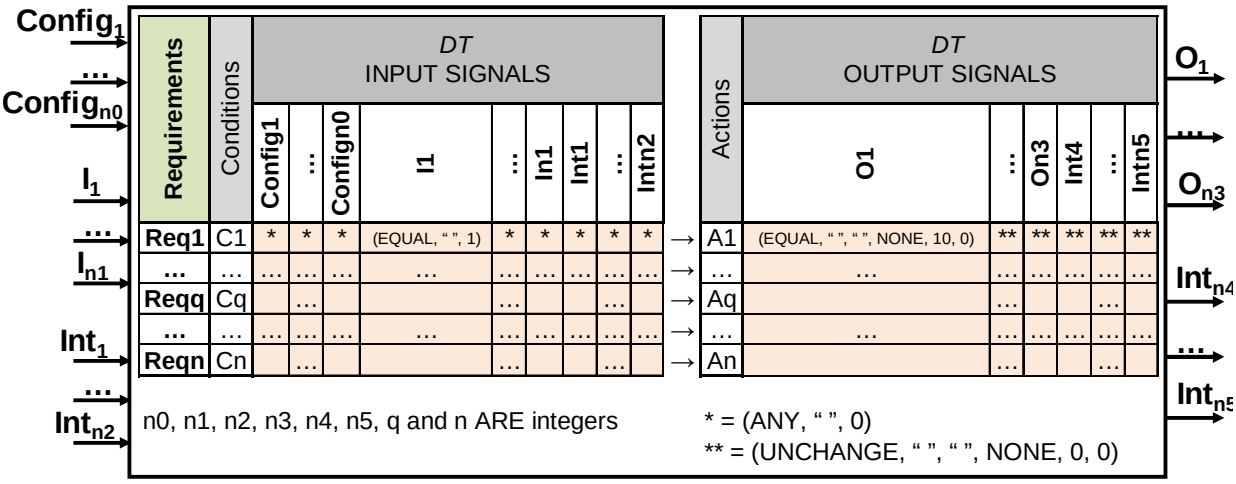


Figure 5.7 – A Decision Table element

Let us consider the *DT element* illustrated in *Figure 5.8*. This *DT* has 2 *input signals* and 2 *output signals*: *I1*, Domain = {0, 1}; *I2*, Domain = {1, 2, 3}; *O1*, Domain = {0, 1}; *O2*, Domain = {0, 1}. When designing this *DT element*, designers did not consider all the possible *conditions* on the *input signals* of a *DT* (3 out of 6 possible conditions, Cf. *Figure 5.8a*). They only identify the *conditions (Ci)* which were explicitly specified in the customer requirements. In fact, when dealing with a small *DT*, all the possible *conditions* can be easily identified. In *Figure 5.8b*, we illustrate the *exhaustive DT* of the one of *Figure 5.8a*. On the one hand, a *condition* could be splitted into 2 or more *conditions* with the same *actions* on the *output signals* (*C1* to *C1.1*, *C1.2* and *C1.3*). On the other hand, some *conditions* do not have any impact on the *output signals* (*C4*, *UNCHANGE*). However, in industrial context, the problem is a little bit more difficult since the number of the *DT input signals* can exceed 10 and the domain length of one signal can exceed 100 (for instance, when sampling the "vehicle speed" signal). In that case, its remains a very difficult task to identify manually all the possible *conditions* and their corresponding *actions*. Therefore, an automatic generation of all the possible *conditions* on the *input signals* of a *DT* could be judicious. One could develop a *computer macro* able to generate automatically an exhaustive list of *conditions* for a *DT*. Unfortunately, we do not have enough time to develop this *macro* and in our experiments (Cf. *Chapter 10*), we design manually exhaustive *DTs*.

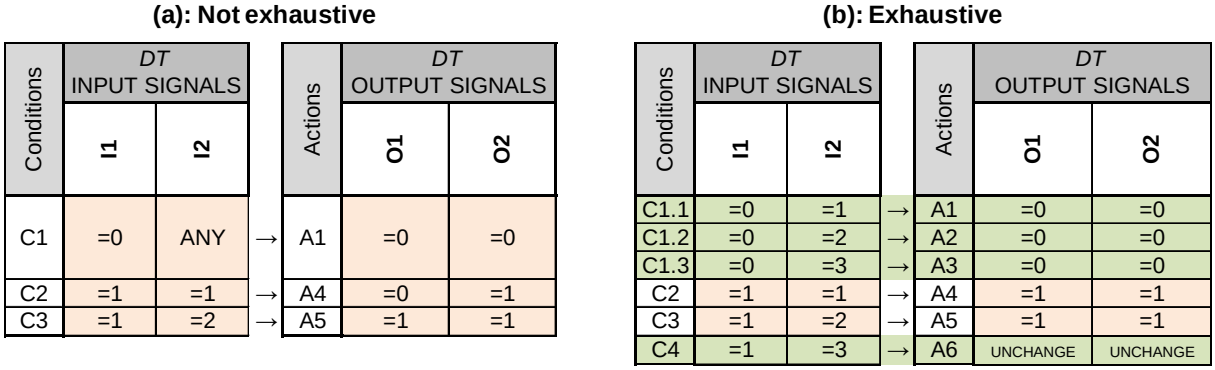


Figure 5.8 – Not exhaustive vs. exhaustive Decision Table element

2. Finite State Machine element (FSM)

Gill (Gill 1962) introduces *FSM* theory in 60's. Since, many applications (Chow 1978) such as in software engineering have been performed. We use a *FSM element* to model a *feature* composed form one or more sequential functional requirements. In our case and in addition to

the input and *output signals* of a *FSM*, each *FSM* can have a timing signal *(FSMTempo)* and a set of internal signals *(FSMInt$_m$)*. The timing signal helps to model timing requirements and the internal signals characterize the *states* of a *FSM*. A graphical illustration of a *Finite State Machine element* is illustrated in *Figure 5.9*. It is composed from:

- an *initial state (S0)* and a finite number of *states (Si)* with a set of *actions (Ai)* on the *FSM* output, internal and timing signals. The *FSM* timing signal is set to 0 each time the *state* of the *FSM* changes. In fact, the *FSM* timing signal computes the time spent in each *state*.
- a set of *transitions (Tij)* from original *state (Si)* to destination *state (Sj)*, and for each *transition (Tij)*, a set of exclusive *conditions (Cij,q)* on the *FSM* input, internal and timing signals. Each set of *conditions (Cij,q)* represents a requirement in a *FSM element*.

For one set of *conditions* (for example, *Cij,1* in *Figure 5.9*), it must require that at least one *input signal* of the *FSM* or one *FSM* internal signal or the *FSM* timing signal is set to a specific value (*I1=1*), the other signals of the *FSM* may be indifferent (*ANY*).

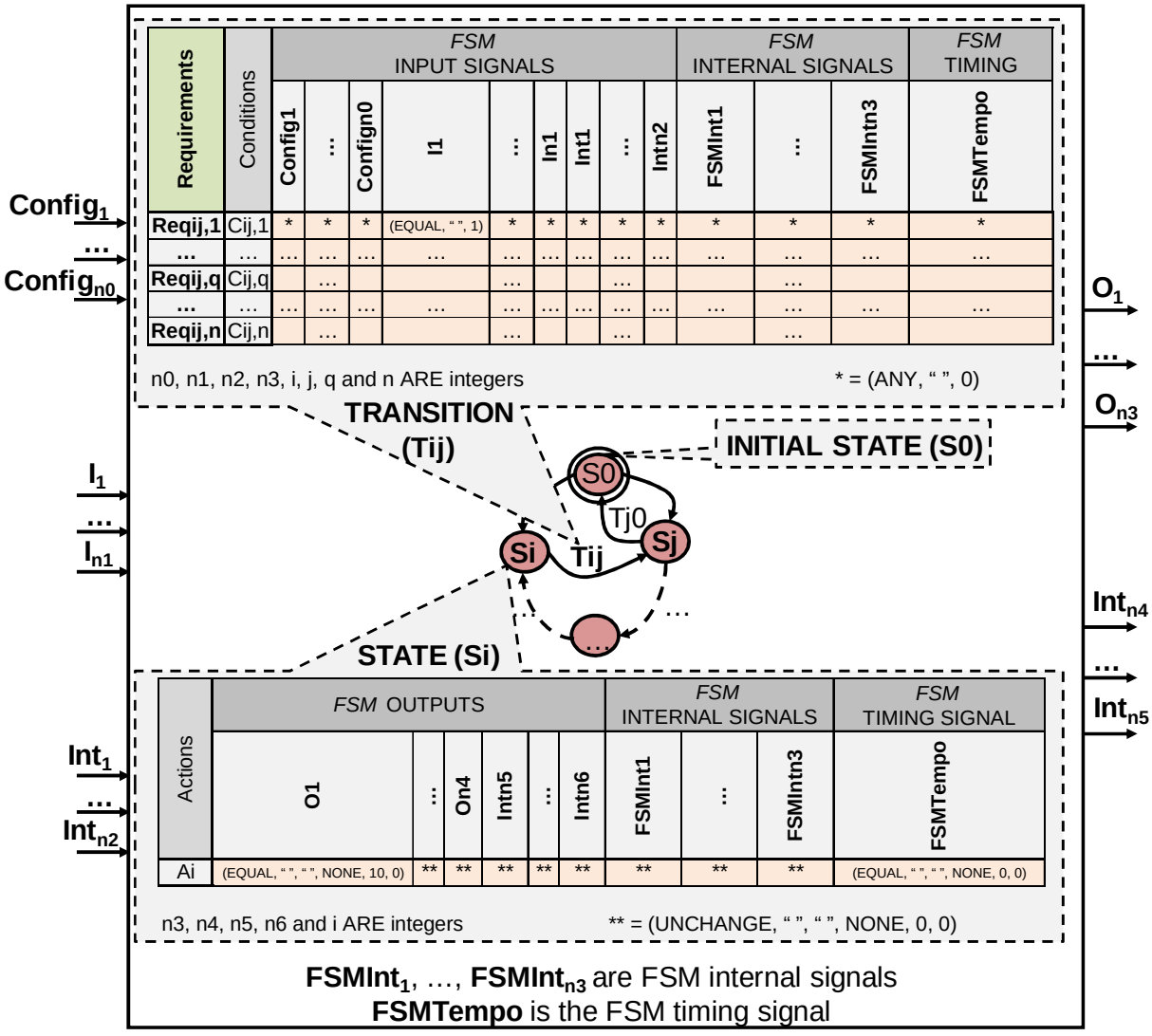
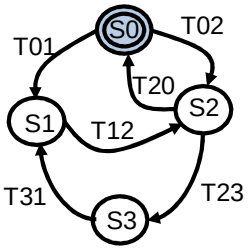


Figure 5.9 – A graphical illustration of a Finite State Machine element

Let us consider the *FSM element* illustrated in *Figure 5.10*. This *FSM* has 2 *input signals*, 2 *output signals*: *I1*, Domain = {0, 1}; *I2*, Domain = {1, 2, 3}; *O1*, Domain = {0, 1}; *O2*, Domain = {0, 1}. When designing this *FSM element*, designers did not consider all the

transitions and *conditions*. They only consider the *transitions (Tij)* and *conditions (Ci)* which were explicitly specified in the customer requirements (6 out of 8 *transitions* and 7 out of 9 *conditions*, Cf. *Figure 5.10a*). In order to be exhaustive when designing a *FSM*, *modelers* must identify all the *transitions* and *conditions* that get out of a *state* even if they do not allow to change the *state* of the *FSM* (Cf. *Figure 5.10b*).

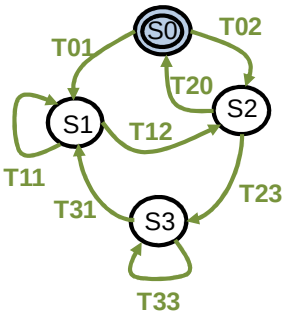
(a): Not exhaustive



States	Actions	*FSM* OUTPUT SIGNALS	
		O1	**O2**
S0	A0	=0	=0
S1	A1	=0	=1
S2	A2	=1	=0
S3	A3	=1	=1

Transitions	Conditions	*FSM* INPUT SIGNALS	
		I1	**I2**
T01	C1	=1	ANY
T02	C1	=0	ANY
T12	C1	=0	ANY
T20	C1	ANY	=3
T23	C1	ANY	=1
	C2	ANY	=2
T31	C1	=1	ANY

(b): Exhaustive



Transitions	Conditions	*FSM* INPUT SIGNALS	
		I1	**I2**
T01	C1.1	=1	=1
	C1.2		=2
	C1.3		=3
T02	C1.1	=0	=1
	C1.2		=2
	C1.3		=3
T11	C1.1	=1	=1
	C1.2		=2
	C1.3		=3
T12	C1.1	=0	=1
	C1.2		=2
	C1.3		=3
T20	C1.1	=0	=3
	C1.2	=1	
T23	C1.1	=0	=1
	C1.2	=1	
	C2.1	=0	=2
	C2.2	=1	
T33	C1.1	=0	=1
	C1.2		=2
	C1.3		=3
T31	C1.1	=1	=1
	C1.2		=2
	C1.3		=3

Figure 5.10 – Not exhaustive vs. exhaustive Finite State Machine element

IV. The functional simulation process of our software requirements model

A synchronized functional simulation can be performed on our model of software functional requirements. The simulation is done with an oriented acyclic logic going from input to *output signals* of the software functionality. To better illustrate the simulation mechanism, let us consider the example of the *Figure 5.3*. The simulation order of the *features* has to be defined when designing the model (*Feature 1* then *Feature 2* then *Feature 3* then *Feature 4*). The "*Clock*" signal synchronizes the behavior of the functional model. Indeed, at each *cycle time*, all the *features* are simulated following the predefined order. Simulating a *feature* consists of assessing its *output signals* values according to its *input signals* values.

In case of a feature modeled using a Decision Table element, all *conditions (Cq)* have to be checked. There is no specific checking order for these *conditions* since up to one *condition* can be fulfilled at a time. Values on the *DT output signals* are updated according to the *action* associated to the satisfied *condition*. Note that, in some cases, none of the *conditions (Cq)* can be fulfilled and therefore no *actions (Aq)* have to be done on the *DT output signals*. In fact, the *DT conditions* do not often consider all the possible combinations between the values of all the *DT input signals*. Let us consider the *DT element* of the *Figure 5.8a*. In *Figure 5.11*, a simulation scenario of this *DT* is shown.

- **(*Figure 5.11a*):** *After initialization*, *I1* is set to *0* and *I2* is set to *2*. On the next front edge of the "*Clock*" signal, all the *conditions (Ci)* are checked following the predefined order. Once a set of *conditions* is satisfied *(C1)*, the corresponding *actions (A1)* on the *DT output signals* are performed and the *conditions* checking is stopped.
- **(*Figure 5.11b*):** *I1* is set to *1*. On the next front edge of the "*Clock*", *C3* is satisfied.
- **(*Figure 5.11c*):** *I2* is set to *3*. On the next front edge of the "*Clock*", none of the *conditions* is fulfilled.

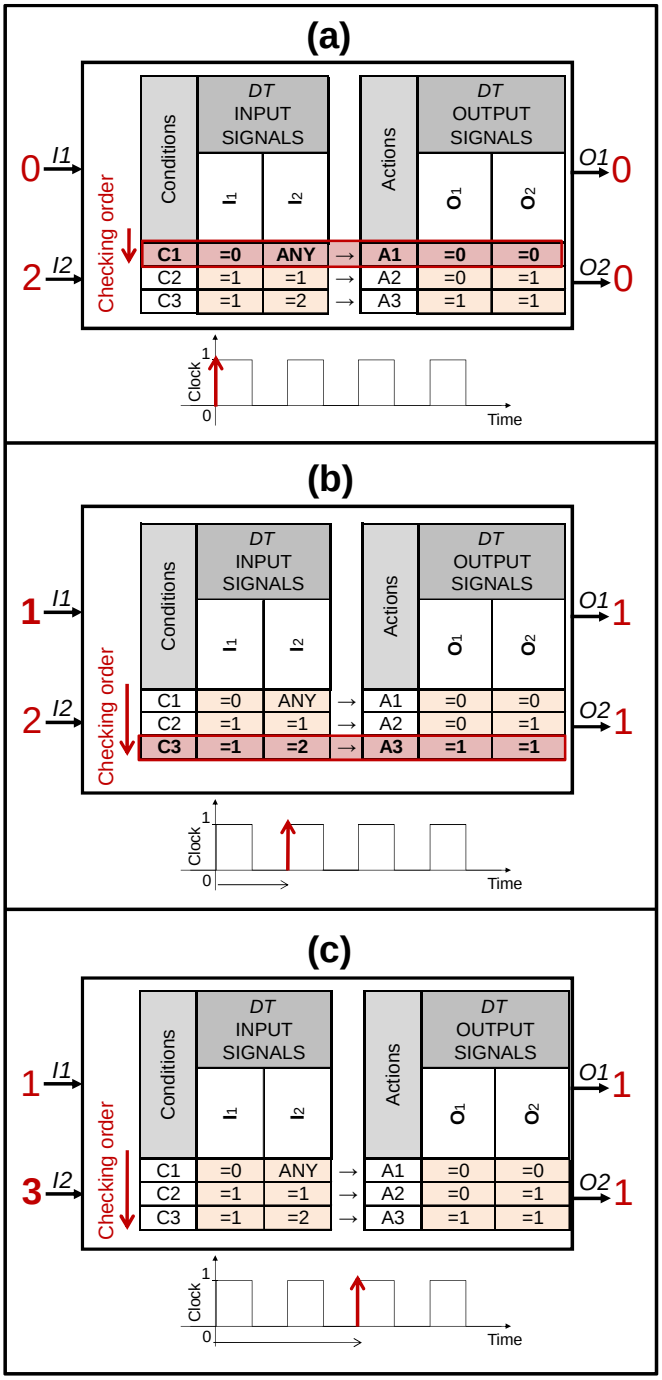


Figure 5.11 – An example to illustrate the simulation process of a Decision Table element

In case of a feature modeled using a Finite State Machine element, one *state* must always be activated. When simulating a *FSM*, all *conditions* of all the *transitions* that get out of the activated *state* have to be checked. There is no specific checking order for *transitions* and *conditions* since they are exclusive and up to one *condition* (*transition*) can be satisfied (crossed) at a time. Therefore, after each *FSM* simulation, at maximum one *transition* is crossed. The origin *state* of the *transition* is deactivated, the destination *state* is activated and values on *output signals* are updated. However, in some cases, none of the *transitions* that get out of the activated *state* can be satisfied and therefore the activated *state* remains the same and no *actions* have to be done on the *FSM output signals*. In fact, the *conditions* of all the *transitions* that get out of the same *state* do not often consider all the possible combinations

between the values of all the *FSM* input, internal and timing signals. Let us consider the *FSM element* of the *Figure 5.10a*. In *Figure 5.12*, a simulation scenario of this *FSM* is shown.

(*Figure 5.12a*): *After initialization*, *I1* is set to *1* and *I2* is set to *1*. On the next front edge of the "*Clock*" signal, all the *conditions (Ci)* on all the *transitions (T0j)* that get out of the activated *state* (S0) are checked following the predefined order. Once a set of *conditions* is satisfied *(T01, C1)*, the corresponding *transitions (T01)* is crossed, the origin *state (S0)* is deactivated, the destination *state (S1)* is activated and the *action (A1)* on the destination *state* is performed.

(*Figure 5.12b*): the activated *state* is *S1*. *I2* is set to *2*. On the next front edge of the "*Clock*", all the *conditions (Ci)* on all the *transitions (T1j)* that get out of the activated *state (S1)* are checked following the predefined order. Since, none of these *conditions* is satisfied, the activated *state* does not change *(S1)* and the values on the *FSM output signals* no more.

(*Figure 5.12c*): the activated *state* is *S1*. *I1* is set to *0*. On the next front edge of the "*Clock*", the *transition (T12)* is crossed. The new activated *state* is *(S2).*

(*Figure 5.12d*): the activated *state* is *S2*. *I2* is set to *1*. On the next front edge of the "*Clock*", the *transition (T23)* is crossed. The new activated *state* is *(S3).*

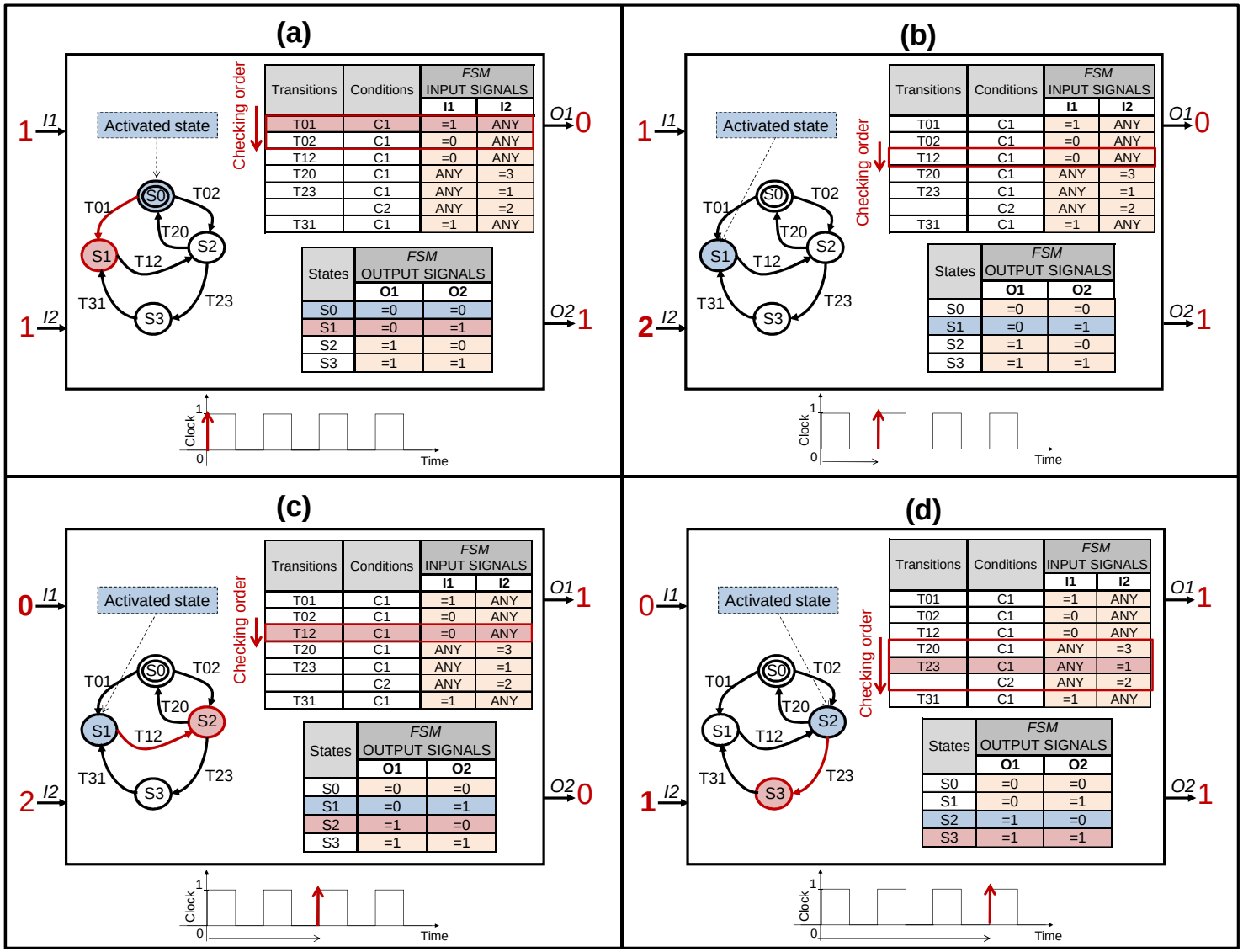


Figure 5.12 – An example to illustrate the simulation process of a Finite State Machine element

V. A case study on modeling software functional requirements using our new formal and simulation language

In this section, we develop a simple case study in order to illustrate how an engineer can design a *simulation model* of a set of software functional requirements. To do this, we consider a functionality ("Auto_Light") which has 3 configuration signals, 5 *input signals* and 2 *output signals*:

- *Config1* ("Auto_Light_Config"), Domain = {0, 1}
- *Config2* ("Follow_Me_home_Config"), Domain = {0, 1}
- *Config3* ("Follow_Me_home_Calib"), Domain = {5, 10, 15}
- *I1* ("Reset"), Domain = {0, 1}
- *I2* ("Luminosity_Sensor"), Domain = {0, 1, 2, 3, 4, 5, 6, 7}
- *I3* ("Car_Locked"), Domain = {0, 1}
- *I4* ("Ignition"), Domain = {0, 1}
- *I5* ("Light_Combi_Switch"), Domain = {0, 1}
- *O1* ("Head_Lamp"), Domain = {0, 1}
- *O2* ("Tail_Lamp"), Domain = {0, 1}

The software functional requirements of this functionality were specified by the carmaker using the natural language (Cf. *Figure 5.13*). In fact, this functionality can be decomposed into 3 *features*: *Feature 1*, *Feature 2* and *Feature 3*.

Feature 1 ("Luminosity_Level_Calculation")
Req1.1: In case of "Luminosity_Sensor" is equal to 1 or 2, then "Luminosity_Level" must be equal to 1
Req1.2: In case of "Luminosity_Sensor" is equal to 3, 4 or 5, then "Luminosity_Level" must be equal to 2
Req1.3: In case of "Luminosity_Sensor" is equal to 6 or 7, then "Luminosity_Level" must be equal to 3
Req1.4: In other cases, "Luminosity_Level" must be equal to 0
Feature 2 ("Follow_Me_Home_Mode")
Req2.1: In case of "Car_Locked" is equal to 1 and "Ignition" is equal to 0, then "Follow_Me_Home_Activate" must be equal to 1
Req2.2: In other cases, "Follow_Me_Home_Activate" must be equal to 0
Feature 3 ("Head_Tail_Activation")
Req3.1: Once "Reset" is set to 1, "Head_Lamp" and "Tail_Lamp" must be set at 0
Req3.2: In case of "Ignition" is equal to 1 and "Light_Combi_Switch" is equal to 1 and "Auto_Light_Config" is equal to 1 and "Luminosity_Level" is equal 1 or 2, then "Head_Lamp" must be equal to 0 and "Tail_Lamp" must be equal to 1
Req3.3: In case of "Ignition" is equal to 1 and "Light_Combi_Switch" is equal to 1 and "Auto_Light_Config" is equal to 1 and "Luminosity_Level" is equal 3, then "Head_Lamp" must be equal to 1 and "Tail_Lamp" must be equal to 0
Req3.4: If "Head_Lamp" is equal to 1 or "Tail_Lamp" is equal to 1 and in case of "Follow_Me_Home_Activate" is equal to 1 and "Follow_Me_home_Config" is equal to 1, then "Head_Lamp" must be equal to 0 and "Tail_Lamp" must be equal to 1.
If "Ignition" is set to 0, then ***Req3.5*** and ***Req3.6***
Req3.5: In case of "Head_Lamp" was equal to 1, wait "Follow_Me_home_Calib" ms and then set "Head_Lamp" and "Tail_Lamp" to 0
Req3.6: In case of "Head_Lamp" was equal to 0, than wait "Follow_Me_home_Calib"/2 ms and then set "Head_Lamp" and "Tail_Lamp" to 0

Figure 5.13 – The software functional requirements of the functionality "Auto_Light" as they were specified by the carmaker

Once analyzing the requirements of *Figure V13*, we came up to the conclusion that *Feature 1* and *Feature 2* can be modeled using *Decision Table elements* and *Feature 3* can be modeled using a *Finite State Machine element*. We also identified two *intermediate signals*: *Int1* ("Luminosity_Level") and *Int2* ("Follow_Me_Home_Activate"). A graphical illustration of the requirements model of the functionality "Auto_Light" is developed in *Figure 5.14*.

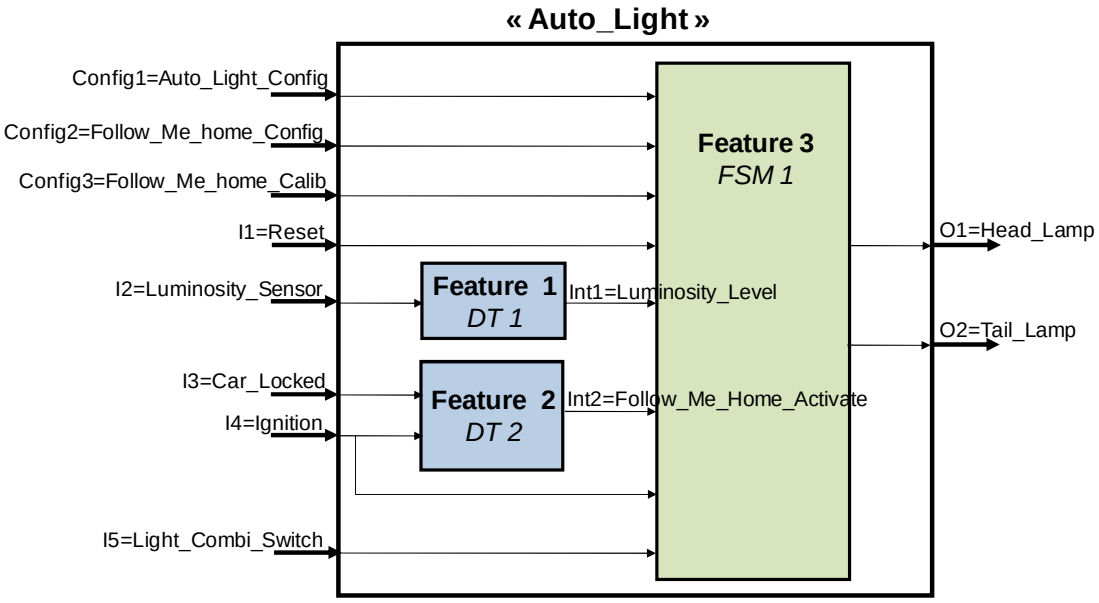


Figure 5.14 – A graphical illustration of the requirements model of the functionality "Auto_Light"

The *Decision Table elements* of the *Features 1* and *2* are developed in *Figure 5.15* and *Figure 5.16*. These *DT* are exhaustive.

Feature 1 - *DT 1*

I2= Luminosity_Sensor

Carmaker req ID	Our model req ID	Conditions	*DT1* INPUTS	Actions	*DT1* OUTPUTS
			I2		**Int1**
Req1.4	Req1.4	C1	=0	A1	=0
Req1.1	Req1.1.1	C2	=1	A2	=1
Req1.1	Req1.1.2	C3	=2	A3	=1
Req1.2	Req1.2.1	C4	=3	A4	=2
Req1.2	Req1.2.2	C5	=4	A5	=2
Req1.2	Req1.2.3	C6	=5	A6	=2
Req1.3	Req1.3.1	C7	=6	A7	=3
Req1.3	Req1.3.2	C8	=7	A8	=3

4 carmaker requirements splitted into 8 requirements in our model

Int1= Luminosity_Level

Figure 5.15 – Feature 1 modeled using a Decision Table element

Feature 2 - *DT 2*

I3= Car_Locked

I4= Ignition

Carmaker req ID	Our model req ID	Conditions	*DT2* INPUTS		Actions	*DT2* OUTPUTS
			I3	**I4**		**Int2**
Req2.2	Req2.2.1	C1	=0	=0	A1	=0
Req2.2	Req2.2.2	C2	=0	=1	A1	=0
Req2.1	Req2.1	C3	=1	=0	A3	=1
Req2.2	Req2.2.3	C4	=1	=1	A4	=0

2 carmaker requirements splitted into 4 requirements in our model

Int2= Follow_Me_Home_Activate

Figure 5.16 – Feature 2 modeled using a Decision Table element

A graphical illustration of the *Finite State Machine element* of the *Feature 3* is developed in *Figure 5.17*. This *FSM* is not exhaustive. *Figure 5.18* details the *states*, *transitions* and *conditions* of this *FSM*. In fact, the *Feature 3* has a sequential behavior since the behavior of the signals *O1* and *O2* depends not only on the signals *Config1*, *Config2*, *Config3*, *I1*, *I5*, *Int1*

and *Int2* but also on the current *state* of the *feature*. We also define a *FSM* timing signal *(FSM1Tempo)* and one *FSM* internal signal (*FSM1Int1*, Domain = {0, 1}) in order to model the requirements *Req3.5* and *Req3.6*.

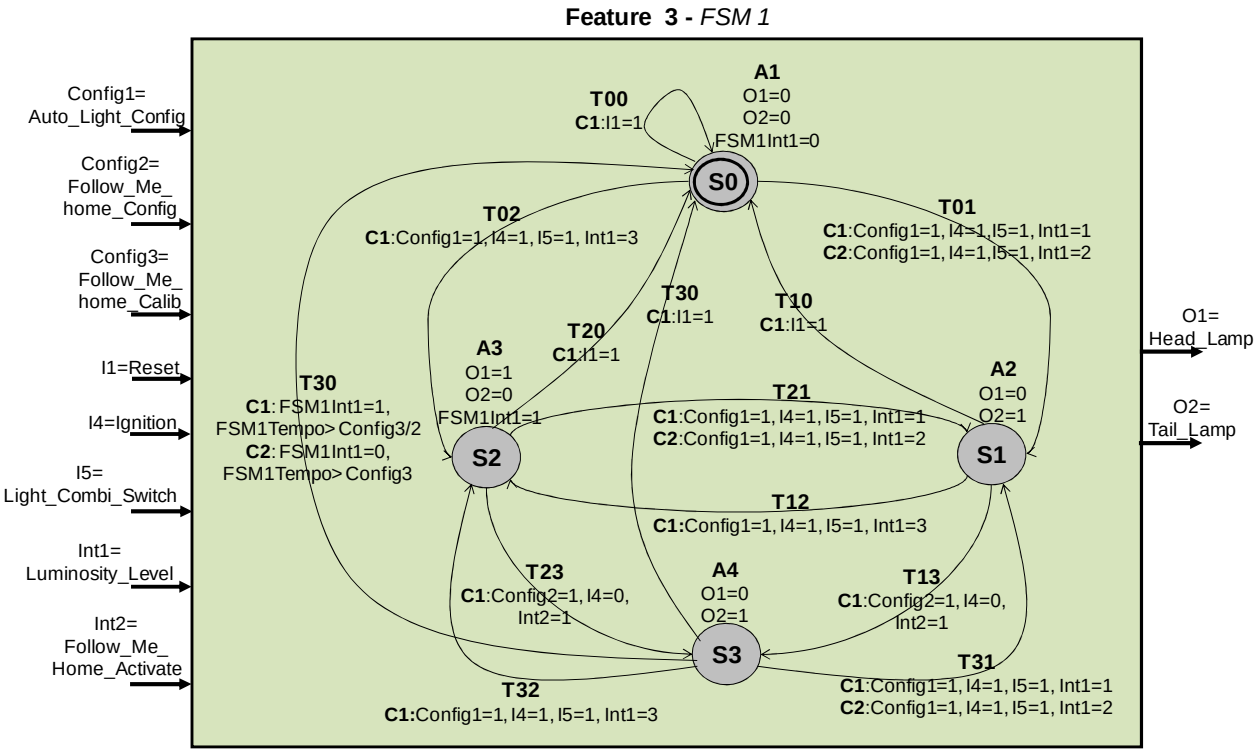
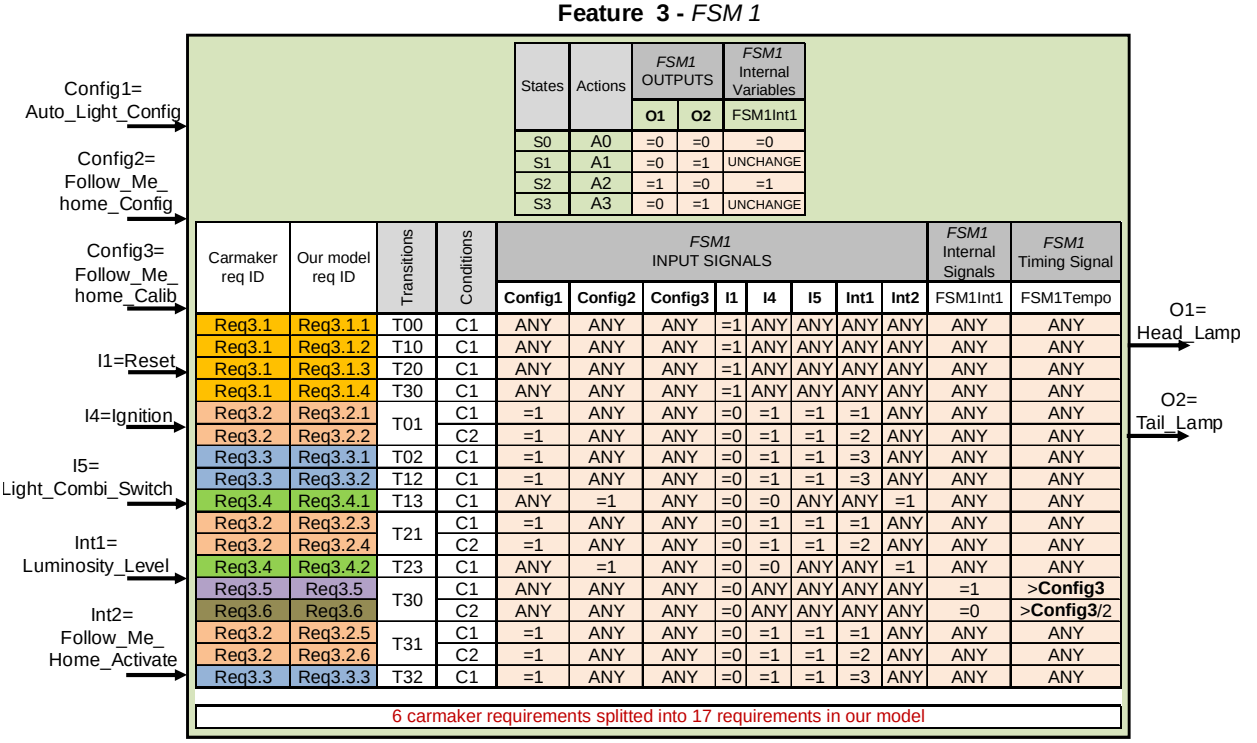


Figure 5.17 – Feature 3 modeled using a Finite State Machine element – Graphical illustration



States	Actions	*FSM1* OUTPUTS		*FSM1* Internal Variables
		O1	O2	FSM1Int1
S0	A0	=0	=0	=0
S1	A1	=0	=1	UNCHANGE
S2	A2	=1	=0	=1
S3	A3	=0	=1	UNCHANGE

Carmaker req ID	Our model req ID	Transitions	Conditions	*FSM1* INPUT SIGNALS								*FSM1* Internal Signals	*FSM1* Timing Signal
				Config1	Config2	Config3	I1	I4	I5	Int1	Int2	FSM1Int1	FSM1Tempo
Req3.1	Req3.1.1	T00	C1	ANY	ANY	ANY	=1	ANY	ANY	ANY	ANY	ANY	ANY
Req3.1	Req3.1.2	T10	C1	ANY	ANY	ANY	=1	ANY	ANY	ANY	ANY	ANY	ANY
Req3.1	Req3.1.3	T20	C1	ANY	ANY	ANY	=1	ANY	ANY	ANY	ANY	ANY	ANY
Req3.1	Req3.1.4	T30	C1	ANY	ANY	ANY	=1	ANY	ANY	ANY	ANY	ANY	ANY
Req3.2	Req3.2.1	T01	C1	=1	ANY	ANY	=0	=1	=1	=1	ANY	ANY	ANY
Req3.2	Req3.2.2		C2	=1	ANY	ANY	=0	=1	=1	=2	ANY	ANY	ANY
Req3.3	Req3.3.1	T02	C1	=1	ANY	ANY	=0	=1	=1	=3	ANY	ANY	ANY
Req3.3	Req3.3.2	T12	C1	=1	ANY	ANY	=0	=1	=1	=3	ANY	ANY	ANY
Req3.4	Req3.4.1	T13	C1	ANY	=1	ANY	=0	=0	ANY	ANY	=1	ANY	ANY
Req3.2	Req3.2.3	T21	C1	=1	ANY	ANY	=0	=1	=1	=1	ANY	ANY	ANY
Req3.2	Req3.2.4		C2	=1	ANY	ANY	=0	=1	=1	=2	ANY	ANY	ANY
Req3.4	Req3.4.2	T23	C1	ANY	=1	ANY	=0	=0	ANY	ANY	=1	ANY	ANY
Req3.5	Req3.5	T30	C1	ANY	ANY	ANY	=0	ANY	ANY	ANY	ANY	=1	>**Config3**
Req3.6	Req3.6		C2	ANY	ANY	ANY	=0	ANY	ANY	ANY	ANY	=0	>**Config3**/2
Req3.2	Req3.2.5	T31	C1	=1	ANY	ANY	=0	=1	=1	=1	ANY	ANY	ANY
Req3.2	Req3.2.6		C2	=1	ANY	ANY	=0	=1	=1	=2	ANY	ANY	ANY
Req3.3	Req3.3.3	T32	C1	=1	ANY	ANY	=0	=1	=1	=3	ANY	ANY	ANY

6 carmaker requirements splitted into 17 requirements in our model

Figure 5.18 – Feature 3 modeled using a Finite State Machine element – States, Transitions and Conditions

Even with instructions and guidelines, we are conscious that two different *modelers* can design two different models for the same software functional requirements. To overcome this problem, we plan to demonstrate that for two or more different models of the same functional requirements, the generated test cases allow to detect the same bugs (Cf. *Chapter 10 – Section 8.B*).

VI. Conclusion

Managing the software functional requirements is considered as one of the key issues in the software development process. In fact, these requirements are the main input for the design and implementation processes of the software product but also for the *verification and validation* processes. Ten years ago, *formal* methods were rarely used in automotive industry, contrary to medical, avionics and railways industries. Now, in automotive industry, *semi-formal* and *formal* methods are more and more used to specify software functional requirements (Cf. *Diagnosis 3*). However, there is a lack of a standard *formal*ism shared between carmakers and suppliers. In fact, for each project, the supplier has to adapt its processes to the *formal*ism used by the carmaker.

In this chapter, we developed our new *formal* and *simulation* language to model software functional requirements (Cf. *Diagnosis 6*). A *simulation model* of these requirements could help to avoid ambiguity, incompleteness and inconsistency in customers' requirements (Cf. *Diagnosis 5*). Development and validation teams could communicate more easily with the customer and fix specification's problems (Cf. *Diagnosis 2*). Moreover, through a *simulation model*, one could automate the assessment process of all the expected outputs values of a software product (Cf. *Diagnosis 10, 12 and 15*). In fact, when designing test cases, test engineers could perform the selected *operation* on the *requirements model* and assess the expected output values automatically by simulating the model. Finally, one could *formally* measure the *coverage* of the requirements model (Cf. *Diagnosis 11*).

In the following chapter, we develop how a *modeler* can verify and validate the *completeness*, the *consistency*, the *accuracy* and the *compliance* of a requirements model with the carmaker's requirements.

CHAPTER 6. VERIFICATION AND VALIDATION OF A SOFTWARE FUNCTIONAL REQUIREMENTS MODEL

I. Introduction

Simulation models are increasingly being used in problem solving and in decision making. The developers and users of these models, the decision makers using information derived from the results of the models, and people affected by decisions based on such models are all rightly concerned with whether a model and its results are "correct". This concern is addressed through *Model Verification and Validation* (*Model V&V*). In this dissertation, we adopt the definition of *Model V&V* proposed by Balci in (Balci 1997).

Definition 6.1: Model Verification (Balci 1997)

Model Verification is substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy. Model verification deals with building the model right. The accuracy of transforming a problem formulation into a model specification or the accuracy of converting a model representation from a micro flowchart form into an executable computer program is evaluated in model verification.

Definition 6.2: Model Validation (Balci 1997)

Model Validation is substantiating that the model, within its domain of applicability, behaves with satisfactory accuracy consistent with the M&S (Modeling and Simulation) objectives. Model validation deals with building the right model.

Model V&V are essential parts of the model development process if the model has to be used by organizations. *Model V&V* is not a phase or step in the *M&S* life cycle, but a continuous activity throughout the entire *M&S* life cycle. The *M&S* life cycle should not be interpreted as strictly sequential. The *M&S* life cycle is iterative in nature and reverse transitions are expected. Deficiencies identified by *Model V&V* activity may necessitate returning to an earlier process and starting all over again. The *Model V&V* activity throughout the entire *M&S* life cycle is intended to reveal any quality deficiencies that might be present as the *M&S* progresses from the problem definition to the completion of the *M&S* application. Errors should be detected as early as possible in the *M&S* life cycle.

In this chapter, we develop scenarios in order to verify and validate a software functional requirements model developed using our *formal* simulation language. A survey on verifying and validating simulation model is performed in *Section 2*. We consider a simplified version of the modeling process. We discuss the basic approaches used in deciding model validity. We also describe various *Model V&V* techniques. Based on the literature review, techniques and rules to help modelers in validating the *Conceptual Model*, verifying the *Computerized Model* and finally checking the *Operational Validity* of a requirements model are respectively proposed in *Section 3*, *4* and *5*. These proposals take the industrial constraints and the automotive context into account.

II. A survey on verifying and validating a simulation model

A model should be developed for a specific purpose and its validity determined with respect to that purpose. If the purpose of a model is to answer a variety of questions, the validity of the model needs to be determined with respect to each question. Several sets of experimental conditions are usually required to define the domain of a model's intended applicability. A model may be valid for one set of experimental conditions and invalid in another. A model is considered valid for a set of experimental conditions if its accuracy is within its acceptable range, which is the amount of accuracy required for the model's intended purpose. Several

versions of a model are usually developed prior to obtaining a satisfactory valid model. The substantiation that a model is valid (e.g. *Model V&V*) is generally considered to be a process and is usually part of the model development process.

It is often too costly and time consuming to determine that a model is absolutely valid over the complete domain of its intended applicability. Tests and evaluations are conducted until sufficient confidence is obtained that a model can be considered valid for its intended application. The relationships of cost (a similar relationship holds for the amount of time) of performing model validation and the value of the model to the user as a function of *model confidence* is proposed by Sargent in (Sargent 2005) and illustrated in *Figure 6.1*. The cost of model validation is usually quite significant, particularly when extremely high *model confidence* is required.

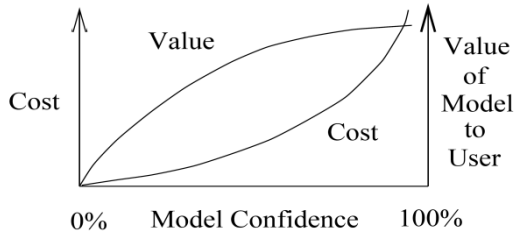


Figure 6.1 – Model confidence (Sargent 2005)

A. A simplified version of the modeling process

Sargent (Sargent 2005) has proposed a simplified version of the modeling process in *Figure 6.2*. 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 experimentation phase.

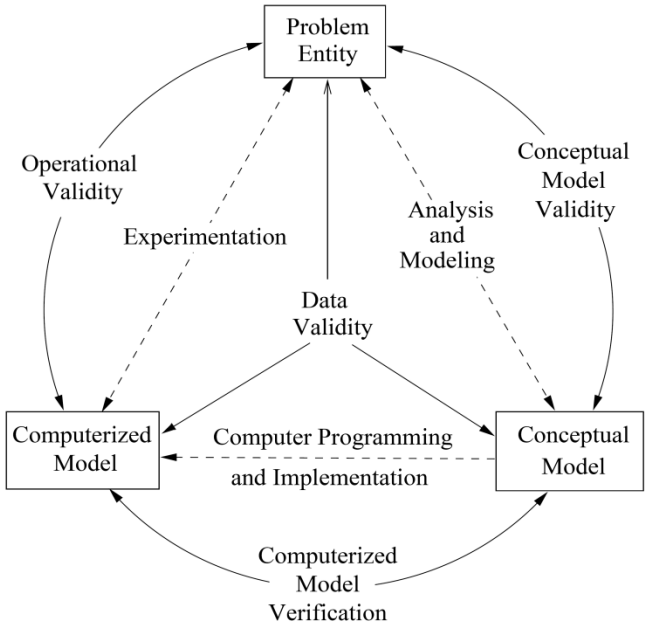


Figure 6.2 – A simplified version of the modeling process (Sargent 2005)

In the next three sections, we develop the three *Model V&V* activities (*Conceptual Model Validity*, *Computerized Model Verification*, *Operational Validity*) related to the simplified version of the modeling process presented in *Figure 6.2*.

1. Conceptual Model Validity

Conceptual Model Validity is determining that 1) the theories and assumptions underlying 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. Next, each sub-model and the overall model must be evaluated to determine if they are reasonable and correct for the intended purpose of the model. This should include determining if the appropriate detail and aggregate relationships have been used for the model's intended purpose, and if the appropriate structure, logic, and mathematical and causal relationships have been used. The primary validation techniques used for these evaluations are *Face validity* and *Traces* (Cf. *Section 2.C* for more details on these techniques). *Face validity* is based on experts' evaluation of the *Conceptual Model* in order to determine if it is correct and reasonable for its purpose (*Problem Entity*). This usually requires examining the flowchart or graphical model, or the set of model equations. The use of *Traces* is the tracking of entities through each sub-model and the overall model to determine if the logic is correct and if the necessary accuracy is maintained. If errors are found in the *Conceptual Model*, it must be revised and *Conceptual Model Validity* performed again.

2. Computerized Model Verification

Computerized Model Verification ensures that the computer programming and implementation of the *Conceptual Model* are correct. To help ensure that a correct computer program is obtained, program design and development procedures found in the field of *software engineering* should be used in developing and implementing the computer program. One should be aware that the type of computer language used affects the probability of having a correct program. The use of a special-purpose simulation language generally results in having fewer errors than if a general-purpose simulation language is used, and using a general purpose simulation language generally results in having fewer errors than if a general purpose higher order language is used. Not only does the use of simulation languages increase the probability of having a correct program, programming time is usually reduced significantly. After the computer program has been developed and implemented, the program must be tested for correctness. Main functions but also sub-functions must be tested to see if they are correct. It is necessary to be aware while checking the correctness of the computer model that errors may be caused by the *Conceptual Model* or the computer implementation.

3. Operational Validity

Operational Validity is concerned with determining that the model's output behavior has the accuracy required for the model's intended purpose over the domain of its intended applicability. This is where most of the validation and evaluation techniques take place. The *Computerized Model* is used in *Operational Validity*, and thus any deficiencies found may be due to an inadequate *Conceptual Model*, an improperly programmed or implemented *Conceptual Model* (e.g. due to programming errors or insufficient numerical accuracy), or due to invalid data. All of the *Model V&V* techniques discussed in *Section 2.C* are applicable to *Operational Validity*. Which techniques to use must be decided by the model development team and other interested parties. The major attribute affecting *Operational Validity* is

whether the *Problem Entity* (or system) is observable, where observable means it is possible to collect data on the operational behavior of the program entity.

Finally, *Data Validity* is defined as ensuring that the data necessary for model building, model evaluation and testing, and conducting the model experiments to solve the problem are adequate and correct.

B. How to decide whether a simulation model is valid or not?

According to Sargent (Sargent 2005), three basic approaches are used in deciding whether a simulation model is valid or invalid. Each of the approaches requires the model development team to conduct the *Model V&V* as part of the model development process:

1. The most common approach is based on the model development team who has to make the decision whether the model is valid or not. This is a subjective decision based on the results of the various tests and evaluations conducted as part of the model development process.
2. Another approach, often called "*Independent Verification and Validation*" (*IV&V*), uses a third (independent) party to decide whether the model is valid. The third party is independent of both the model development team and the model user(s). After the model is developed, the third party conducts an evaluation to determine its validity. Based upon this validation, the third party makes a subjective decision on the validity of the model. The evaluation performed in the *IV&V* approach ranges from simply reviewing the *Model V&V* conducted by the model development team to a complete *verification and validation* effort. According to Wood (Wood 1986), a complete *IV&V* evaluation is extremely costly and time consuming for what is obtained.
3. Balci (Balci 1989) proposes an approach based on a *scoring model* for determining whether a model is valid or not. Scores (or weights) are determined subjectively when conducting various aspects of the validation process and then combined to determine category scores and an overall score for the simulation model. A simulation model is considered valid if its overall and category scores are greater than some passing score(s). This approach is infrequently used in practice. Sargent (Sargent 2005) does not believe in the use of a scoring model for determining validity, because 1) the subjectiveness of this approach tends to be hidden and thus appears to be objective, 2) the passing scores must be decided in some (usually subjective) way, 3) a model may receive a passing score and yet have a defect that needs correction, and finally 4) the score(s) may cause overconfidence in a model or be used to argue that one model is better than another.

Several versions of a model are usually developed in the modeling process prior to obtaining a satisfactory valid model. During each model iteration, *Model V&V* are performed. A variety of techniques could be used. In the next section, we develop these techniques.

C. Model Verification and Validation techniques

Taxonomy of more than 77 *Model V&V* techniques for simulation models is identified in (Balci 1997). Most of these techniques come from the *software engineering* discipline and the remaining are specific to the modeling and simulation field. Details on these techniques are proposed in (DoD 1996, Balci 1997). The taxonomy used by the authors classifies the *Model V&V* techniques into four primary categories: *informal*, *static*, *dynamic*, and *formal*. The use of mathematical and logic *formal*ism by the techniques in each primary category increases from *informal* to *formal*. Likewise, the complexity also increases as the primary category

becomes more *formal*. In the following, we describe various techniques used in *Model V&V*. Most of the techniques described here are found in the literature (Balci 1984, Sargent 2005), although some may be described slightly differently (adapted to our context). They can be used either subjectively or objectively. By "objectively", we mean using some type of statistical test or mathematical procedure (e.g. confidence intervals). A combination of techniques is often used for validating and verifying the sub-models and the overall model.

Animation: The model's operational behavior is displayed graphically as the model moves through time.

Comparison to Other Models: Various results (e.g. outputs) of the simulation model being validated are compared to results of other (valid) models.

Degenerate Tests: The degeneracy of the model's behavior is tested by appropriate selection of values of the input and internal parameters.

Event Validity: The *events* of occurrences of the simulation model are compared to those of the real system to determine if they are similar.

Extreme Condition Tests: The model structure and output should be plausible for any extreme and unlikely combination of levels of factors in the system.

Face Validity: Face validity is asking people knowledgeable about the system whether the model and/or its behavior are reasonable. This technique can be used in determining if the logic in the *Conceptual Model* is correct and if a model's input-output relationships are reasonable.

Fixed Values: Fixed values (e.g., constants) are used for various model input and internal variables and parameters. This should allow the checking of model results against easily calculated values.

Historical Data Validation: If historical data exist (or if data are collected on a system for building or testing the model), part of the data is used to build the model and the remaining data are used to determine (test) whether the model behaves as the system does.

Internal Validity: Several replications (runs) of a *stochastic model* are made to determine the amount of (internal) stochastic variability in the model. A high amount of variability (lack of consistency) may cause the model's results to be questionable and may question the appropriateness of the system being investigated.

Parameter Variability - Sensitivity Analysis: This technique consists of changing the values of the input and internal parameters of a model to determine the effect upon the model's behavior and its output. The same relationships should occur in the model as in the real system.

Predictive Validation: The model is used to predict (forecast) the system behavior, and then comparisons are made between the system's behavior and the model's forecast to determine if they are the same. The system data may come from an operational system or from experiments performed on the system.

Traces: The behavior of different types of specific entities in the model is traced (followed) through the model to determine if the model's logic is correct and if the necessary accuracy is obtained.

Turing Tests: People who are knowledgeable about the operations of a system are asked if they can discriminate between system and model outputs.

Unfortunately, no algorithms or procedures exist to select which techniques to use. However, some attributes that affect which techniques to use are discussed by Sargent in (Sargent 1984). In the next three sections (*Section 3*, *4* and *5*), we specify techniques, rules and scenarios to help modelers in validating the *Conceptual Model*, verifying the *Computerized Model* and finally checking the *Operational Validity* of a requirements model. Our proposals take not only the Sargent's recommendations into account but also our industrial context.

III. Using the experts' knowledge to validate a Conceptual requirements Model (Conceptual validity)

In our context, the *Conceptual Model* is developed through an analysis and modeling of the software functional requirements. For each software functionality, modelers draw a sketch of the requirements model by (Cf. *Chapter 5 – Section 3*):

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 *elements* (*DT* and *FSM*) and the *intermediate signals* and their domains and
4. finally specifying each *element*. For a *DT*, identify the *conditions* and their associate *actions*. For a *FSM*, identify the *states* and their associate *actions*, the *transitions* and their associate *conditions* and if needed the *internal* and *timing signals*.

Once the *Conceptual Model* is designed, each element and the overall model must be evaluated to determine if they are reasonable, correct and complete regarding the carmaker's requirements. We propose to use *Face validity* and *Turing tests* in order to valid our *Conceptual Model*. In fact, the experts' knowledge is the main source of validating our *Conceptual Model*. People knowledgeable about the system under test are asked to discriminate between the model and the carmaker's requirements and to give their confidence in the model and/or its behavior.

IV. A set of integrity rules to verify a Computerized requirements Model

The *Computerized Model* is developed through a computer programming and implementation of the *Conceptual Model*. We provide modelers a high level graphical language to help them computerizing their *Conceptual requirements Model*s. The main items of this language are illustrated in *Figure 6.3*.

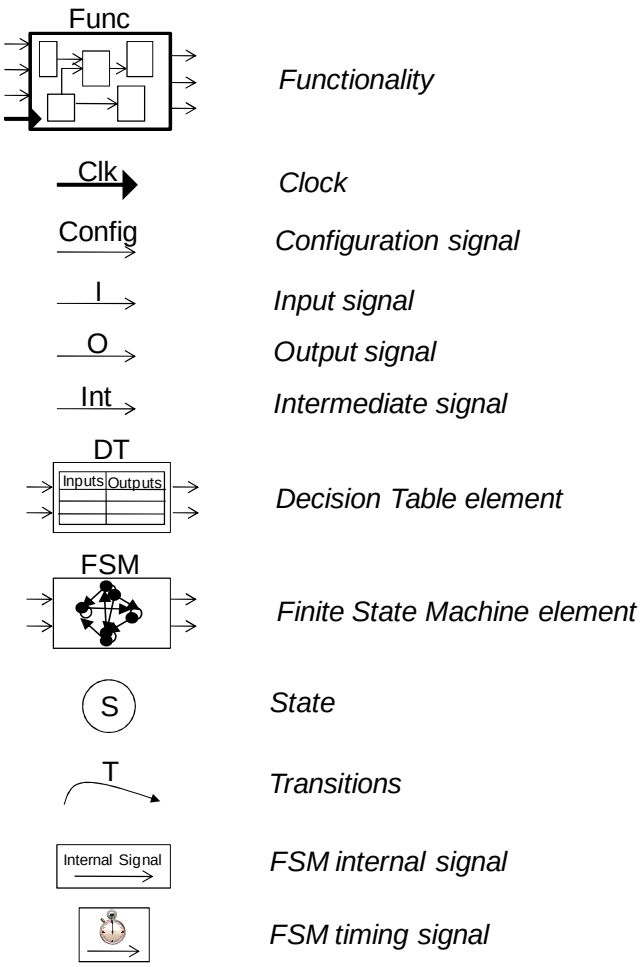


Figure 6.3 – A high level graphical language to computerize a Conceptual requirements Model

Computerized Model Verification ensures that the computer programming and implementation of the *Conceptual Model* are correct. The use of a graphical simulation language generally results in having fewer errors and programming time is usually reduced significantly. To help ensure that a correct computer model is obtained, we develop a set of integrity rules to be checked automatically on the computer model. These rules are developed in *Table 6.1*.

Rule #	Rule description
Rule 1	Each functionality must have one clock
Rule 2	Each functionality must have at least 1 input and 1 output signals
Rule 3	Each functionality must have at least one element (a Decision Table or a Finite State Machine)
Rule 4	All the input signals of the functionality must be inputs of elements
Rule 5	All the output signals of the functionality must be outputs of elements
Rule 6	All the intermediate signals of the functionality must be inputs or outputs of elements
Rule 7	All the inputs and outputs of elements must be input, output or intermediate signals of the functionality
Rule 8	Each value of an input, output and intermediate signal of the functionality must be considered in at least one condition or action of an element
Rule 9	Each DT must have at least one condition
Rule 10	Each condition of a DT must have one associated action
Rule 11	Each condition of a DT must have at least one input or intermediate signal of the functionality
Rule 12	Each action of a DT must have at least one output or intermediate signal of the functionality
Rule 13	Each FSM must have at least two states and two transitions
Rule 14	Each transition of a FSM must have at least one condition
Rule 15	Each state of a FSM must have one associated action
Rule 16	Each condition of a FSM must have at least one input or intermediate signal of the functionality
Rule 17	Each action of a FSM must have at least one output or intermediate signal of the functionality
Rule 18	Each state of a FSM must have at least one transition that gets in the state and one transition that gets out of the state
Rule 19	Each transition of a FSM must have an origin and a destination state

Table 6.1 – Integrity rules for verifying a Computerized requirements Model

V. Three possible scenarios to validate a Computerized requirements Model (Operational Validity)

Computerized Model Verification ensures that mistakes have not been made in the computer implementation of the *Conceptual Model*. It does not ensure the compliance of the computer model with the (original) carmaker requirements. The *Operational Validity* aims to verify that the computer model behavior has the accuracy required by the carmaker. To do this, computer experiments are conducted on the *Computerized Model* in the experimentation phase. This is where most of the model deficiencies are detected. Errors may be due to an erroneous *Conceptual Model* or to programming errors in computerizing the *Conceptual Model*. In our context, all of the *Model V&V* techniques discussed in *Section 2.C* are applicable. Which techniques to use must be decided by the constraints on the system under test but also by the model development team. In fact, we identify three possible scenarios to help modelers validating a *Computerized requirements Model* (*Operational Validity*). These scenarios can be used concurrently or separately.

A. First scenario: Animate our requirements model

The most used technique to validate a simulation model is obviously to animate and trace it (Cf. *Figure 6.4*). The behavior of *intermediate* and *output signals* is provided graphically through time. However, two questions can be raised when animating a model 1) How do I choose the *input data* of the model? and 2) How can I be sure that the model behaves well (expected *output data*)? In order to answer the first question, we can refer to some of the *Model V&V* techniques provided in *Section 2.C* (for instance, *degenerate tests, extreme condition tests, fixed values, and parameter variability - sensitivity analysis*). The second question is more related to the *formal*ism of the (original) carmaker requirements. In case of *informal* or *semi-formal* requirements, modelers have to predict the system behavior by analyzing the requirements. In case of *formal* and *simulated* requirements, expected *output data* can be assessed automatically by simulating the requirements.

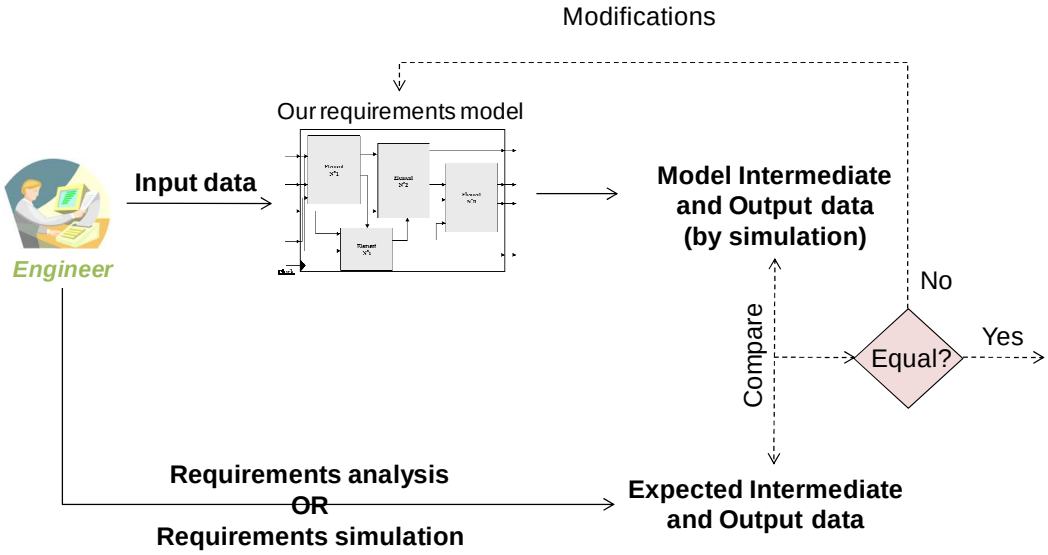


Figure 6.4 – Animate the requirements model

B. Second scenario: Simulate the test cases delivered by the carmaker on our requirements model

Sometimes, carmakers deliver a set of *test cases* for the software product under test. These *test cases* can be used to valid our requirements model (Cf. *Figure 6.5*).

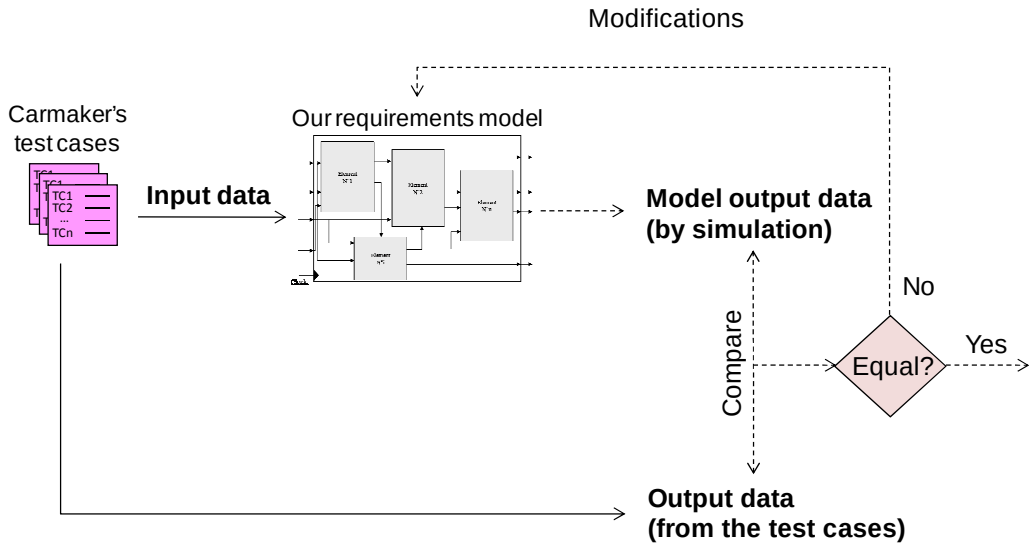


Figure 6.5 – Simulate the test cases delivered by the carmaker on our requirements model

C. Third scenario: Compare our requirements model to another valid model of the requirements

Ten years ago, *simulation* methods were rarely used in automotive industry. Now, automotive electronics architecture becomes more and more complex and carmakers outsource the design of electronic products. Therefore, it becomes crucial for carmakers to simulate their global electronics architecture in order to better integrate and validate different electronic parts from different suppliers. Presently, in automotive industry, *formal* (simulation) methods are more and more used to specify software functional requirements (Cf. *Chapter 2 – Section 4.A*). In fact, simulation helps engineers to make better decisions all along the product life cycle and detect problems early in the development process. Unfortunately, there is a lack of a standard *formal*ism shared between carmakers and suppliers (Cf. *Diagnosis 3*). However, some existing simulation tools (StateMate[32], Matlab/Simulink[33]) are currently used by carmakers and suppliers to simulate software specifications. A graphical interface generated automatically from a *formal* specification (delivered by a carmaker to Johnson Controls) of the "Front Wiper" functionality is illustrated in *Figure 5.6*. An engineer can animate this model manually or simulate a set of *input data* automatically (set values on the *input signals*) and check the expected behavior of the functionality (check values on the *output signals*).

[32] http://www.telelogic.com/products/statemate/index.cfm (Consulted on November 2008)
[33] http://www.mathworks.com/products/simulink/ (Consulted on November 2008)

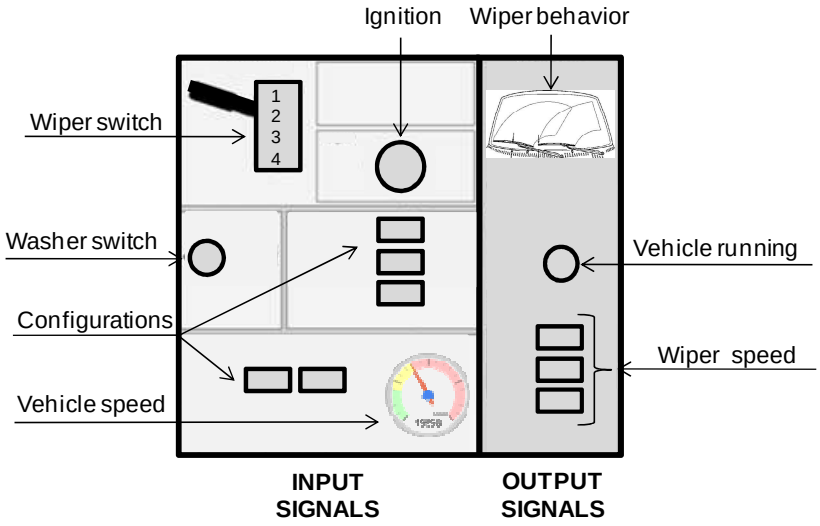


Figure 6.6 – A graphical interface generated automatically from a formal specification of the "Front Wiper" software functionality

Once a simulation model of the software functionality under test exists (Cf. *Figure 6.7*), an engineer can choose a set of *input data* (using techniques such as *degenerate tests, extreme condition tests, fixed values, parameter variability - sensitivity analysis*) and simulate these data on the "*valid*" model (model delivered by the carmaker) and on our requirements model in order to verify the validity of our model.

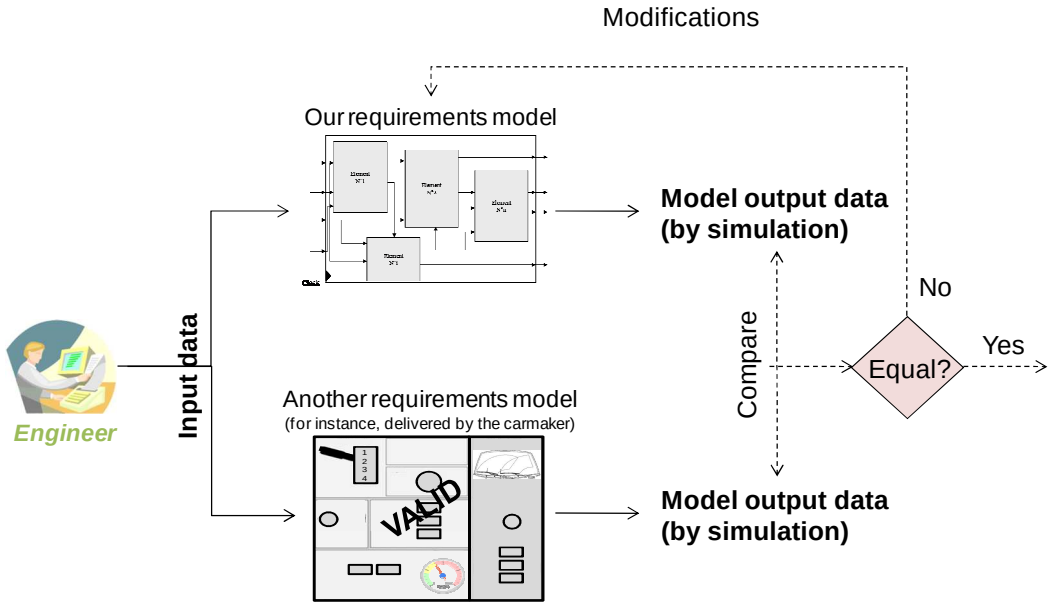


Figure 6.7 – Compare our requirements model to another valid model of the requirements

VI. Conclusion

Assessing credibility of a simulation model is an onerous task. Applying *Model V&V* techniques throughout a simulation model is time consuming and costly. However, the *Model V&V* activity is extremely important for successful completion of complex and large-scale *Modeling and Simulation* (*M&S*) efforts. Unfortunately, there is no set of specific tests that can easily be applied to determine the "correctness" of the model. Furthermore, no algorithm exists to determine what techniques or procedures to use. Every new simulation project

presents a new and unique challenge. However, there is considerable literature on *Model V&V*.

In this chapter, we developed a process to verify and validate a software functional requirements model. We use proposals developed in the literature as a starting point for defining methods and techniques more adapted to our context. We consider a simplified version of the modeling process: *Problem Entity*, *Conceptual Model* and *Computerized Model*. Firstly, we propose to validate (Conceptual Validity) a *Conceptual requirements Model* via experts' knowledge. Secondly, we define a set of integrity rules to verify a *Computerized requirements Model.* Finally, we develop three possible scenarios to validate (*Operational Validity*) a *Computerized requirements Model*.

In the following chapter, we describe how *test cases* can be generated automatically from a requirements model.

CHAPTER 7. AUTOMATIC GENERATION OF TEST CASES

I. Introduction

As the software products become more and more complex (Cf. *Chapter 1*), it is illusory to be able to check that the software product responds correctly to all possible *operations*. In *Chapter 8 – Section 2*, we further demonstrate that *software testing* is a *NP-Complete* problem and therefore it is impossible to be able to cover all the *operation space*. In Johnson Controls, the current strategy to select *operations* within the *operation space* (*operation selection strategy*) is a manual subjective one based the test engineers' experience and intuition (Cf. *Diagnosis 13*). After choosing an *operation* to be performed on the software under test, test engineers analyze the *source code* and/or the carmaker requirements of this software in order to assess the expected values to be checked on some *output signals*. In fact, this assessment is based on the engineers' understanding of the code and/or requirements and may lead to errors (*Cf. Diagnosis 12*). In automotive industry, these tasks become laborious tasks that account for more than 50% of the total *time and budget* of a software project. In fact, the stopping criteria used is based on *test coverage*. Researches in *code coverage* measurement have reached a high level of maturity and many automated tools were commercialized (Cf. *Chapter 2 – Section 6.A.1*). However, *requirement coverage* is still immature and the accuracy of a *requirement coverage* measurement depends on the degree of formalism used when specifying a set of requirements (Cf. *Diagnosis 11*). In addition, sometimes, for *time* and *budget* reasons, test engineers stop designing *test cases* even if 100% *coverage* is not reached.

In this chapter, we develop our strategy to generate *test cases* (*operations* and *expected outputs*) automatically from the requirements model. We also describe our *stopping aggregated criterion* based on *formal* measurement of *coverage*. The test case generation is based on a new concept named "*operation matrix*" presented in *Section 2*. The process of generating a *test case* is described in *Section 3*. The quality objectives and the time and cost constraints when designing *test cases* are developed in *Section 4*. A new *stopping aggregated criterion* is proposed in *Section 5*. Finally, our heuristic algorithm to optimize the generation of *test cases* is specified in *Section 6*.

II. The new concept of "operation matrix"

The generation of *operations* and *inter-operation times* for a *test case* is performed based on the concept of "*operation matrix*". In fact, for each software functionality under test, we propose to set probabilities and time intervals between all possible successive *operations*. Therefore, we build a matrix that we name "*operation matrix*" which is a square matrix with all possible *operations* in columns and in rows. Between the two *operations* of a pair we define:

- The *probability* that the two operations be in sequence. The total of the probabilities by row must be equal to 1.
- The *inter-operation time between these two operations*, modeled as an interval of possible values (a uniform probability).

Let us consider a software functionality with 3 *input signals* and two *output signals*: *I1*, Domain = {0, 1}; *I2*, Domain = {1, 2, 3}; *I3*, Domain = {0, 1}; *O1*, Domain = {0, 1}; *O2*, Domain = {0, 1}. The "operation matrix" associated to this example is illustrated in *Figure 7.1*.

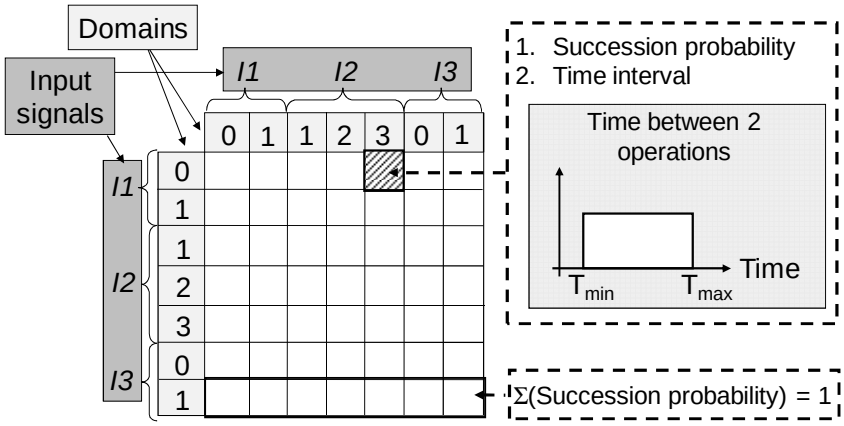


Figure 7.1 – An example to illustrate the concept of "operation matrix"

Moreover and through the "*operation matrix*", engineers can enrich the requirements model with knowledge on the *end-user* (driver) recurrent *operations* and the test engineers' experience. Indeed, high probabilities and specific *inter-operation times* can be set between recurrent and/or critical *operations*. The use of *driver behavior's profile*, *past bugs* and *existing test cases* in order to refine the *operation space* description is developed in *Chapter 8*.

One major question is: How an engineer can design an "*operation matrix*"? The basic solution is to fill in manually each case of the "*operation matrix*" by a succession probability and a time interval. However, a functionality can have more than 20 *input signals* and 100 possible values for these signals. In fact, the domain length of an analog *input signal* (for instance, vehicle speed) depends on the level of details when sampling the analog domain. In consequence, the size of an "*operation matrix*" can easily reach the 10000 cases which become inconceivable to be filled manually. One solution is to generate the "*operation matrix*" automatically. For each functionality under test, we propose to generate two "*operation matrices*" automatically. For the two matrices, the time interval can be set automatically to a "generic" interval defined by experts. Let us consider the same example of the *Figure 7.1*. The first matrix called "*Nominal 1*" (Cf. *Figure 7.2*) considers that all the *operations* have the same succession probability.

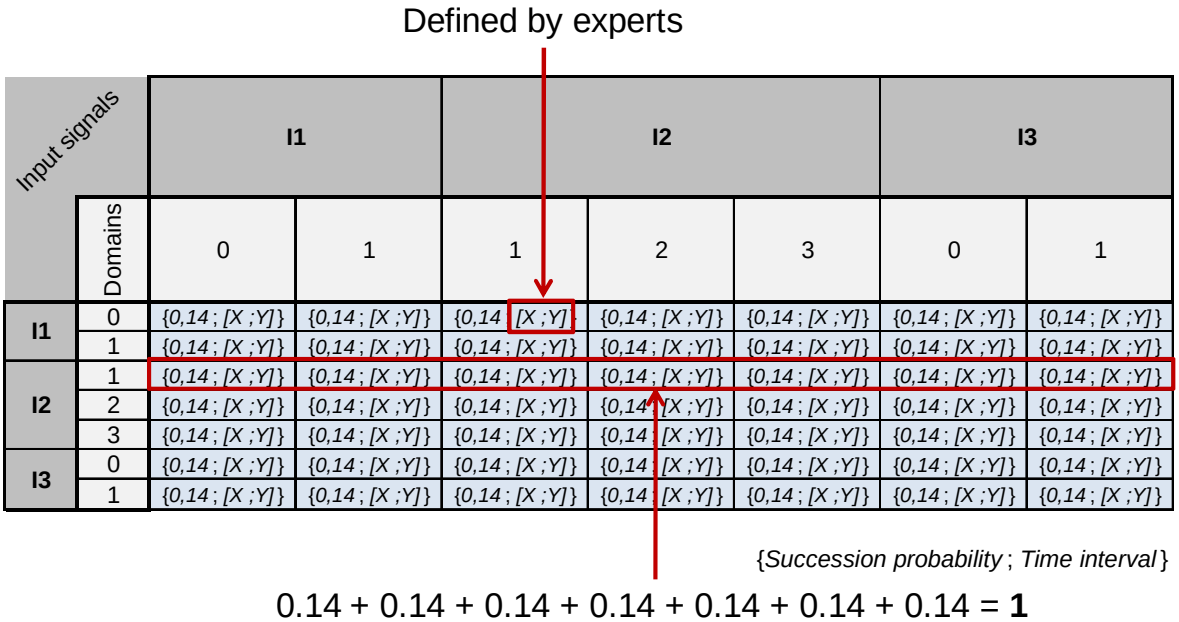


Figure 7.2 – An example of a Nominal 1 "operation matrix"

The second matrix called "*Nominal 2*" (Cf. *Figure 7.3*) considers that the probability to choose an *operation* on the *input signal* I_i is the same than the one on the *input signal* I_j.

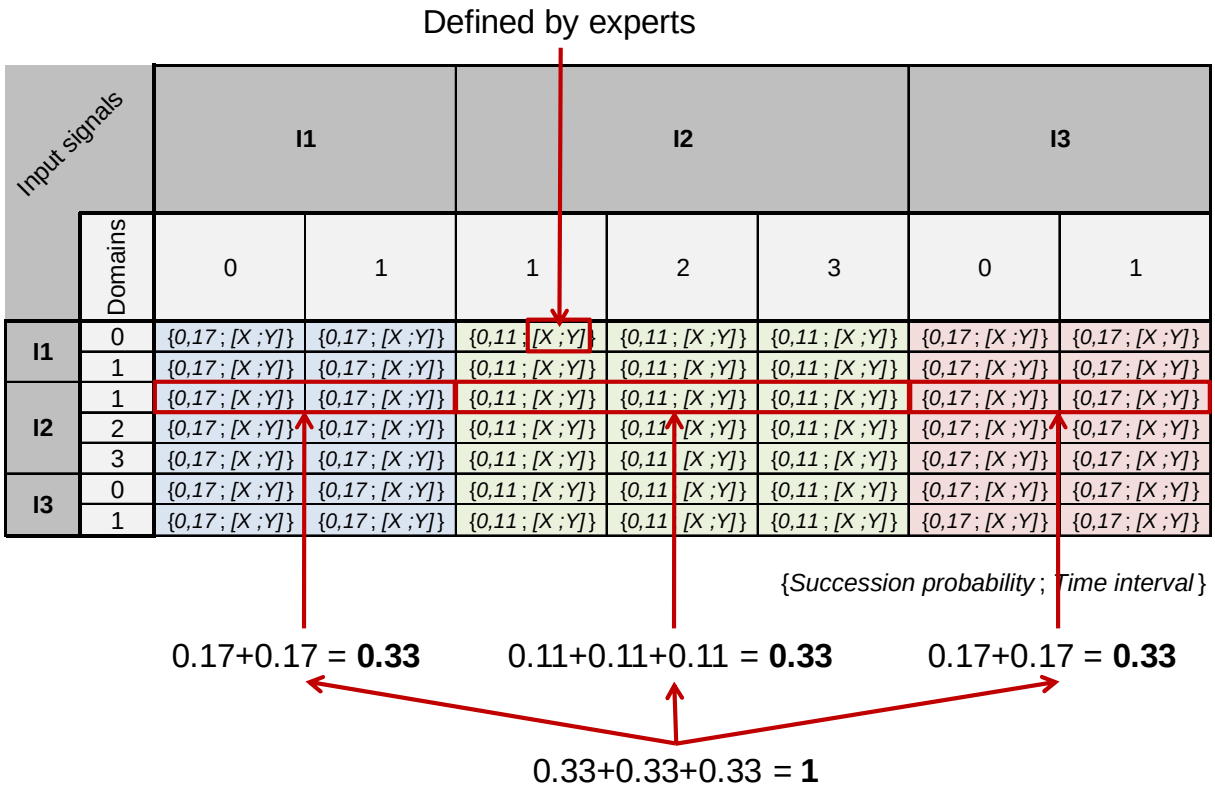
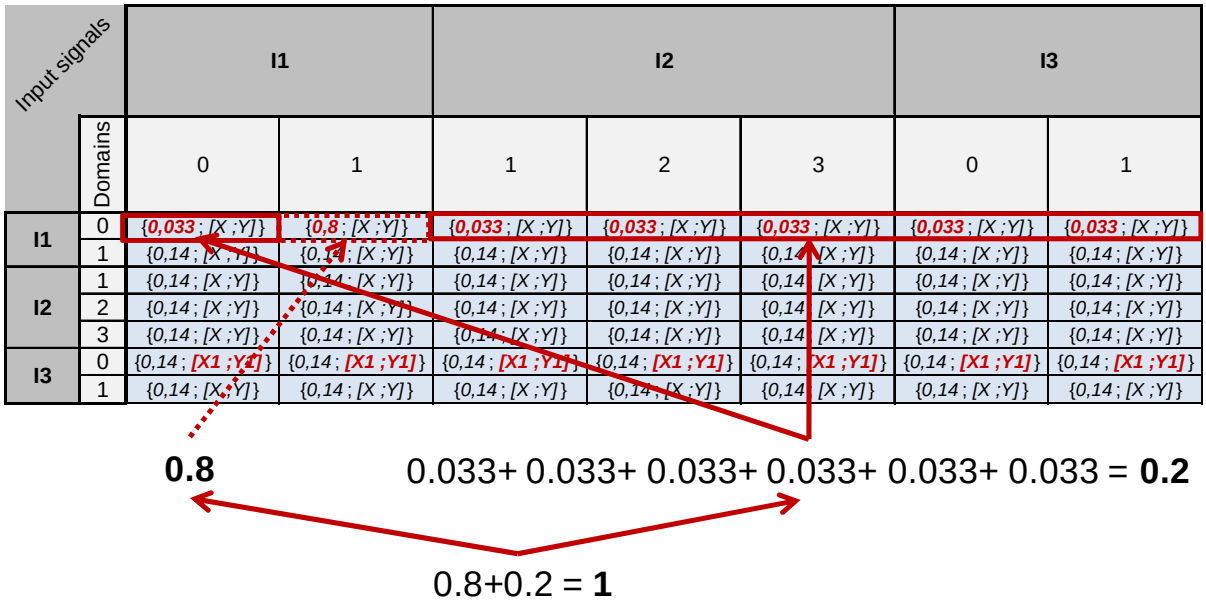


Figure 7.3 – An example of a Nominal 2 "operation matrix"

Moreover, once these matrices are generated automatically, engineers have the possibility to adjust manually the succession probability and the time interval between some specific *operations*. Following the engineers' modifications, the probability distribution by row is updated in order to take into account the matrix constraints. For instance, let us consider the *Nominal 1 "operation matrix"* of the *Figure 7.2*. One engineer can decide to:

- set the succession probability between the *operation "I1=0"* and the *operation "I1=1"* to *0.8*
- and set all the time intervals after the *operation "I3=0"* to *[X1, Y1]*

The modified *Nominal 1 "operation matrix"* is presented in *Figure 7.4*.



Input signals	Domains	I1		I2			I3	
		0	1	1	2	3	0	1
I1	0	{0,033 ; [X ;Y]}	{0,8 ; [X ;Y]}	{0,033 ; [X ;Y]}	{0,033 ; [X ;Y]}	{0,033 ; [X ;Y]}	{0,033 ; [X ;Y]}	{0,033 ; [X ;Y]}
	1	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}
I2	1	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}
	2	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}
	3	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}
I3	0	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}	{0,14 ; [X1 ;Y1]}
	1	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}	{0,14 ; [X ;Y]}

Figure 7.4 – An example of a Nominal 1 "operation matrix" after engineers' modifications

III. How to generate a "Test Case"?

Generating a *test case* automatically requires generating a set of *test steps* until *stopping criterion* is validated. The process of generating a *test case* is illustrated in *Figure 7.5*.

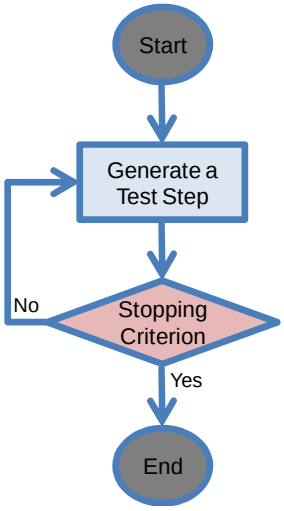


Figure 7.5 – The process of generating a test case

The definition of our *stopping aggregated criterion* is developed in *Section 5*. In the following, we focus on the generation of a *test step*. Based on the definition of a *test step* (Cf. *Definition 2.11*), designing a *test step* requires to choose an *operation*, an *inter-operation time* and assess the *expected results* to be checked on the *output signals* of the software under test. Through our approach, two automated activities are necessary to generate a *test step*:

A. Activity 1: A Monte Carlo simulation on the "operation matrix"

In order to generate an *operation* and an *inter-operation time*, we propose to perform a *Monte Carlo simulation* on the "*operation matrix*". Two steps are required:

- *Step 1:* an *operation* is chosen according to the probabilities on successive operations. In *software testing* (Marre 1992), this technique is known under the *statistical testing* technique. Before start generating a *test case*, the *input signals* of the requirements model are set to specific values. Therefore, the starting *operation* of the *test case* is randomly chosen among the initial *operations* (initial values of the *input signals*). Sometimes, the starting *operation* is chosen in order to favor a specific succession of *operations* at the beginning of the *test case*.
- *Step 2:* the *inter-operation time* is randomly chosen within the time interval of the chosen *operation*.

B. Activity 2: A simulation of the software requirements model

The chosen *operation* is performed on the requirements model and a simulation of the model (synchronized by the *cycle time* of the *Clock signal*) starts until the *inter-operation time* ran out. The values on the *output signals* of the model are the *expected results* of the *test step*. Let us consider the example of *Figure 7.1* with the *"operation matrix"* of *Figure 7.4*. The process of generating a *test step* is illustrated using this example in *Figure 7.6*.

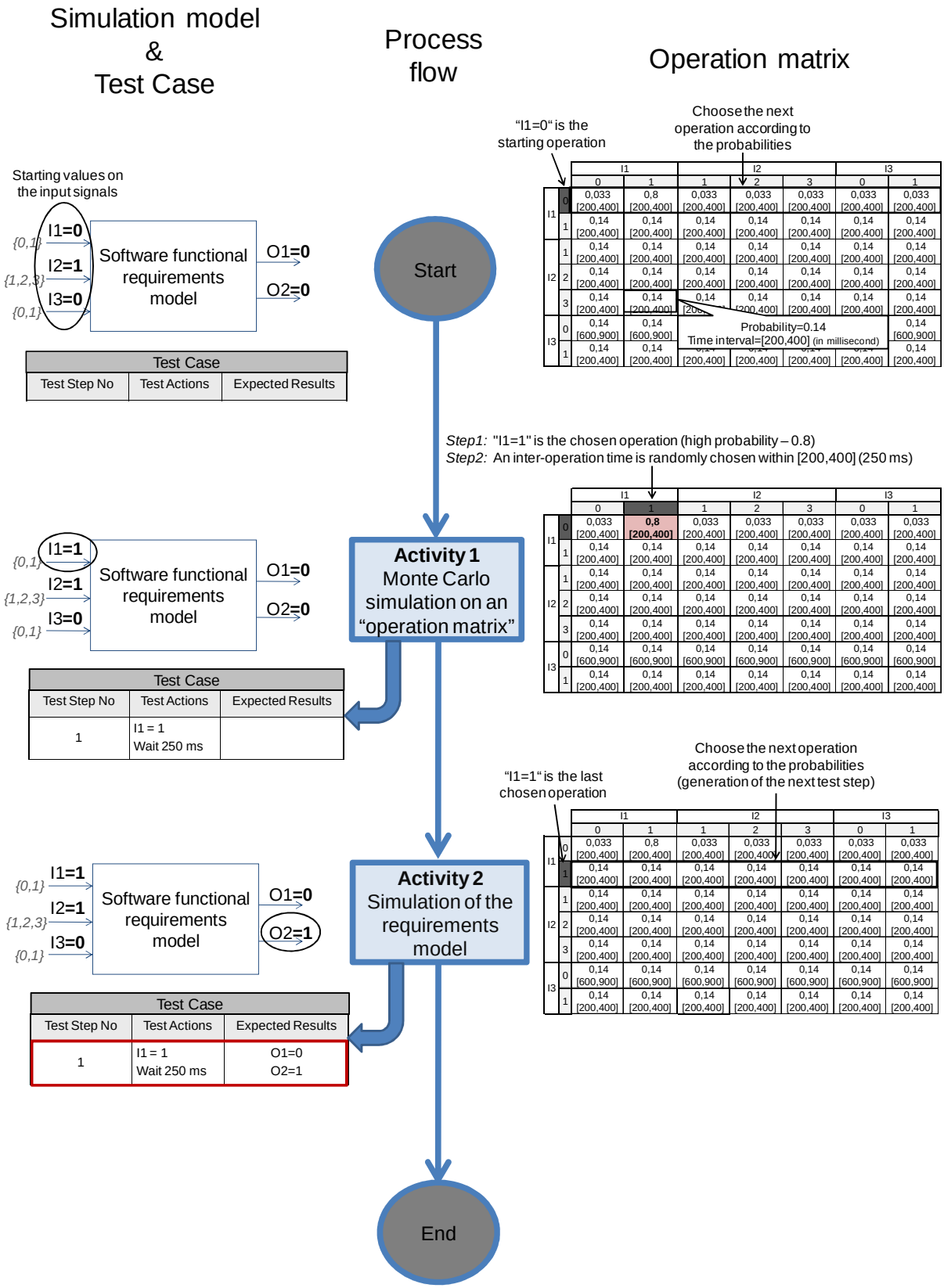


Figure 7.6 – The process of generating a test step

IV. Test generation objectives and constraints

Testing software exhaustively remains a major problem from the computing point of view. Therefore, *software testing* must often be based on specific assumptions and objectives which help test engineers and managers to decide when to stop the testing protocol. In order to monitor our automatic generation of *test cases*, we develop an *objective function* based on a *formal structural* (code) and *functional* (requirement specification) *coverage* and the execution time and cost of generated tests. In our approach, test engineers can generate *test cases* according to their quality objectives but also time and cost constraints.

A. Structural (code) coverage objectives

While generating a *test case*, and for each generation of a *test step*, we execute the *test step* on the software product under test and we evaluate the *code coverage* in terms of *statements*, *procedures*, *conditions* and *decisions coverage*. To do so, we use *C-Cover* from *Bullseye* as a *code coverage* measurement tool. A detailed description of the *code coverage* is given in *Chapter 2 – Section 6.A.1*. Since the *code coverage* measurement is already formalized using commercial tools, we focus our efforts on formalizing the *requirement coverage* measurement.

B. Functional (requirement specification) coverage objectives

Once we define a model to formally represent and simulate the software functional requirements, we consider a *formal coverage rate* of the requirements model. In *Figure 7.7*, we illustrate the functional coverage indicators through the example of *Figure 5.3*.

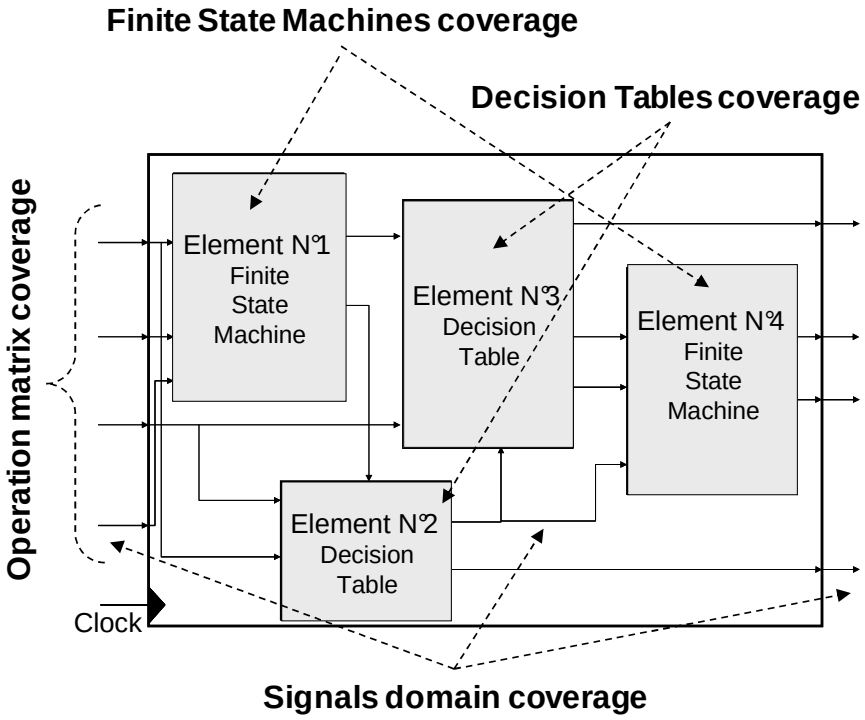


Figure 7.7 – Functional (requirement specification) coverage indicators

While generating a *test case*, test engineers can visualize in real time the covered zones of the requirements model (Cf. *Figure 7.8*, *7.9*, *7.10* and *7.11*).

1. The coverage rate of a signal domain

Each *input*, *output* or *intermediate signal* has a discrete domain. The *signal domain coverage* of a requirements model consists of the *coverage rate* of the domains of these signals. In addition, since testing the boundary values of a signal often reveals many problems, we also assess the *coverage rate* of the minimum and maximum values of each signal. In *Figure 7.8*, we illustrate the *coverage* of a signal by a practical example. After generating a *test case*, some values of the signals domains have been highlighted. In fact, the signal *"Signal_3"* is covered at 100% while the two values of this signal were visited at least once by the generated *test case*. The signal *"Signal_1"* has a *coverage rate* of 33,33% (1 value visited over 3 values in total).

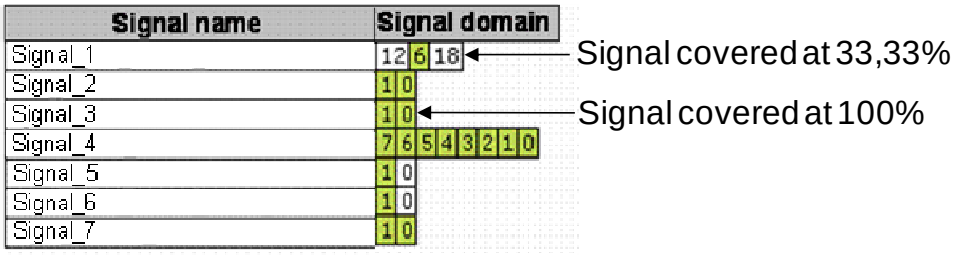


Figure 7.8 – Signals domain coverage

2. The coverage rate of an operation matrix

The *operation matrix coverage* of a requirements model consists of the *coverage rate* of all successions between pairs of *operations* visited. Once a succession probability is set between each two *operations*, we define a *coverage rate* of the critical successions where the succession probability is above a certain level defined by the engineer. In *Figure 7.9*, we illustrate the *coverage* of an *"operation matrix"* by a practical example. After generating a *test case*, some cases of the matrix have been highlighted. In fact, in the generated *test case*, the *operation #4* has followed the *operation #1*, the *operation #2* has followed the *operation #2*, the *operation #2* has followed the *operation #3* and so on. This way, we compute the *coverage rate* of successions between pairs of *operations* (around 38%; 5 successions of *operations* were covered over 13 possible successions)

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

Covered succession of operations

Non-covered succession of operations

Figure 7.9 – Operation matrix coverage

3. The coverage rate of an element (DT or FSM)

The *element coverage* of a requirements model consists of the *coverage rate* of the *conditions* of each *Decision Table* (*DT*) and the *coverage rate* of the *states*, *transitions* and *conditions* of each *Finite State Machine* (FSM). In *Figure 7.10*, we illustrate the *coverage* of a *DT* by a practical example. After generating a *test case*, some *conditions* of the *DT* have been highlighted. In fact, the *conditions* are covered at 75% (3 *conditions* visited over 4 *conditions* in total).

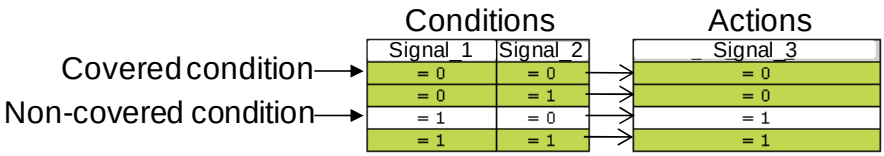


Figure 7.10 – Decision Table coverage

In *Figure 7.11*, we illustrate the *coverage* of a *FSM* by a practical example. After generating a *test case*, some *states* and *transitions* of the *FSM* have been highlighted. In fact, the *states* are covered at 75% (3 *states* visited over 4 *states* in total). The *transitions* are covered at 43% (3 *transitions* visited -different from 0%- over 7 *transitions* in total). The *conditions* are covered at 29% ((50%+0%+100%+0%+50%+0%+0%)/7=29%).

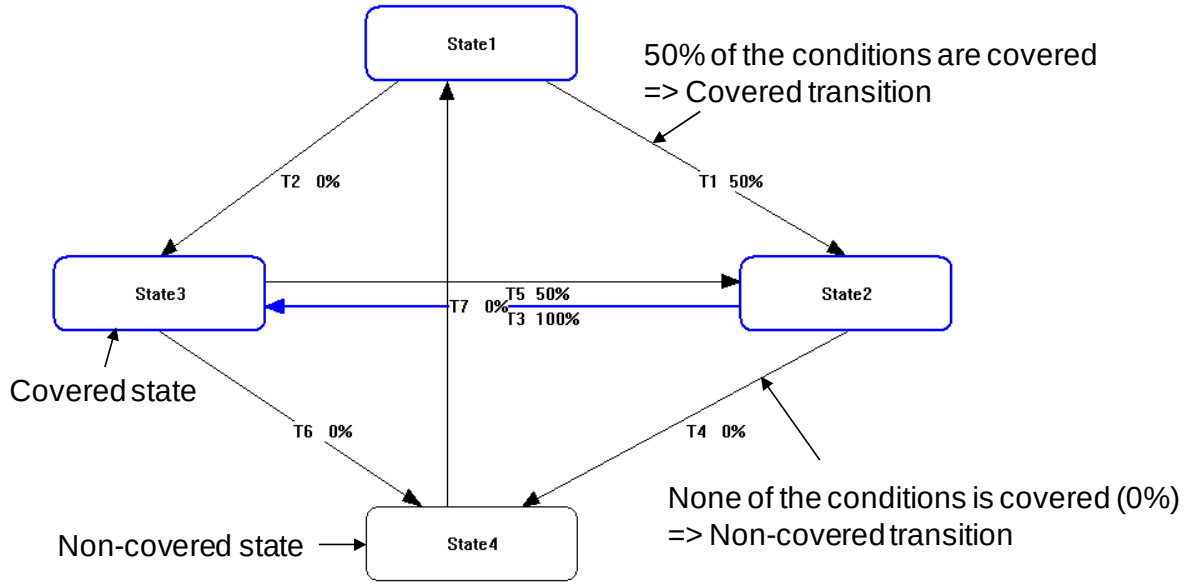


Figure 7.11 – Finite State Machine coverage

Moreover, when designing the requirements model, engineers can affect to *conditions*, *states* and *transitions* a normalized criticity level between 0 and 1. Consequently, we define a second set of *coverage rates* for expressing the degrees of *coverage* of the most critical *conditions*, *states* and *transitions* (in fact, this is a *weighted coverage* of an *element*).

C. Test execution time and cost constraints

Presently, in the automotive industry, the time and money spent to test a software product is the major criterion to stop testing. We have time and money spent to analyze carmakers' requirements, to design *test cases* and to execute *test cases* on the software product under test (Cf. *Chapter 2 – Section 6*). In our approach, we generate *test cases* automatically and therefore, one can have a tendency to generate too many tests. However, executing *test cases* on the software product under test and analyzing the results can cost too much time and money (Cf. *Chapter 4 – Section 4.C.1*) and more especially when the execution is performed manually by a test engineer (Cf. *Chapter 2 – Section 5.D.1*). In our approach, when generating a *test case*, test engineers can set targets not to be exceeded (constraints) on time and cost indicators:

- *Indicator 1: Execution time*. The time that a test engineer will spend in executing manually the generated *test case* on the software product.
- *Indicator 2:* Number of *test steps* in the generated *test case*.

- *Indicator 3:* Number of "distinct" *test steps* in the generated *test case*. Two *test steps* are distinct if they perform different *operations*.

Let us consider the *test case* of the *Figure 7.12*. The total *execution time* is 1150 ms (250ms+200ms+300ms+400ms=1150ms, around 19 seconds). The *test case* is composed from 4 *test steps* and the number of "distinct" *test steps* is 3 (*Test Step 1* and *Test Step 4* perform the same *operation "I1=1"*).

Test Case		
Test Step No	Test Actions	Expected Results
1	I1 = 1 Wait 250 ms	O1 = 0 O2 = 1
2	I2 = 2 Wait 200 ms	O1 = 0 O2 = 1
3	I3 = 1 Wait 300 ms	O1 = 1 O2 = 1
4	I1 = 1 Wait 250 ms	O1 = 0 O2 = 1

Figure 7.12 – An example of test case

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

V. Our stopping aggregated criterion

A. The objective function combining objectives and constraints

Based on the *coverage* objectives and the time and cost constraints identified in *Section 4*, we develop a panel interface to allow the test engineers to set precise targets on the test generation objectives and constraints (Cf. *Figure 7.13*). The quality objectives (*code* and *requirement specification coverage*) are expressed in terms of ratios of *coverage* and, then, are normalized which aim to reach a value of 100%. The *execution time* constraint is expressed in *millisecond* (*ms*). In addition, we define a set of weights (*wi*) that test engineers can associate for each defined target: 0 (to be ignored), 1 (not very important), 5 (important), 10 (very important). The panel presented in *Figure 7.13* helps test engineers to express their targets in terms of the required software quality and tests cost and therefore generate *test cases* fulfilling their objectives and constraints.

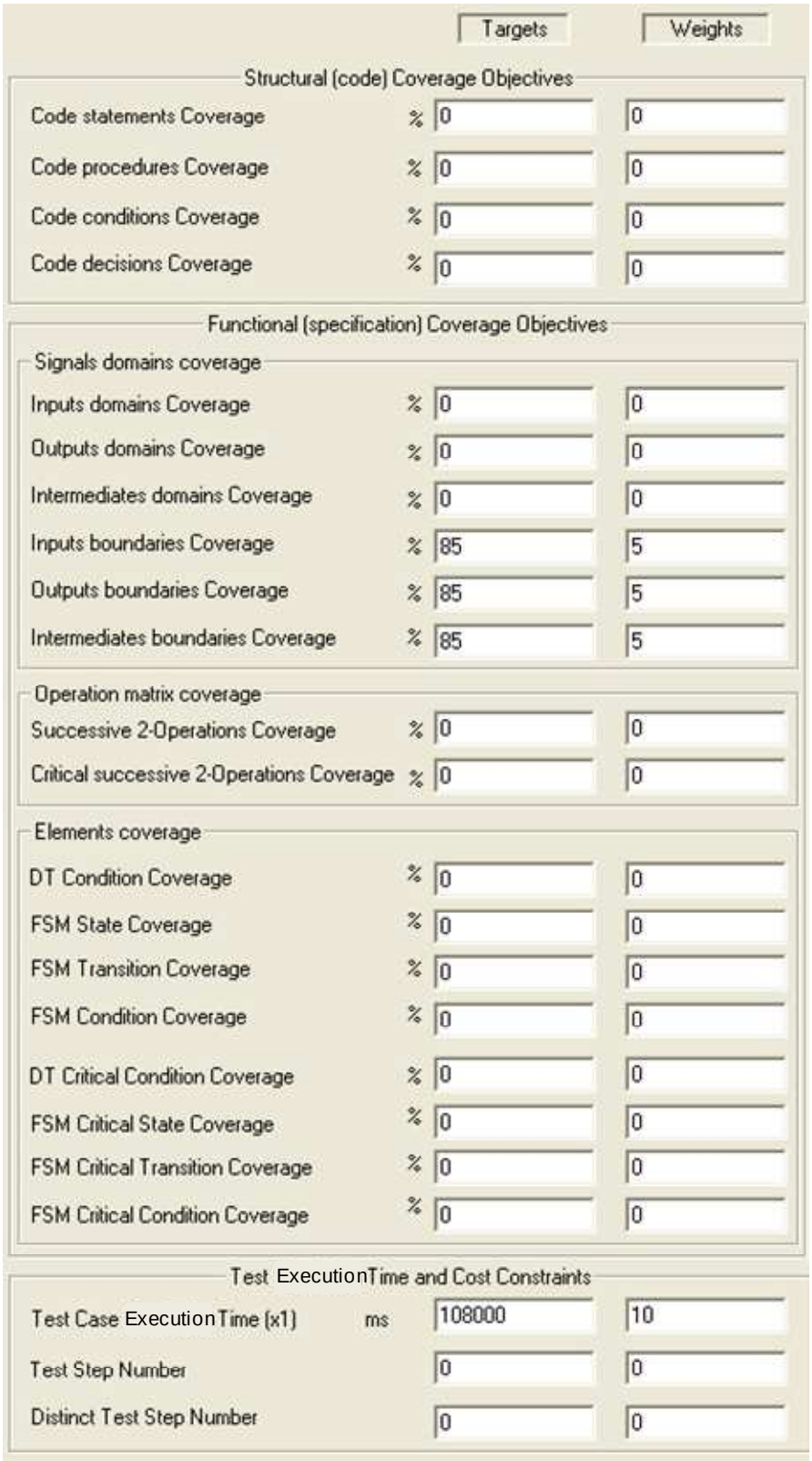


Figure 7.13 – Panel of the quality, time and cost indicators for monitoring the automatic generation of test cases

In fact, through our approach, the automatic generation of tests is monitored by the set of targets and weights predefined for each of the quality, time and cost indicators. During one test generation session, the targets may be completed following different orders and the first

target completed does not immediately stop the process. We stop only when the *aggregated preference*, *F*, defined as:

$$F = \underbrace{\sum \left| O_{T\arg et} - O_{Current} \right| \times w_i}_{F_{objectives}} + \underbrace{\sum \left| C_{T\arg et} - C_{Current} \right| \times w_i}_{F_{constraints}}$$

where *Os* are the *coverage* objectives, *Cs* are the *normalized* time and cost constraints and w_i*s* are weights, attains zero or does not decrease for a certain number of successive generated *test steps*. This number is one of 8 parameters of the *heuristic algorithm* used to optimize the *objective function* (*F*) when generating a *test case*. The algorithm and its parameters are described in *Section 6*. Since the objectives *Os* are normalized to 100% and in order to have a consistent *aggregated preference* (*F*), we normalize to 100% the time and cost constraints (*test case execution time, test step number, distinct test step number*). These constraints are expressed in *millisecond* (*ms*) and in number of generated *test steps*. We illustrate our normalization process of these constraints through an example. At each time, test engineers decide to set a constraint *ci*, the normalized target of this constraint $C_{Target}(ci)$ is immediately set to 100%. For instance, once a test engineer decide to generate a *test case* that the total *execution time* do not exceed 108000 *ms* (value set in the panel of quality, time and cost indicators, Cf. *Figure 7.13*), the normalized target of the *test execution time* constraint is set to 100% (C_{Target}*(test execution time)*= 100%). After generating a set of *test steps*, the normalized current value of this constraint ($C_{Current}(ci)$) is assessed by calculating the ratio (*current_constraint_value*100/target_constraint_value*). In our example, we generate a set of *test steps* with a total *execution time* of 21600 *ms*. Therefore, $C_{Current}$*(test execution time)* is assessed to (21600*100)/108000 ($C_{Current}$*(test execution time)*= 20%).

B. A simple example to illustrate our stopping aggregated criterion

Let us consider a practical *software testing* problem in order to illustrate the purpose of our objective function. Through the *experience feedback* of the *software testing* experts, some software bugs often occur when a signal is set to its *boundaries values*. Consequently, test engineers could always decide to generate a *test case* (a set of test steps) which aims to detect potential bugs related to the *boundaries values*. Hereafter, we consider the functionality which consists in managing the *front wiper* in a vehicle. The corresponding software component is made of 1229 *Lines Of Code* (blank and comment lines excluded), 18 *input signals* and 8 *output signals*. We decide to generate a *test case* fulfilling the following targets and weights in terms of coverage objectives (Cf. *Figure 7.13*):

- Cover the *boundaries input signals* at 85% with a weight of 5
- Cover the *boundaries output signals* at 85% with a weight of 5
- Cover the *boundaries intermediate signals* at 85% with a weight of 5

While respecting the constraint:

- Do not exceed 30 minutes (108000 ms) of tests execution with a weight of 10

After generating a *test case* with the objectives and constraints defined below, a *test report* is generated automatically. An excerpt of this report is illustrated in *Figure 7.14*. In this report, the reached (current) values on the objective and constraint indicators are illustrated. In fact, even if the *inputs and outputs boundaries coverage* have respectively reached and exceeded their targets (respectively 85% and 94% of *coverage*), our optimization algorithm did not stop

the generation of *test steps* expecting that the *intermediate boundaries coverage* reaches its target. But once the maximum *test execution time* which has a weight of 10 (very important) has been exceeded (110255 ms instead of 108000 ms), the optimization algorithm decides to stop generating *test steps* even if the *intermediates boundaries coverage* is not already reached. The excess of a *constraint* out of its bounds is accounted for a penalty that irremediably increases the overall objective value.

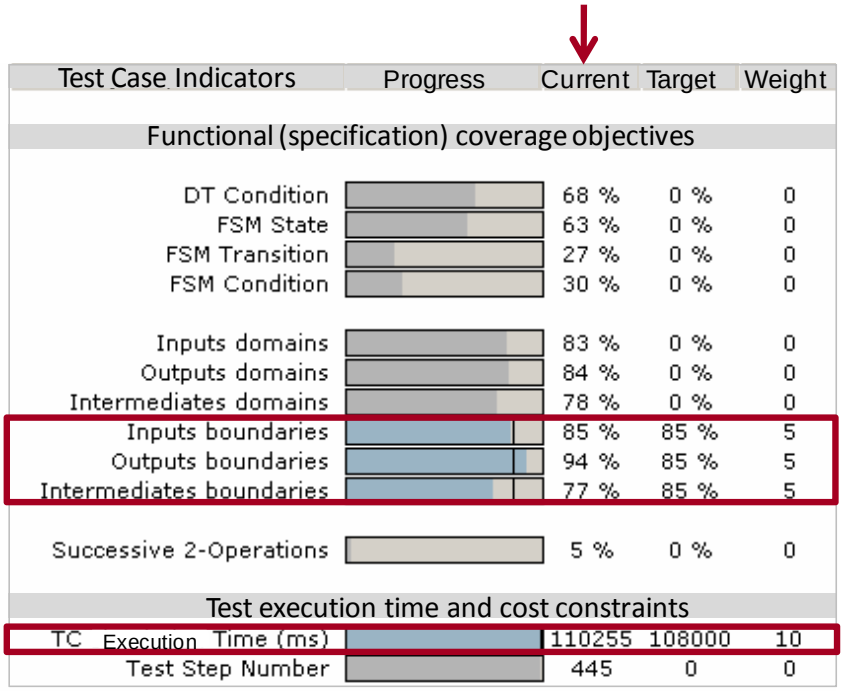


Figure 7.14 – An excerpt of the report delivered automatically after generating a test case

VI. Our heuristic algorithm to optimize the objective function

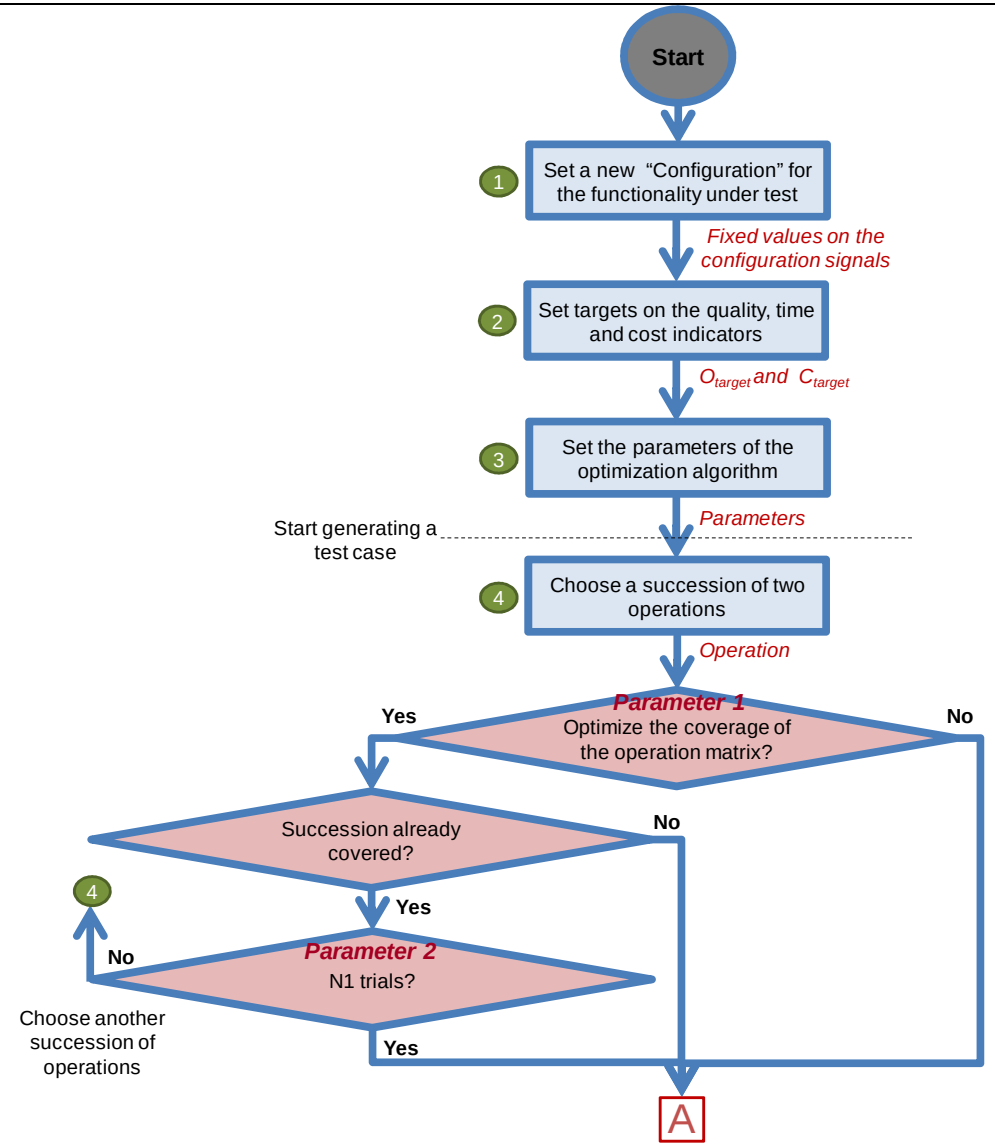
A. Process flow

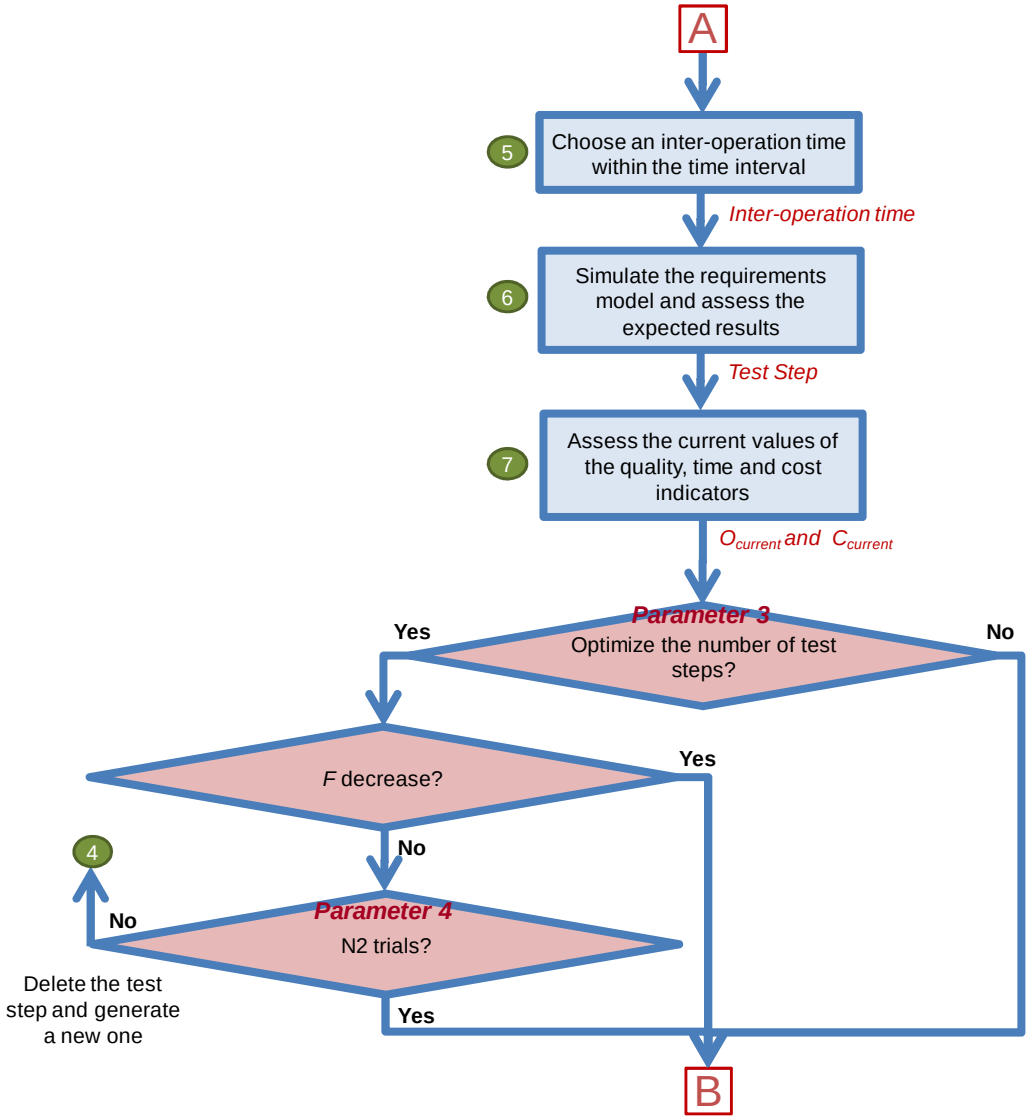
In conclusion, for a set of targets and weights on the *coverage* objectives and the time and cost constraints, the test engineer can generate one or more *test cases* fulfilling these predefined objectives and constraints. Afterwards, the "Optimal" *test case* is selected automatically. To do so, we compare the generated *test cases* in pairs and we select the one which has the lowest value of the *aggregated preference* $F_{objectives}$ of the quality (*coverage*) indicators. If the two *test cases* have the same value of $F_{objectives}$, we select the one which has the lowest value of the *aggregated preference* $F_{constraint}$ of the time and cost indicators. If the two *test cases* have the same value of $F_{constraint}$, we select the utmost one that respects each individual constraints going from the higher to the lower weights.

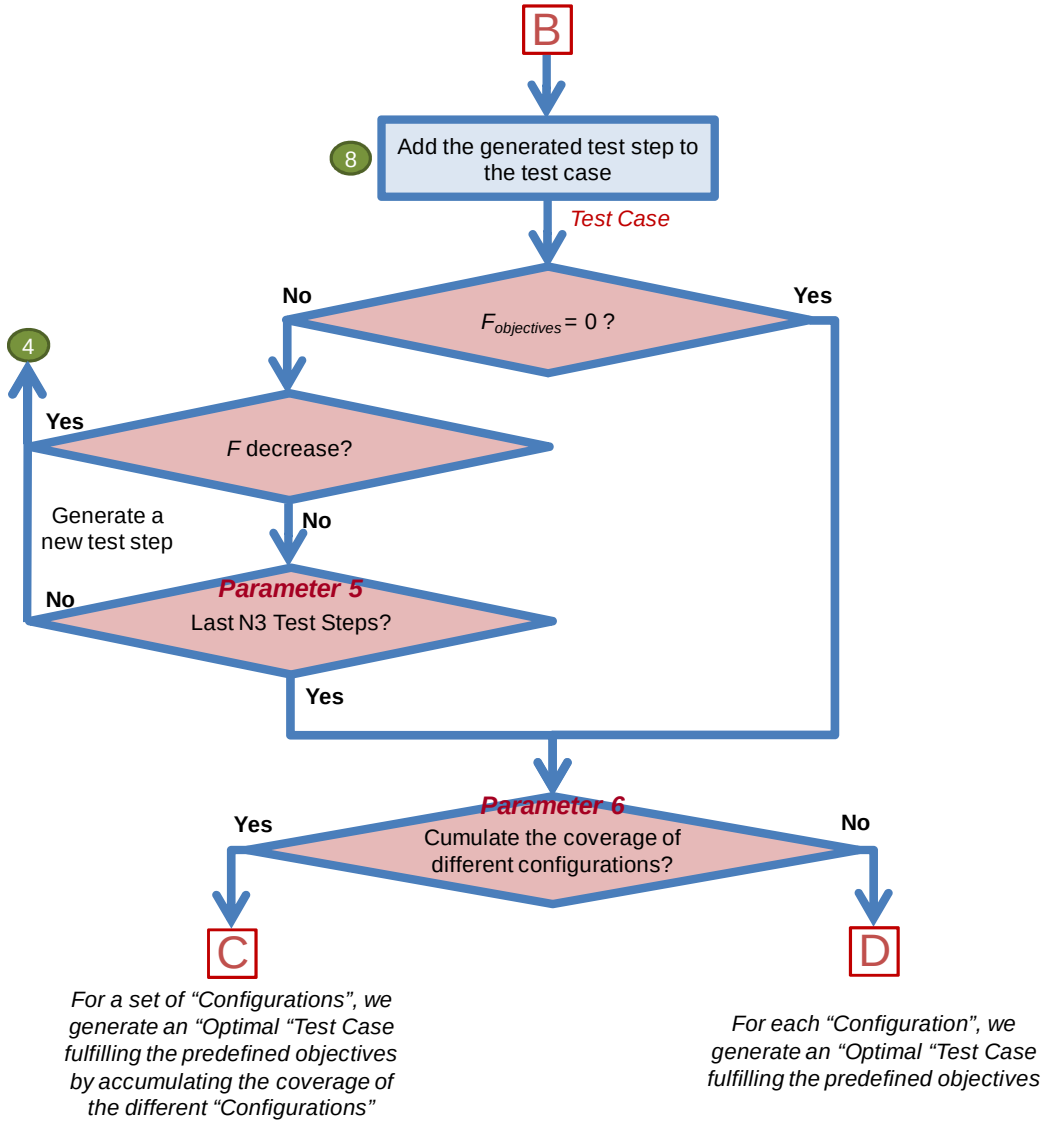
Moreover, a software functionality under test has often *configuration signals* (Cf. *Chapter 5 – Section 3.A*) which allow to parameterize the functionality (for instance, by activating or deactivating one feature of the functionality). A "*Configuration*" of a functionality consists to set all the *configuration signals* of the functionality to fixed values. Through our algorithm, two strategies are possible for managing the "Configuration" of a functionality. On the one hand, test engineers can generate one or more *test cases* for each specific "*Configuration*" of the functionality. The *configuration signals* are set to fixed values all over a *test case*. On the other hand, test engineers can generate one or more *test cases* where each *test case* considers a

set of predefined "*Configurations*" of the functionality. In this case, the values of the *configuration signals* can change from one *test step* to another within the same *test case*.

In *Table 7.1*, we describe the process flow of our optimization algorithm. The parameters of this algorithm are identified in *Table 7.1* (*Parameter i*) and described in the next section.

Process flow	Brief description
	Test engineers can decide to test a functionality under different "*Configurations*" (*Config1*, *Config2* …). We set a new "*Configuration*" for the functionality under test. Test engineers have to define the targets and weights for the *coverage* objectives and for the time and cost constraints Test engineers have to set the parameters of the optimization algorithm. We start generating a *test step* by choosing a succession of *operations* Did the test engineer decide to optimize the *coverage* of the *"operation matrix"*? If NO, we keep the chosen *operation* If YES, we check if the chosen *operation* is already covered by another *test step* in the *test case* If NO, we keep the chosen *operation* If YES, we check if we already made *N1* trials and all the chosen *operations* were already covered If NO, we choose a new succession *operation* If YES, we keep the last chosen *operation* ***To be continued …***

Process flow	Brief description
	We choose an *inter-operation time* within the time interval We set the chosen *operation* on the requirements model, we simulate the model during the *inter-operation time* and we assess the expected results on the *output signals* **A new *test step* is generated** We assess the current values of the *coverage* objectives and the execution time and cost constraints Did the test engineer decide to optimize the number of *test steps* in a *test case*? If NO, we keep the generated *test step* If YES, we check if the *aggregated preference F* has decreased If YES, we keep the generated *test step* If NO, we check if we already generated successively *N2 test steps* with no decrease of the *aggregated preference F*. If NO, we delete the last generated *test step* and we generate a new one If YES, we keep the last generated *test step* ***To be continued …***

Process flow	Brief description
	We add the test step to the current *test case* **The current *test case* is updated with a new *test step*** We check if the "set" *coverage* objectives have reached their targets ($F_{objectives}$=0) If YES, we stop generating *test steps* for the current *test case* If NO, we check if the *aggregated preference F* has decreased If YES, we generate a new *test step* for the current *test case* If NO, we check if the *aggregated preference F* have not been decreased with the *N3* last *test steps* of the current *test case* If NO, we generate a new *test step* If YES, we stop generating *test steps* for the current *test case* Did the test engineer decide to cumulate the *coverage* of different "*Configurations*" in order to reach the targets on the *coverage* objectives? If YES, follow C If NO, follow D ***To be continued …***

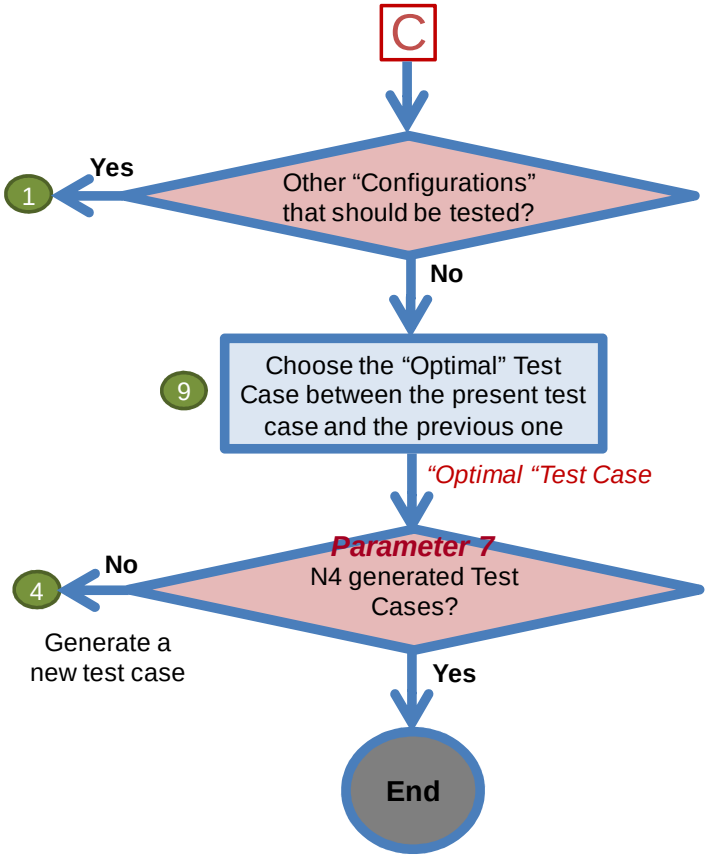
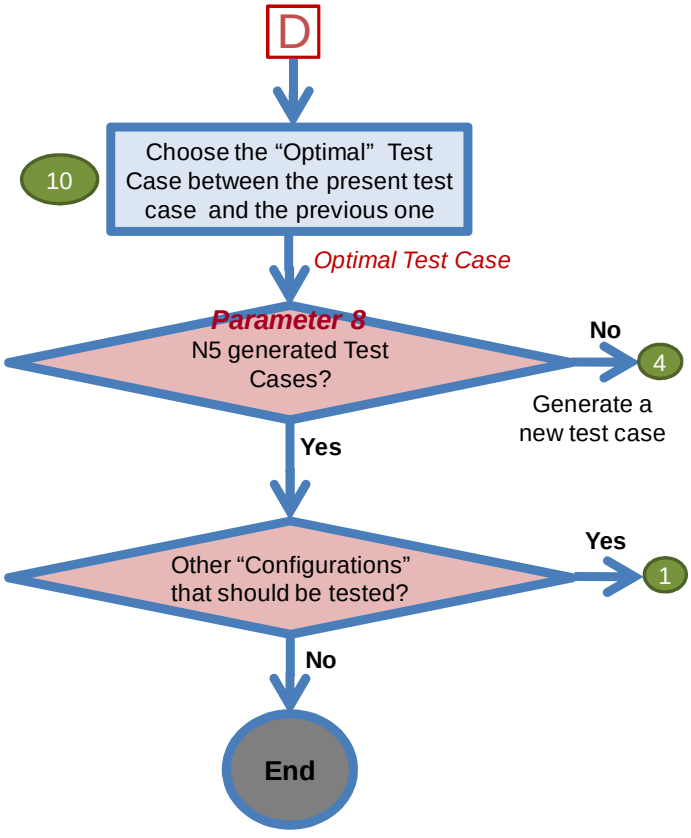
Process flow	Brief description
	Is there other "*Configurations*" for the functionality that should be tested? If YES, we change the "*Configuration*" and continue generating *test steps* for the same *test case* If NO, **the current *test case* is finalized** We choose the "Optimal" *test case* between the current *test case* and the previous one. **The "Optimal" *test case* is chosen** Do we generate *N4 test cases* in order to choose the "Optimal" one? If NO, we generate a new *test case* for the set of "*Configurations*" If YES, **we stop the algorithm and we deliver the "Optimal" *test case* for the set of "*Configurations*"**
	**The current *test case* is finalized** We choose the "Optimal" *test case* between the current *test case* and the previous one. **The "Optimal" *test case* is chosen** Do we generate *N5 test cases* in order to choose the "Optimal" one? If NO, we generate a new *test case* with the same "*Configuration*" If YES, we check if there is other "*Configurations*" for the functionality that should be tested? If YES, we change the "*Configuration*" and start generating a new set of *test cases* If NO, **we stop the algorithm and we deliver an "Optimal" *test case* for each "*Configuration*"**

Table 7.1 – Our heuristic algorithm to optimize the generation of test cases

B. Parameters

In the *Table 7.1*, we identify 8 parameters that must be set by the test engineer before start generating *test cases*:

- ***Parameter 1***: Test engineer can decide to optimize the *coverage* of the "*operation matrix"*. To do so, *Parameter 1* must be set to *1*. Otherwise, it is set to *0*.
- ***Parameter 2***: In order to optimize the *coverage* of the "*operation matrix"*, we check if the chosen succession of *operations* is already covered or not. When it is already covered, we propose to choose another succession of *operations* and so on. However, we have to avoid the non-stop loop in the algorithm. The *Parameter 2* specifies the maximum number of unsatisfied trials *(N1)* before the algorithm exists the loop.
- ***Parameter 3***: Test engineer can decide to optimize the number of *test steps* in a *test case*. To do so, *Parameter 1* must be set to *1*. Otherwise, it is set to *0*.
- ***Parameter 4***: In order to optimize the number of *test steps* in a *test case*, we check if a generated *test step* decreases the *aggregated preference F*. In case of no decrease of *F*, we propose to delete the *test step* and generate a new one. However, we have to avoid the non-stop loop in the algorithm. The *Parameter 4* specifies the maximum number of unsatisfied trials *(N2)* before the algorithm exists the loop.
- ***Parameter 5***: In case of generating one or more *test cases* for each specific "*Configuration*" of the functionality, this parameter allows to stop generating *test steps* for each *test case*. In fact, we check if the *aggregated preference F* has not been improved on the last *(N3) test steps* of the current *test case*. If it is the case, we stop generating *test steps*. It not, we continue generating *test steps*.
- ***Parameter 6***: Test engineer can generate one or more *test cases* where each *test case* considers a set of predefined "*Configurations*" of the functionality. To do this, *Parameter 6* must be set to *1*. When this parameter is set to *0*, each generated *test case* considers only one specific "*Configuration*".
- ***Parameter 7*** (used when *Parameter 6* = 1): This parameter defines the number of *test cases* (*N4*) to be generated in order to identify the "Optimal" one.
- ***Parameter 8*** (used when *Parameter 6* = 0): This parameter defines the number of *test cases* (*N5*) to be generated in order to identify the "Optimal" one.

VII. Conclusion

In automotive industry, the activity of designing manually *test cases* for software products becomes more and more laborious and time consuming. This activity accounts for more than 50% of the total software project time and budget (Cf. *Chapter 1 – Section 5.C.2*). Despite the huge time and money spent in testing a software product and after each delivery to the carmaker, some bugs are detected by the carmaker. Since the late 90's, the automation of the *test case design process* became a hot topic and automotive electronic suppliers are still looking for a relevant automation of this process.

In this chapter, we developed our strategy to generate *test cases* automatically from our *formal* model to represent software functional requirements (Cf. *Diagnosis 15*). The selection of *operations* is performed based on a *Monte Carlo simulation* on an "*operation matrix*" (Cf. *Diagnosis 13*). All the expected values on the *output signals* of the functionality are assessed through a simulation of the requirements model (Cf. *Diagnosis 10 and 12*). Moreover, test engineers could parameterize the generation of *test cases* in order to take into account quality objectives but also time and cost constraints. Indeed, the decision to stop designing *test cases* is based on a *formal* measurement of the *code* and *requirement coverage* and the test time and

cost (Cf. *Diagnosis 11*). A *heuristic algorithm* is in charge of optimizing the generation of *test cases* while fulfilling quality objectives and constraints.

In the following chapter, we suggest refining the *operation space* by focusing on critical *operations* or succession of *operations*. To do this, we define *driver behavior's profile* and propose to reuse *bugs* and *test cases* capitalized on similar projects in the past.

CHAPTER 8. REFINING THE OPERATION SPACE DESCRIPTION WITH THE DRIVER BEHAVIOR'S PROFILE, PAST BUGS AND TEST CASES

I. Introduction

As automotive software products become more and more complex (Cf. *Chapter 1*), it is illusory to be able to check that the software product responds correctly to all possible *operations*. In other words, it is impossible to cover all the *operation space* of a software product (Cf. *Chapter 2 – Section 6*). In fact, each engineer has a different perception of the possible and critical *operations* (based on her/his experience). When designing *test cases*, test engineers aim to reach a code or *requirement coverage* objective. In fact, test engineers do not always select *operations* that simulate the *real use* of the software product under test. Moreover, they do not *formally* use capitalized bugs and *test cases* in order to improve the *test design process* on future developments.

In this chapter, we point up how the *operation space* can be objectively refined by focusing on critical test scenarios. The complexity of testing exhaustively a software product is illustrated in *Section 2*. An overview on our *operation space* reducing techniques is proposed in *Section 3*. In *Section 4*, *5* and *6*, we develop respectively each of these techniques: focusing on test scenarios regularly done by the *end-user* of the product, focusing on recurrent types of bugs through an analysis of the *problems' database* and finally focusing on test engineers' *experience feedback* by reusing *test cases* capitalized on previous projects.

II. The impossibility of testing exhaustively a software product

Testing exhaustively a software product is a *NP-Complete* problem from a computational viewpoint. In other words, it is very complex to test all the *inputs*, *combinations of inputs* and *paths* of a software. In computational *complexity theory*, the complexity class *NP-Complete* also known as *NP-C* or *NPC*, is a subset of the *NP* class ("*Non-deterministic Polynomial time*" class, (Karp 1972)). They are the most difficult problems in *NP*. To prove that an *NP* problem *A* is in fact an *NP-Complete* problem it is sufficient to show that an already known *NP-Complete* problem reduces to *A*. There are more than 3000 known *NP-Complete* problems. Most of the problems are listed in Garey and Johnson's seminal book (Garey 1979). In (Seroussi 1988), Seroussi and Bshouty prove that the design of an optimal exhaustive *test case* for an arbitrary logic circuit is an *NP-Complete* problem. In fact, they demonstrate that finding the minimal *test case* (and its size) which covers the logic circuit is an *NP-Complete* problem. In order to do this, they first show that the problem can be solved by a nondeterministic algorithm in polynomial time. Then, they use the standard technique of reduction to prove that the problem is *NP-Complete*: for a given problem *P* (the *Graph Coloring problem*[34]) known to be *NP-Complete*, they show that if our problem is solvable in deterministic polynomial time, then so is *P*. In (Cheng 2003, Hessel 2007), the authors prove that finding optimal *test cases* for a software product is an *NP-Complete* problem. Indeed, they reduce the problem of generating *test cases* to the *set-covering problem* (an *NP-Complete* problem).

For our research project, we propose to generate automatically *test cases* for a software product. Our approach is based on modeling the software functional requirements and generating *test cases* from this model. To guide the design of *test cases*, *code* and/or *requirement coverage* criteria are used. A *coverage* criterion can be seen as a set of items (relations between inputs and outputs) in the *source code* or *requirements model* to be covered. Therefore, our test generation problem can be formulated as a *Reachability*

[34] Graph coloring problem, http://en.wikipedia.org/wiki/Graph_coloring (Consulted on November 2008)

problem[35] (an *NP-Complete* problem) which consists to explore the *operation space* only if it might increase the total *coverage*. In fact, a *test case* is a set of ((*TS1*,*Cov1*), (*TS2*, *Cov2*), (*TS3*, *Cov3*), … (*TSn*, *Covn*)), where *TSi* is a *test step* and *Covi* is the *coverage* contribution performed by *TSi*. Ideally, this set should be reduced so that the total *coverage ΣCovi* is not changed, and the length of the *test case*, e.g. *Σ|TSi|* is minimized. However and as it was shown above, designing a subset of *test steps* with this property is an *NP-Complete* problem (the *Reachability problem*).

A present, all known algorithms for *NP-Complete* problems require time that is *superpolynomial* (for instance, exponential) in the inputs size, and it is unknown whether there are any faster algorithms. The following techniques can be applied to solve computational problems in general and they often give rise to substantially faster algorithms:

- *Randomization:* Use *randomness* to get a faster average running time, and allow the algorithm to fail with some small probability.
- *Heuristic:* An algorithm that works "reasonably well" on many cases, but for which there is no proof that it is both always fast and always produces a good result. In *Chapter 7 – Section 6*, we develop the *heuristic algorithm* that we use in order to explore the *operation space* of a software product and monitor the generation of *test cases*.

In the next sections of this chapter, we develop how to reduce the *operation space* of a software product by highlighting and eliminating some operations or succession of *operations*. Our purpose is to explore the *operation space* of a software product efficiently.

III. Reduce the operation space

As said in the previous section, selecting *operations* from the whole *operation space* in order to reach a coverage objective is a *NP-Complete* problem. For that reason, it can be useful to reduce the *operation space* by:

A. Focusing on recurrent operations done by the end-user of the product

We analyzed in 2006 a set of software bugs (the number of these bugs is confidential) detected on different types of products by carmakers and end-users (drivers) and not detected by Johnson Controls. The conclusion which was validated by Johnson Controls software experts is that some of these bugs (more than 50%) can only be detected via successions of *operations* regularly done by the end-user of the product. Therefore, we propose to generate *test cases* that simulate the behavior of the end-user of the product. To do so, *test cases* must be generated from *"operation matrices"* where illogic (from end-users' viewpoint) successions of *operations* are eliminated (for instance, set the vehicle speed at 100 km/h then open the trunk) and regular successions of *operations* are favored (for instance, close the driver door and set the ignition). Our process to define a *driver behavior's profile* is developed in *Section 4*.

B. Focusing on specifics operations with high probability to detect bugs

We performed in 2007 a study on 70 software bugs detected on the same functionality developed in Johnson Controls for 5 different projects respectively in 1997 (27 bugs), 2001 (4

[35] Reachability problem, http://en.wikipedia.org/wiki/Reachability (Consulted on November 2008)

bugs), 2003 (4 bugs), 2003 (13 bugs) and 2007 (22 bugs) (Cf. *Chapter 2 – Section 7.C*). The studied functionality has 7 *features*, so we classified these bugs by project and by *feature* (Cf. *Figure 2.22*). We also classified these bugs by project and by type of problem (Cf. *Figure 2.23*). In fact, we only consider 4 types of problem (Beizer 1990, Chillarege 1992, Grady 1992, IEEE Std. 1044-1993): *code implementation*, *control flow*, *data* and *processing*. A full typology of software bugs is described in *Section 5.B*. The conclusions which were validated by Johnson Controls software experts are 1) engineers have the tendency to make errors in implementing the same *features* of a functionality (*Feature* 1, 3 and 7) and 2) these errors are related to the same types of problem (*Control flow* and *processing*). As a consequence and when testing a functionality, we propose to reuse related stored bugs in order to generate *test cases* which verify the nonexistence of recurrent bugs. To do so, *test cases* must be generated from *"operation matrices"* where the successions of *operations* that reveal the recurrent bugs are favored. Moreover, classifying the recurrent bugs of a functionality (by *feature* and/or by type of problem) could help the test engineers to better focus the generation of *test cases* on critical *features* or on specific types of problem. More details on reusing stored bugs are provided in *Section 5*.

C. Focusing on specifics operations recurrently done by the test engineers on previous projects

Using capitalized *test cases* seems to be beneficial in automotive context since more than 50% of functionalities performed by software products are common to any series of cars (Johnson Controls source). Moreover, *test cases* management and reuse are considered as one of the main characteristics of a mature software organization. Therefore, when testing a functionality that we already implemented in the past on another project, it is judicious to reuse existing *test cases*. To do so, we propose to analyze *test cases* developed in the past for the same functionality and design *"operation matrices"* where the *operation space* is reduced by focusing on the test scenarios based on our returns of experience. *Test cases* generated from these *"operation matrices"* contain similar successions of *operations* as in the one designed manually or generated automatically in the past. More details on reusing capitalized *test cases* are provided in *Section 6*.

IV. Four types of constraints for the definition of a driver behavior's profile

We define four types of *constraints* that test engineers can affect to each *input signal* of a requirements model in order to, when generating *test cases* automatically, eliminate or favor specific successive *operations*. Each *input signal* can have one or more *constraints*. These *constraints* aim to reduce the number of possible combinations on *input signals* and to more thoroughly pinpoint which ones are frequently set once the product is launched on the market. These four *constraints* are: *logical constraint*, *conditional constraint*, *succession constraint* and *timing constraint*.

A. Logical constraint

This constraint forbids that an *input signal* switches between inadequate values from a use point of view. In order to illustrate this constraint, let us consider the input signal *I1*, which has a domain *D(I1)={0,1,2,3}*. We classify *input signals* into two types:

Acyclic (Cf. Figure 8.1): *Input signal I1* is acyclic if, at any moment, all the *operations* (*I1=0* or *I1=1* or *I1=2* or *I1=3*) on the signal are possible. A practical example of an *acyclic signal*

is the *rain intensity signal* measured via a sensor. When it is raining so hard (*I1=3*), it can stop raining (*I1=0*) at any moment without a decrease of the raining intensity (*I1=3 → I1=2 → I1=1 → I1=0*).

I1	0	1	2	3
0	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1

Figure 8.1 – Acyclic signal

Cyclic (Cf. Figure 8.2): *Input signal I1* is cyclic if the future *operation* (*I1=0* or *I1=1* or *I1=2* or *I1=3*) on the signal depends on the one did in the past. A practical example of a cycle signal is the *wipers' switch* signal. When wiping at a high speed (*I1=3*), user cannot immediately switch off the wipers (*I1=0*) via the switcher. In fact, she/he must progressively slow down the wiper speed until the complete stop (*I1=3 → I1=2 → I1=1 → I1=0*).

I1	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	0	1	1	1
3	0	0	1	1

Figure 8.2 – Cyclic signal

B. Conditional constraint

This constraint characterizes a specific user behavior between two or more correlated *input signals*. In other words, when one or more *inputs* fulfill specific conditions, the domain of other *inputs* is adapted (shrinked) automatically. For instance (Cf. *Figure 8.3*), the vehicle speed cannot be more than 0 (*I1>0*), only if the vehicle is running (*I2=1)* and the vehicle can be running (*I2=1*) only if the car engine is switched on (*I3=1*).

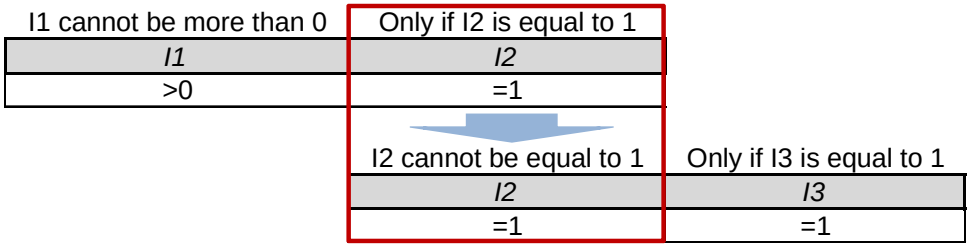


Figure 8.3 – Conditional constraint

C. Succession constraint

In practical use of an electronic product, two or more *operations* have a high probability to succeed (sometimes, must intuitively succeed). Through this type of constraint, we favor such successive *operations*. For example (Cf. *Figure 8.4*), when drivers close the driver door (*I1=1*), they often (with a probability of 0,75) switch on the car engine (*I3=1*).

		I1		*I2*			***I3***	
		0	1	1	2	3	0	**1**
I1	0	…	…	…	…	…	…	…
	1	0,04	0,04	0,04	0,04	0,04	0,04	0,75
I2	1	…	…	…	…	…	…	…
	2	…	…	…	…	…	…	…
	3	…	…	…	…	…	…	…
I3	0	…	…	…	…	…	…	…
	1	…	…	…	…	…	…	…

Probability only

Figure 8.4 – Succession constraint

D. Timing constraint

Major Johnson Controls software experts approve that time interval between *operations* plays a major role in bugs' detection. *On the one hand*, two specific *operations* can be performed with a specific time interval (Cf. *Figure 8.5*). For instance, in case of a taxi driver, the driver door is closed (*I1=1*) and the car engine is switched on (*I3=1*) within a small time interval ([50ms[36], 100ms] according to experts).

		I1		*I2*			***I3***	
		0	1	1	2	3	0	**1**
I1	0	…	…	…	…	…	…	…
	1	…	…	…	…	…	…	[50,100]
I2	1	…	…	…	…	…	…	…
	2	…	…	…	…	…	…	…
	3	…	…	…	…	…	…	…
I3	0	…	…	…	…	…	…	…
	1	…	…	…	…	…	…	…

Inter-operation time interval only

Figure 8.5 – A specific time interval between two operations

On the other hand, a specific *operation* can be performed during a specific time (Cf. *Figure 8.6*). For instance, the ignition is switched off (*I3=0*) for more than 5 seconds (according to experts) in order to reset a functionality.

		I1		*I2*			*I3*	
		0	1	1	2	3	0	1
I1	0	…	…	…	…	…	…	…
	1	…	…	…	…	…	…	…
I2	1	…	…	…	…	…	…	…
	2	…	…	…	…	…	…	…
	3	…	…	…	…	…	…	…
I3	**0**	[5000,5000]	[5000,5000]	[5000,5000]	[5000,5000]	[5000,5000]	[5000,5000]	[5000,5000]
	1	…	…	…	…	…	…	…

Inter-operation time interval only

Figure 8.6 – An operation set during a specific time

V. Reuse of bugs detected on previous projects

Each bug stored in the *bug's database* has a set of 111 attributes "theoretically" filled by the engineer while resolving the bug. In *Chapter 2 – Section 7.B*, we estimate that 75% of these

[36] ms: millisecond

attributes are filled; the remaining 25% are systematically unfilled. On the 75% filled attributes, 25% of these attributes are free fields. Moreover, we deduce that the *problems' database* in Johnson Controls is mainly used to manage the bugs and keep their traceability. Unfortunately, none of the 111 attributes is useful to identify critical succession of *operations* or recurrent types of problem for a specific functionality. In this section, we propose two strategies in order to reuse bugs detected on previous projects. *The first strategy* consists of defining a specific format to capitalize the initial conditions and the successive *operations* which lead to detect a bug. *The second strategy* aims to define a detailed typology of software problems.

A. A specific format to capitalize the successive operations leading to a bug

Now at Johnson Controls, engineers describe how the bug was detected by filling a free field in the *problems' database* named "*Problem description*" (Cf. *Figure 2.19*). Indeed, apart the requirement of using the English language, no other requirements or recommendations for filling this field are given to database users. In fact, each engineer has to describe the way the bug was detected by giving as much detail as possible. In *Figure 8.7*, we propose a new specific format to describe the successive *operations* leading to a bug.

Problem description	
Initial values on input signals	*The initial state of the functionality under test. Here, practitioner must list all the initial values on the functionality input signals*
Step 1	
First operation	*The first operation. Here, practitioner must put nothing or an input signal set to a specific value*
Inter-operation time (ms)	*The waiting time before performing the second operation. Here, practitioner must put a time in millisecond*
Expected values on output signals	*The expected values on the functionality output signals. Here, practitioner must list all the expected values on the functionality output signals*
Observed values on output signals	*The observed values on the functionality output signals. Here, practitioner must list all the observed values on the functionality output signals*
Step 2	
Second operation	
Inter-operation time (ms)	
Expected values on output signals	
Observed values on output signals	
Step i	
i^{th} operation	
Inter-operation time (ms)	
Expected values on output signals	
Observed values on output signals	
Step n	
n^{th} operation	
Inter-operation time (ms)	
Expected values on output signals	
Observed values on output signals	

Figure 8.7 – A predefined format to fill in the "Problem description" attribute of a bug

Let us consider a practical example of a functionality with 3 *input signals* (*I1*, Domain = {0, 1}; *I2*, Domain = {1, 2, 3}; *I3*, Domain = {0, 1}) and two *output signals* (*O1*, Domain = {0, 1}; *O2*, Domain = {0, 1}). When testing this functionality on a project in 2005, a test engineer

has detected a bug and added this bug to the *bug's database*. In *Figure 8.8*, we illustrate how the "*Problem description*" attribute of this bug was filled in. In *Step 6*, the observed values on *output signals* are different from the expected values (this is the *symptom* of the bug). For each bug in the database, the functionality where the bug was detected is specified. Therefore, when testing the same functionality on a project in 2007 (2 years after), a test engineer could select from the *problems' database* all the bugs detected on this functionality on previous projects. Based on the predefined format of the "*Problem description*" attribute, each bug can be translated automatically into an *"operation matrix"* (Cf. *Figure 8.8*). A glossary of the *input signals* names on the previous and current projects is necessary. In fact, from one project to another, the name of an *input signal* can change even if the use of the signal stills the same. The *test cases* generated from this *"operation matrix"* (Cf. *Figure 8.8*) allow to check if the bug that we detected in the past is present or not in our new development of the functionality.

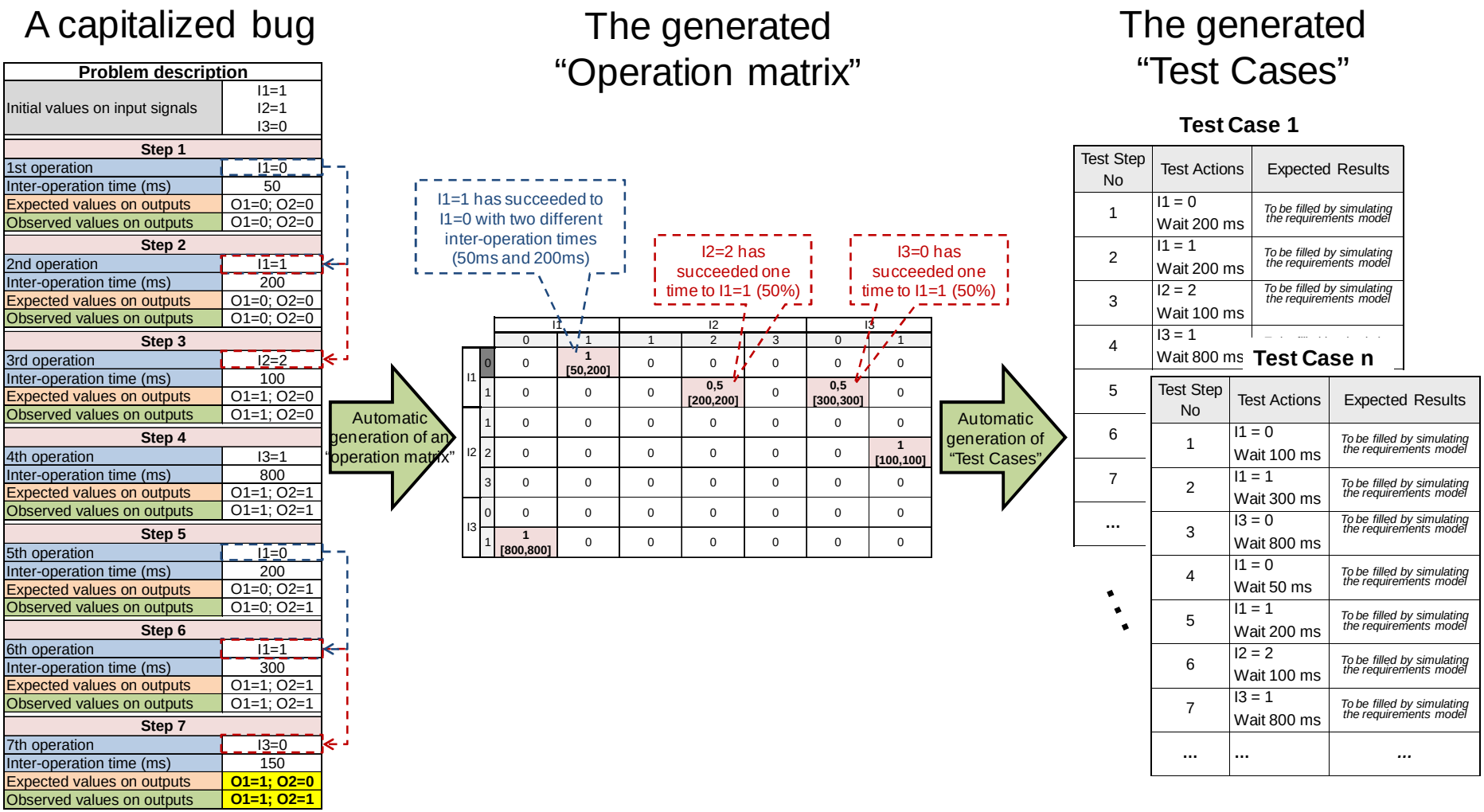


Figure 8.8 – Process of reusing bugs capitalized in the problems' database

In conclusion, for each functionality, test engineers can generate a set of *test cases* from the bugs detected on the same functionality on previous projects. Once executing these *test cases* on the new software of the functionality, test engineers can, at least, guarantee that the new development is free of the bugs already made in the past.

B. A new typology of software problems

A second way to reuse bugs stored in the *problems' database* is to analyze these bugs and identify the recurrent type of problems when implementing a software product. The "*Problem type*" captures the nature of the fix. It addresses the question: "What did the engineer correct in order to resolve the bug?" With this definition of "*Problem type*", the classification is easier for the engineer, since she/he can almost decide objectively which *attribute value* to assign. Presently, Johnson Controls doesn't have a typology of software problems (Cf. *Chapter 2 – Section 7.B*).

In 2007, we participated to a work group on the definition of a new *bug classification model* within Johnson Controls. The aim of this new model is to be able to identify process improvement actions for the development and *Verification and Validation* (*V&V*) processes. In other words, the new *bug classification model* must answer the question of "which types of bugs are injected and detected in which process phase?" The *classification model* currently used in Johnson Controls (Cf. *Chapter 2 – Section 7.B*) does not allow to answer this question since there is no typology of software problems. The new *bug classification model* that we propose is summarized in *Figure 8.9*. It is mainly inspired by the literature on the subject (Cf. *Chapter 4 – Section 4.C.2*). The new model is based on three attributes:

- *Detection phase:* the phase of the process where the bug was detected.
- *Injection phase:* the phase of the process where the bug was injected.
- *Problem/Correction type:* this attribute answers the question of "what did you fix in order to correct the bug?"

For each of these attributes, a set of predefined values was defined in accordance with the Johnson Controls software process (Cf. *Chapter 2*):

- *Detection phase*: Specification review, Design review, Code review, Component test, Integration test, Manufacturing test, Validation test, System test, Test review, Customer test.
- *Injection phase*: System Specification, Specification, System Design, Design, Implementation, Integration, Manufacturing, Testing phases (Component test, Integration test, Validation test, System test, Manufacturing test, Customer test).
- *Problem/Correction type*: A two-level typology of software problems is defined. The values of the first-level typology are: Specification update, Design update, Implementation update, Integration update, Manufacturing update, Test case update, Update none. The second-level typology is detailed in *Appendix E*.

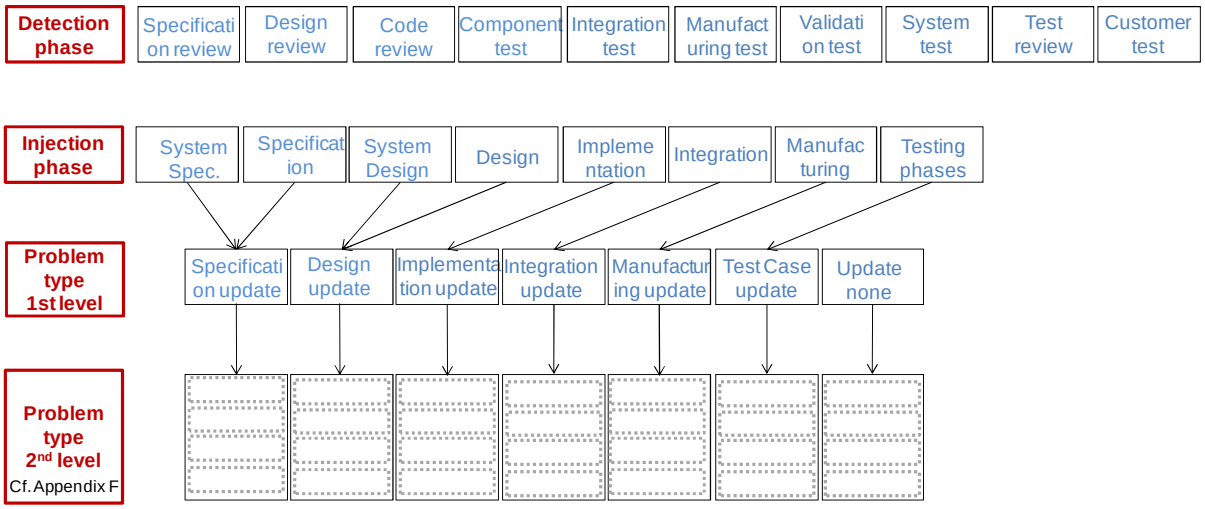


Figure 8.9 – Our new bug classification model

On the one hand, as in *HP* classification model (Chillarege 1992); the attributes *Injection phase* and *Problem/Correction type* are linked. Indeed, the choice of a value for the attribute *Injection phase* defines the possible values for the attribute *Problem/Correction type*. In practice, when an engineer is editing a bug in the *problems' database*, once she/he defines the *Injection phase* of the bug, the list of proposed *Problem/Correction type* must be adapted dynamically to the chosen *Injection phase*. For example, if the bug was injected in the Specification phase, the list of *Problem/Correction type* must be (Cf. *Appendix E*): requirements incorrect, requirements logic, requirements completeness … If the bug was injected in the *Implementation* phase, the list of *Problem/Correction type* must be (Cf. *Appendix E*): data definition, structure, declaration, data access and handling, control flow and sequencing …

On the other hand, at the end of each *design* phase, one or more *V&V* phases have the responsibility to detect all the type of problems injected in this phase (Cf. *Table 8.1*). The *validation and system test* are the last *V&V* phases before the software product delivery to the carmaker. These phases have to check the compliance of the developed software with the carmaker requirements. They have the responsibility to detect all the type of problems injected during the design and development phases and not detected by the corresponding *V&V* technique.

Bugs injected in	Must be detected in
System specification	Specification review *Validation and System test*
Specification	
System design	Design review *Validation and System test*
Design	
Implementation	Code review Component test *Validation and System test*
Integration	Integration test *Validation and System test*
Manufacturing	Manufacturing test *Validation and System test*
Testing phases	Test review

Table 8.1 – Theoretical bugs' injection and detection phases

However, some *V&V* activities (*Detection phases*) are not enough reliable to detect all related types of problems. The new *bug classification model* enables to pinpoint such lacks in the software process. In fact, managers can identify the density and type of software problems

injected in and detected by each phases of the software development life cycle. Therefore, improvement actions on the design phases (develop new design or development rules …) and on the *V&V* phases (develop new *code review* rules, a new testing strategy…) can be performed. The first-level typology of software problems is not enough detailed and the improvement actions that can be raised from will not be enough efficient. For instance, after analyzing a set of bugs related to *Specification Update* problems, one engineer can note that the total number of these bugs is injected in *Specification System* and *Specification* phases and few of them are detected in *Specification Review* phase. As a conclusion, the *Specification Review* process has to be improved. However, it is a vague action. Engineers need to know what they have to improve in the *review* process. Are the requirements unclear? Incorrect? Do the requirements change often? … We work in collaboration with software experts in order to define the second-level software problem typology. Based on the literature review (Beizer 1990, Chillarege 1992, Grady 1992, IEEE Std. 1044-1993) and taking into account the automotive and industrial context, we propose in *Appendix E* a detailed typology of problems.

Through our research project, a new approach to generate *test cases* automatically for a functionality is proposed. The generated *test cases* have the responsibility to detect all the bugs injected during the *Implementation* phase of the *source code*. Analyzing the bugs injected during the *Implementation* phase of the same functionality on previous projects allows test engineers to better parameterize the generation of *test cases*. Instructions to generate *test cases* able to detect one specific type of software implementation problems (Cf. *Appendix E*) are listed in *Table 8.2*.

Types of software implementation problems (Cf. *Appendix E*)	**Can this type of problem be detected by test cases?**	**Instructions when generating test cases in order to detect this type of problem**
Data definition, structure, declaration	YES	Cover at 100% the input signals domain, output signals domain, intermediate signals domain Cover at 100% the inputs boundaries domain, outputs boundaries domain, intermediates boundaries domain
Data access and handling	YES	Cover at 100% the input signals domain, output signals domain, intermediate signals domain
Control flow and sequencing	YES	Cover at 100% the code conditions and decisions Cover at 100% the FSM transitions and conditions
Processing	YES	Cover at 100% the input signals domain, output signals domain, intermediate signals domain Cover at 100% the code procedures Cover at 100% the DT conditions Cover at 100% the FSM states
Coding and typographical	YES	Cover at 100% the code statements
Standards violation	NO, code review or other V&V techniques	---
Documentation	NO, code review or other V&V techniques	---

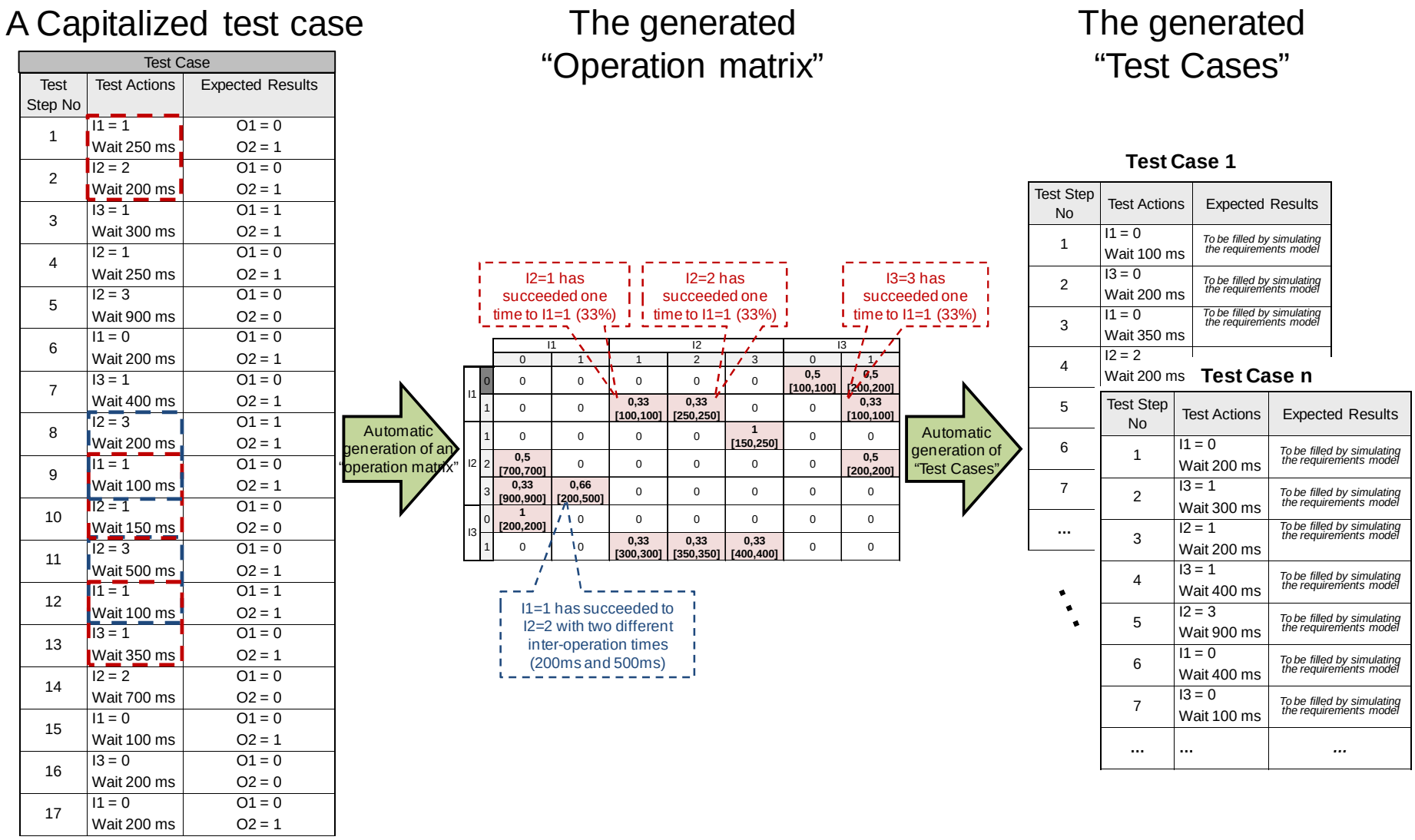
Table 8.2 – Instructions to generate test cases able to detect one specific type of software implementation problems

VI. Reuse of existing test cases from previous projects

Test cases on previous projects are versioned by software functionality and stored in a database. But unfortunately, these *test cases* are not always reused from one project to another. Two main reasons are identified (Cf. *Chapter 2 – Section 8*): 1) the use of different formats when designing manually *test cases*. Sometimes, test engineers write the *test cases* immediately in a computer language (*C language*, *test script*) understandable by the *test execution platform*. Others use the *test case* format presented in *Definition 2.11*. 2) the lack of an automated process to reuse these *test cases*. However, one initiative was launched two years ago and had the purpose to create manually "*Standard Test Cases*" for software validation (Cf. *Chapter 2 – Section 8*). This is a conventional *RETEX* (*RETurn of EXperience*) strategy and the main difficulty of such an approach is to keep these *test cases* updated. Two years after, it is not the case.

Through our research project, we adopt the *test case* format presented in *Definition 2.11* as the standard format to represent a *test case*. Our proposal to reuse existing *test cases* on previous projects is based on this assumption. When testing a functionality, test engineers could select from previous projects all the *test cases* related to the functionality under test. A glossary of the *input signals* names on the previous and current projects is necessary. Then, for each *test case* (independently from the length of the *test case*), an "*operation matrix*" is generated automatically. This "*operation matrix*" has high probability on the successive *operations* regularly done in the *test case*. It also contains the set of *inter-operation time* used between each couple of *operations*. Consequently, when generating *test cases* from these "*operation matrices*", we reduce the *operation space* by focusing on the test scenarios based on our returns of experience.

Let us consider a practical example of a functionality with 3 *input signals* (*I1*, Domain = {0, 1}; *I2*, Domain = {1, 2, 3}; *I3*, Domain = {0, 1}) and 2 *output signals* (*O1*, Domain = {0, 1}; *O2*, Domain = {0, 1}). This functionality was already developed on a previous project in 2005 and one *test case* has been designed. Therefore, when testing the same functionality on a project in 2007 (2 years after), a test engineer selects from the database the *test case* already designed in the past (2005) for this functionality. Each *test case* can be translated automatically into an "*operation matrix*" (Cf. *Figure 8.10*). The *test cases* generated from this "*operation matrix*" (Cf. *Figure 8.10*) focus on specific test scenarios that test engineers have judged critical to perform in the past.



Test Case		
Test Step No	Test Actions	Expected Results
1	I1 = 1 Wait 250 ms	O1 = 0 O2 = 1
2	I2 = 2 Wait 200 ms	O1 = 0 O2 = 1
3	I3 = 1 Wait 300 ms	O1 = 1 O2 = 1
4	I2 = 1 Wait 250 ms	O1 = 0 O2 = 1
5	I2 = 3 Wait 900 ms	O1 = 0 O2 = 0
6	I1 = 0 Wait 200 ms	O1 = 0 O2 = 1
7	I3 = 1 Wait 400 ms	O1 = 0 O2 = 1
8	I2 = 3 Wait 200 ms	O1 = 1 O2 = 1
9	I1 = 1 Wait 100 ms	O1 = 0 O2 = 1
10	I2 = 1 Wait 150 ms	O1 = 0 O2 = 0
11	I2 = 3 Wait 500 ms	O1 = 0 O2 = 1
12	I1 = 1 Wait 100 ms	O1 = 1 O2 = 1
13	I3 = 1 Wait 350 ms	O1 = 0 O2 = 1
14	I2 = 2 Wait 700 ms	O1 = 0 O2 = 0
15	I1 = 0 Wait 100 ms	O1 = 0 O2 = 1
16	I3 = 0 Wait 200 ms	O1 = 0 O2 = 0
17	I1 = 0 Wait 200 ms	O1 = 0 O2 = 1

		I1		I2			I3	
		0	1	1	2	3	0	1
I1	0	0	0	0	0	0	0,5 [100,100]	0,5 [200,200]
	1	0	0	0,33 [100,100]	0,33 [250,250]	0	0	0,33 [100,100]
I2	1	0	0	0	0	1 [150,250]	0	0
	2	0,5 [700,700]	0	0	0	0	0	0,5 [200,200]
	3	0,33 [900,900]	0,66 [200,500]	0	0	0	0	0
I3	0	1 [200,200]	0	0	0	0	0	0
	1	0	0	0,33 [300,300]	0,33 [350,350]	0,33 [400,400]	0	0

Test Case 1

Test Step No	Test Actions	Expected Results
1	I1 = 0 Wait 100 ms	*To be filled by simulating the requirements model*
2	I3 = 0 Wait 200 ms	*To be filled by simulating the requirements model*
3	I1 = 0 Wait 350 ms	*To be filled by simulating the requirements model*
4	I2 = 2 Wait 200 ms	
5		
6		
7		
...		

Test Case n

Test Step No	Test Actions	Expected Results
1	I1 = 0 Wait 200 ms	*To be filled by simulating the requirements model*
2	I3 = 1 Wait 300 ms	*To be filled by simulating the requirements model*
3	I2 = 1 Wait 200 ms	*To be filled by simulating the requirements model*
4	I3 = 1 Wait 400 ms	*To be filled by simulating the requirements model*
5	I2 = 3 Wait 900 ms	*To be filled by simulating the requirements model*
6	I1 = 0 Wait 400 ms	*To be filled by simulating the requirements model*
7	I3 = 0 Wait 100 ms	*To be filled by simulating the requirements model*
...	...	...

Figure 8.10 – Process of reusing test cases capitalized on previous projects

VII. Conclusion

Only exhaustive testing can show that a software product is free from bugs. However, exhaustive testing of a software product is not practical because variable input values and variable sequencing of inputs result in too many possible combinations to test. So it would be useful to concentrate the test on the areas associated with the greatest risks and priorities. However, we have to identify and analyze these risks and priorities.

In this chapter, we developed three strategies able to reduce the *operation space* of a software product. Our main purpose was to focus on test scenarios with a high probability to detect software bugs. *Firstly*, we specified four types of constraints that test engineers can set on the *input signals* of the functionality under test in order to favor or avoid specific successions of *operations*. *Secondly*, we developed a new "*Problem description*" format to capitalize the initial conditions and the successive *operations* that lead to a bug. Based on this new format, tester engineers can generate automatically one or more *test cases* from each capitalized bug. We also developed a detailed software problem typology that helps test engineers to identify recurrent types of problems and better address the generation of *test cases*. *Finally*, we set up an automatic process to reuse *test cases* from one project to another.

In the latest four chapters (*Chapter 5*, *6*, *7* and *8*), we specified our approach to improve the global performance of the Johnson Controls V&V activities. In the following two chapters (*Chapter 9* and *10*), we respectively implement our approach in a computer platform (prototype) and validate it through two industrial case studies.

PART IV – IMPLEMENTATION, VALIDATION AND IMPACTS OF THE PROPOSED APPROACH

CHAPTER 9. PROTOTYPE IMPLEMENTATION

I. Introduction

After specifying our new approach to generate efficient *test cases* automatically (Cf. *Chapter 5, 6, 7 and 8*), we focus in this chapter on the practical use of this approach within an industrial context. We develop a prototype implementing our models, concepts and theories. A "*functional*" view of our approach is illustrated in *Section 2*. A "*process-role-tool*" view of our approach is proposed in *Section 3*. The processes are mainly defined in *Chapter 5, 6, 7 and 8*. Some specific skills which are mandatory when using our approach are detailed. We also describe the three computer tools that we developed in order to automate the generation of *test cases*. The main one is the *Test Case Generation tool* which is a *PC* application. The main functionalities of this tool are developed in details in *Section 4*.

II. A "functional" view of our approach

In *Chapter 2 – Section 6*, we describe how Johnson Controls test engineers currently design *test* cases for software products. They proceed to a manual design of *test cases*. The performance of the design is mainly based on their experience. In *Chapter*s *5*, *6*, *7* and *8*, we develop our new approach to design *test cases* automatically. A *functional* view of our approach is presented in *Figure 9.1*. It is based on eight activities. These activities are:

1. Design a simulation model of the software functional requirements of the functionality under test (*Cf. Chapter 5*).
2. Verify and validate the requirements model (*Cf. Chapter 6*).
3. Define some behavioral characteristics of a car driver when using the functionality under test (*Cf. Chapter 8*).
4. Perform a statistical analysis on bugs detected in the past on the functionality under test (*Cf. Chapter 8*).
5. Perform a statistical analysis on *test cases* developed (in the past) on the functionality under test (*Cf. Chapter 8*).
6. Highlight the relevant, critical and mandatory *operations* and succession of *operations* to be chosen from the *operation space* of the functionality under test (*Cf. Chapter 8*).
7. Automate the design of *test cases* from the requirements model (*Cf. Chapter 7*).
8. Monitor the design of *test cases* by quality objectives and time and cost constraints (*Cf. Chapter 7*).

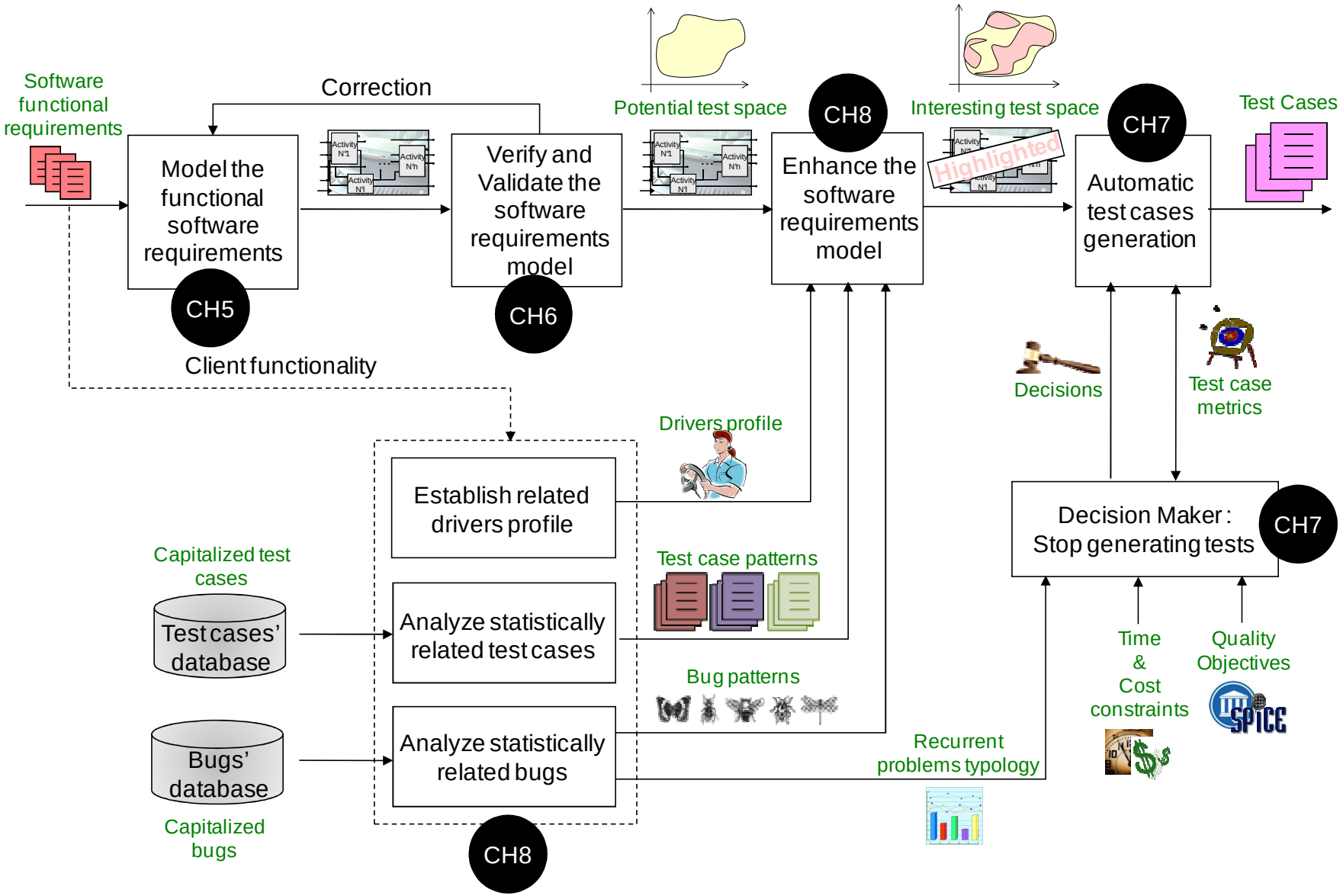


Figure 9.1 – A "functional" view of our approach

III. A “process-role-tool” view of our approach

Our approach presents a much different workflow for designing *test cases* than the present one. The new workflow is presented in *Figure 9.2*. It is composed from seven *processes* which are *manual*, *semi-automatic* or *automatic* and managed by *different individuals* (experts, modelers and test engineers).

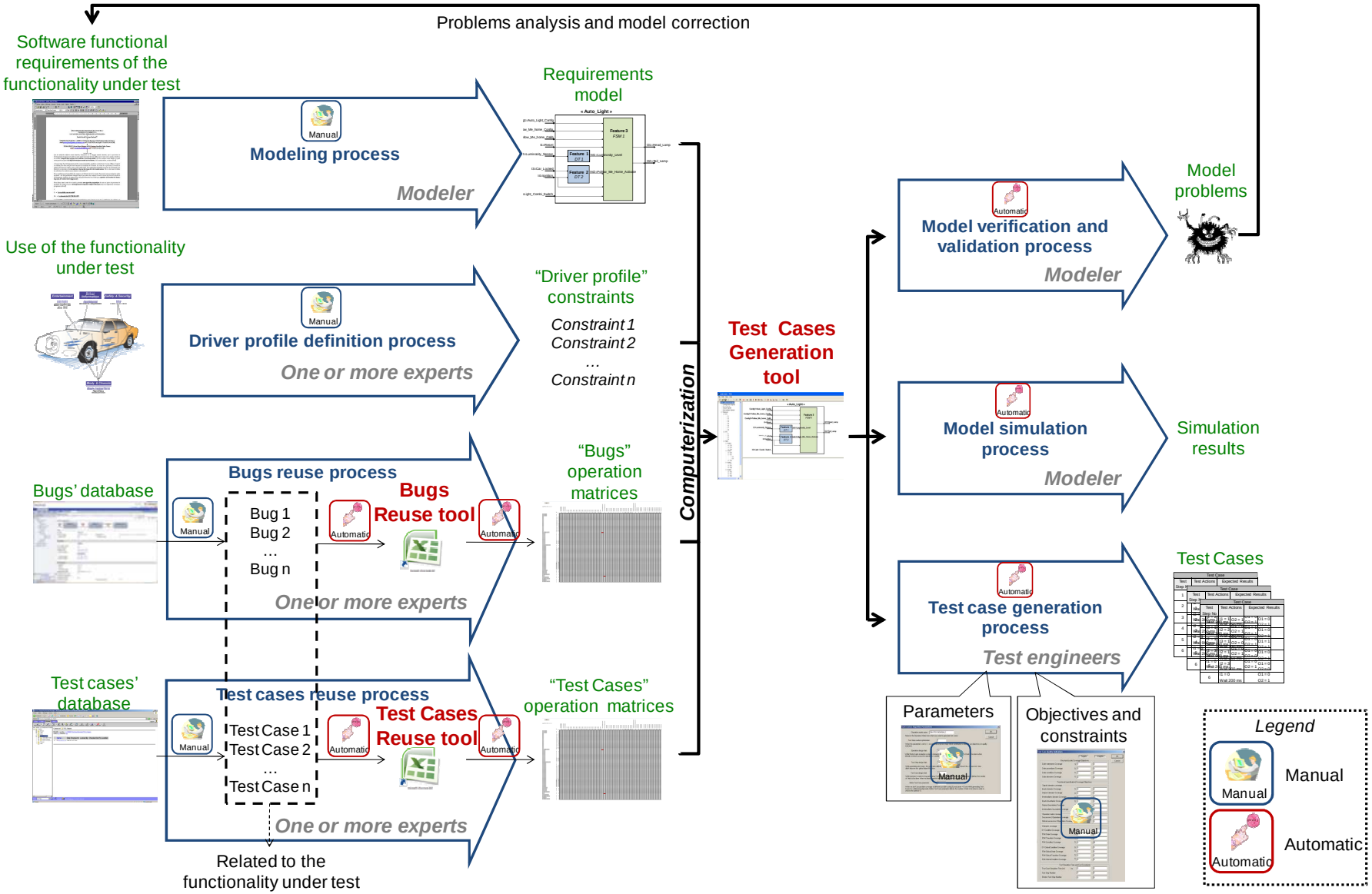


Figure 9.2 – A "process-role-tool" view of our approach

A. Processes

Our approach is composed of seven processes:

1. *Modeling process* (manual): models the software functional requirements using our formal specification language.
2. *Driver profile definition process* (manual): defines the driver behavior when using the functionality under test.
3. *Bugs reuse process* (semi-automatic): establishes a framework in order to reuse the bugs capitalized in the *problems' database* and related to the functionality under test.
4. *Test cases reuse process* (semi-automatic): establishes a framework in order to reuse the *test cases* developed on previous projects and related to the functionality under test.
5. *Model verification and validation process* (automatic): verifies and validates the requirements model consistency and compliance with the carmaker requirements.
6. *Model simulation process* (automatic): simulates the requirements model
7. *Test Case generation process* (automatic): monitors the generation of *test cases* by quality objectives and time and cost constraints

These seven processes have been developed in details in *Chapters 5*, *6*, *7* and *8*.

B. Roles

"Roles" can be allocated to one or more people in a software project provided one has the time and the required skills. Three types of roles have been identified:

- *Modeler:* the main tasks of a modeler are to analyze the software functional requirements, design the requirements model, verify and validate the model and finally simulate it. A modeler has to be familiar with the behavior of the car's software functionalities and a master of the formal specification language. She/he also needs good communication skills. Indeed, she/he has to interact with the carmakers in order to eliminate ambiguities and inconsistencies from the requirements. Finally, analysis skills are necessary for the *Verification and Validation* (*V&V*) of the requirements model.
- *Expert*: the main tasks of an expert or a group of experts are to define a driver profile for the functionality under test, to identify related bugs and *test cases* capitalized on previous projects and to extract from these bugs and *test cases* relevant *lessons learned*. An expert has to be a master of automotive electronics. She/He needs to have a global view of all the projects and software practices within the company.
- *Test engineer:* the main tasks of a test engineer are to parameterize the test generation algorithm and to set the quality objectives and the time and cost constraints. The generation of *test cases* is automatic. However, test engineer has to execute the generated *test cases* on the software product under test and analyze the results. A test engineer has to be familiar with the behavior of the car's software functionalities and the formal specification language. Knowledge about optimization is necessary to better parameterize the optimization algorithm. In addition, she/he has to be a master in *requirements* and *code coverage* in order to set relevant *coverage* objectives. Finally, analysis skills are mandatory for the analysis of the test results and reports.

C. Tools

In this section, we develop the three computer tools supporting the *semi-automatic* and *automatic* processes of our approach: *Bugs Reuse tool*, *Test Cases Reuse tool* and *Test Case Generation tool*.

1. Bugs Reuse tool

In order to reuse the bugs capitalized in the *problems' database*, experts have to identify the relevant bugs related to the functionality under test. In *Chapter 8 – Section 5.B*, we define a new format to fill in the "*Problem description*" attribute of a bug. Based on this format, we develop an *Excel Macro* able to analyze the "*Problem description*" of a bug and to generate the corresponding "*operation matrix*" (Cf. *Figure 9.3*). This matrix is used to generate *test cases* able to detect a similar bug on future development. When analyzing a bug and generating an "*operation matrix*", the *Excel Macro* uses a glossary of *input signals* names on the previous and current projects. The *Macro* has been developed in *Visual Basic* language.

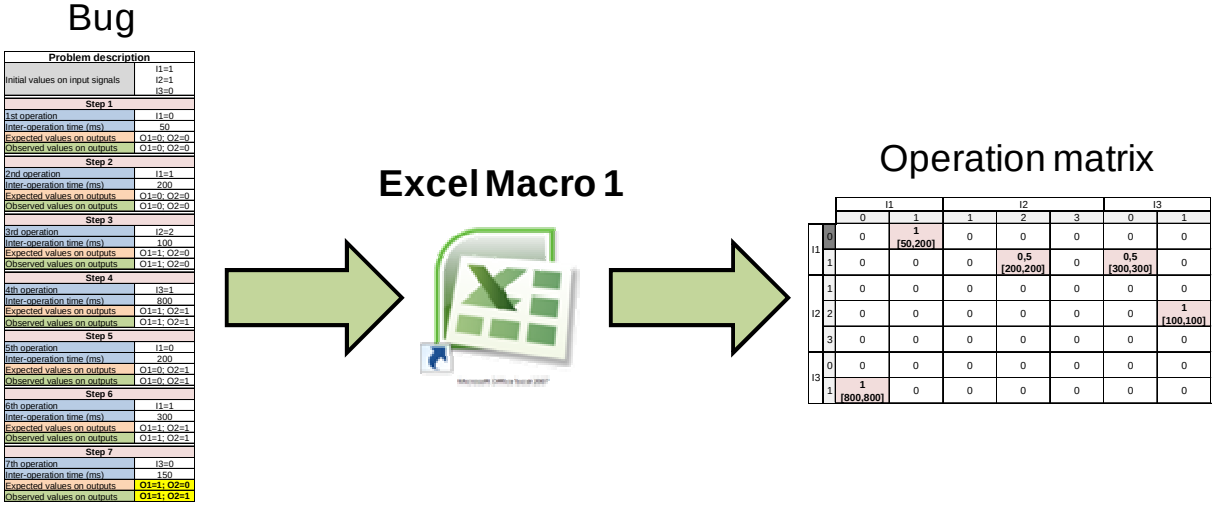
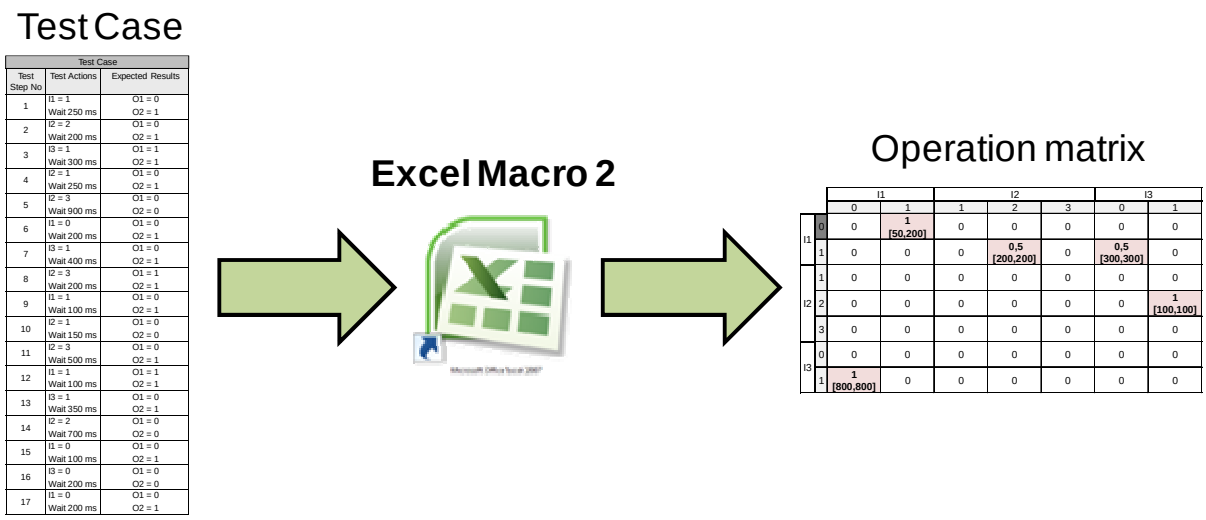


Figure 9.3 – Bugs Reuse tool

A detailed description of the process of analyzing the bug and generating the "*operation matrix*" is given in *Chapter 8 – Section 5.A.*

2. Test Cases Reuse tool

In order to reuse the *test cases* from one project (in the past) to another (in the present or future), experts have to identify the *test cases* related to the functionality under test. In *Chapter 8 – Section 6*, we adopt the *test case* format defined in *Definition 2.11*. Based on this format, we develop an *Excel Macro* able to analyze a *test case* and generate the corresponding "*operation matrix*" (Cf. *Figure 9.4*). In this matrix, the *operation space* is reduced by focusing on the test scenarios based on the returns of experience. *Test cases* generated from this "*operation matrix*" contain similar successions of *operations* as in the one designed manually or generated automatically in the past. A glossary of the *input signals* names on the previous and current projects is also necessary. The *Macro* has been developed in *Visual Basic* language.



Test Case		
Test Step No	Test Actions	Expected Results
1	I1 = 1 Wait 250 ms	O1 = 0 O2 = 1
2	I2 = 2 Wait 200 ms	O1 = 0 O2 = 1
3	I3 = 1 Wait 300 ms	O1 = 1 O2 = 1
4	I2 = 1 Wait 250 ms	O1 = 0 O2 = 1
5	I2 = 3 Wait 900 ms	O1 = 0 O2 = 0
6	I1 = 0 Wait 200 ms	O1 = 0 O2 = 1
7	I3 = 1 Wait 400 ms	O1 = 0 O2 = 1
8	I2 = 3 Wait 200 ms	O1 = 1 O2 = 1
9	I1 = 1 Wait 100 ms	O1 = 0 O2 = 1
10	I2 = 1 Wait 150 ms	O1 = 0 O2 = 0
11	I2 = 3 Wait 500 ms	O1 = 0 O2 = 1
12	I1 = 1 Wait 100 ms	O1 = 1 O2 = 1
13	I3 = 1 Wait 350 ms	O1 = 0 O2 = 1
14	I2 = 2 Wait 700 ms	O1 = 0 O2 = 0
15	I1 = 0 Wait 100 ms	O1 = 0 O2 = 1
16	I3 = 0 Wait 200 ms	O1 = 0 O2 = 0
17	I1 = 0 Wait 200 ms	O1 = 0 O2 = 1

		I1		I2			I3	
		0	1	1	2	3	0	1
I1	0	0	1 [50,200]	0	0	0	0	0
	1	0	0	0	0,5 [200,200]	0	0,5 [300,300]	0
I2	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	1 [100,100]
	3	0	0	0	0	0	0	0
I3	0	0	0	0	0	0	0	0
	1	1 [800,800]	0	0	0	0	0	0

Figure 9.4 – Test Cases Reuse tool

A detailed definition of the process of analyzing the *test case* and generating the "*operation matrix*" is given in *Chapter 8 – Section 6.*

3. Test Case Generation tool

Through our research project, we were asked by the automotive electronic supplier Johnson Controls to automate the design of *test cases* for software products (Cf. *Chapter 1 – Section 6*). Therefore, we develop a computer tool, the *Test Case Generation tool*, able to computerize our requirements models and therefore generate *test cases* automatically.

a. Computer implementation

We use the *Visual C++ tool*[37] and the *C++ language* to develop the *Test Case Generation tool*. First, we perform a global design of the tool using the *UML*[38] language, then we generate automatically the *Visual C++ code-skeleton* of the developed *UML* model and finally, we develop the body of the *code-skeleton.*

We use the *UML* editor of *Rational Rose* (*Rational Rose Modeler tool*[39]) in order to perform a global design of the *Test Case Generation tool*. A simplified *class diagram* with all the classes of the tool is shown in *Figure 9.5*. Two groups of classes are identified. The first one is related to the design of the requirements model. The second one deals with the generation of *test cases*. The detailed diagram with the *attributes* and *methods* of all the classes and the types of relations between classes is not presented here for confidential reasons.

[37] http://msdn.microsoft.com/fr-fr/visualc/default.aspx (Consulter on November 2008)
[38] http://www.uml.org/ (Consulter on November 2008)
[39] http://www-01.ibm.com/software/awdtools/developer/datamodeler/ (Consulter on November 2008)

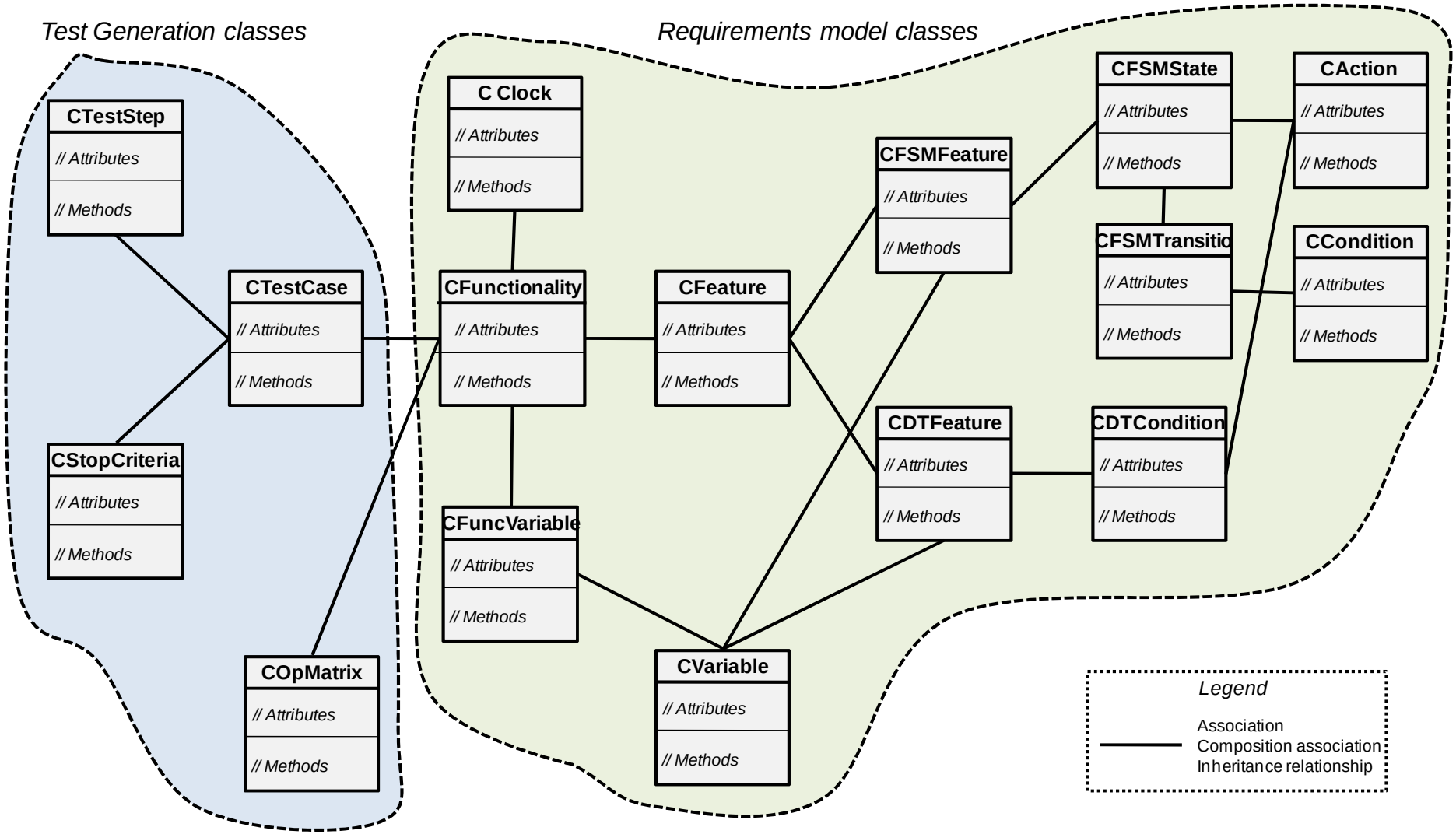


Figure 9.5 – Simplified class diagram of the Test Case Generation tool

We use the *Rational Rose Professional C++ tool*[40] in order to generate the *Visual C++ code-skeleton* from the developed *class diagram*. When generating *code-skeleton*, the tool automatically creates the *.h* and *.cpp* files. It generates the *classes* and adds the *attributes* to them. It also creates the *methods* with empty bodies. Afterwards, we must go in and add the body of the *methods*. A screenshot of the *code-skeleton* generated by *Rational Rose* is presented in *Figure 9.6*.

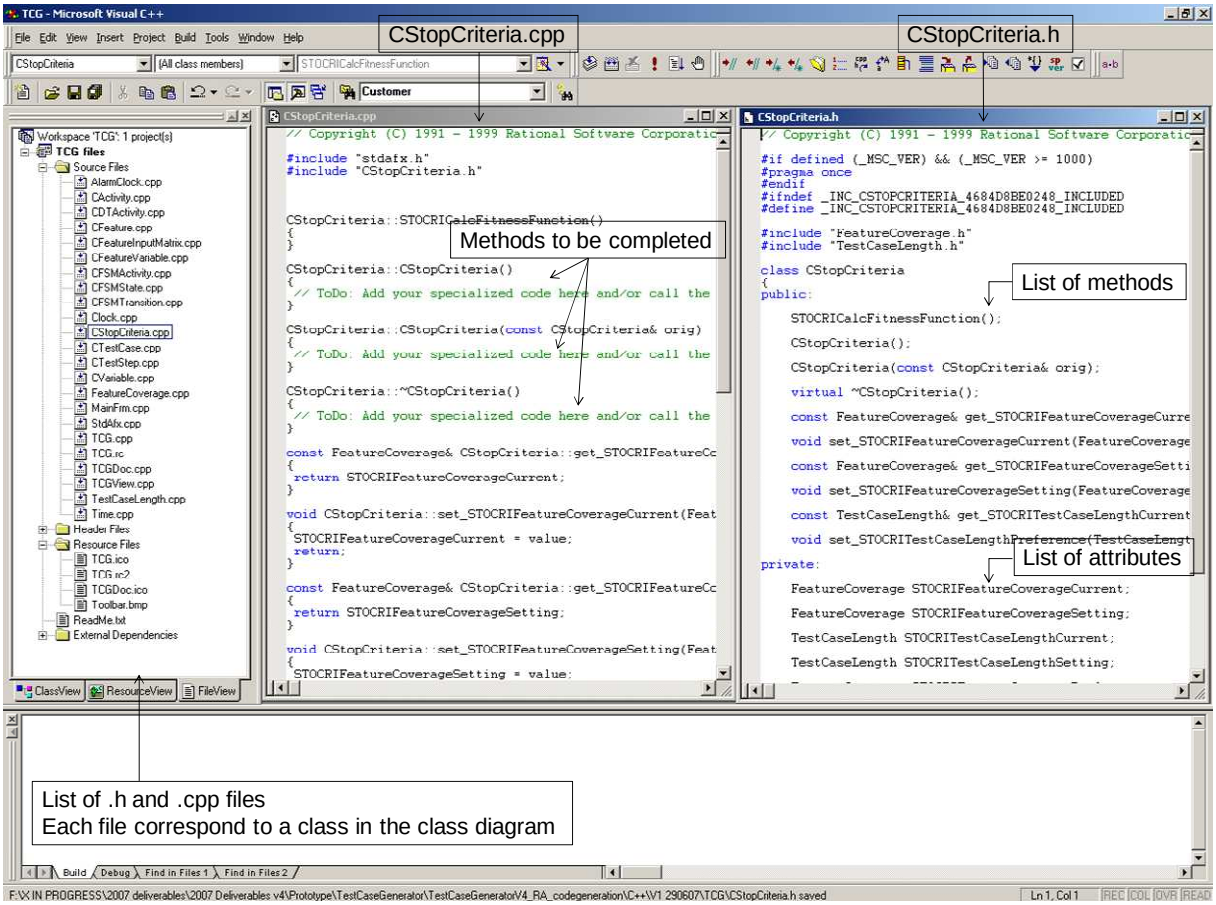


Figure 9.6 – A screenshot of the C++ code-skeleton generated by Rational Rose

Once the software architecture of the *Test Case Generation tool* is generated, we start developing in *C++ language* the body of each *method*. We have developed about 12500 *Lines Of Codes* (excluding comments and blank lines). In fact, we have implemented, using a computer language, all the models developed in *Chapter 5*, *6*, *7* and *8*.

b. List of main functionalities

The main functionalities of the *Test Case Generation tool* are:

- Computerize and verify a requirements model
- Generate *Nominal "operation matrices"* automatically
- Import *"operation matrices"*
- Set constraints on the *input signals* of a requirements model and generate automatically the corresponding *"operation matrix"*
- Simulate a requirements model

[40] http://www-01.ibm.com/software/awdtools/developer/rose/visualstudio/support/ (Consulted on November 2009)

- Generate *test cases* automatically

Each of these functionalities is developed in the next section.

IV. Main functionalities of the Test Case Generation tool

In this section, the main functionalities of the *Test Case Generation tool* are detailed.

A. Computerize and verify a requirements model

After sketching "on paper" the requirements model (Cf. *Chapter 5 – Section 5*), one can computerize this model using the *Test Case Generation tool*. One can also verify the correctness of the computerized model by checking automatically the set of integrity rules presented in *Table 6.1*. A screenshot of the tool after computerizing the requirements model of the *Chapter 5 – Section 5* example is illustrated in *Figure 9.7*.

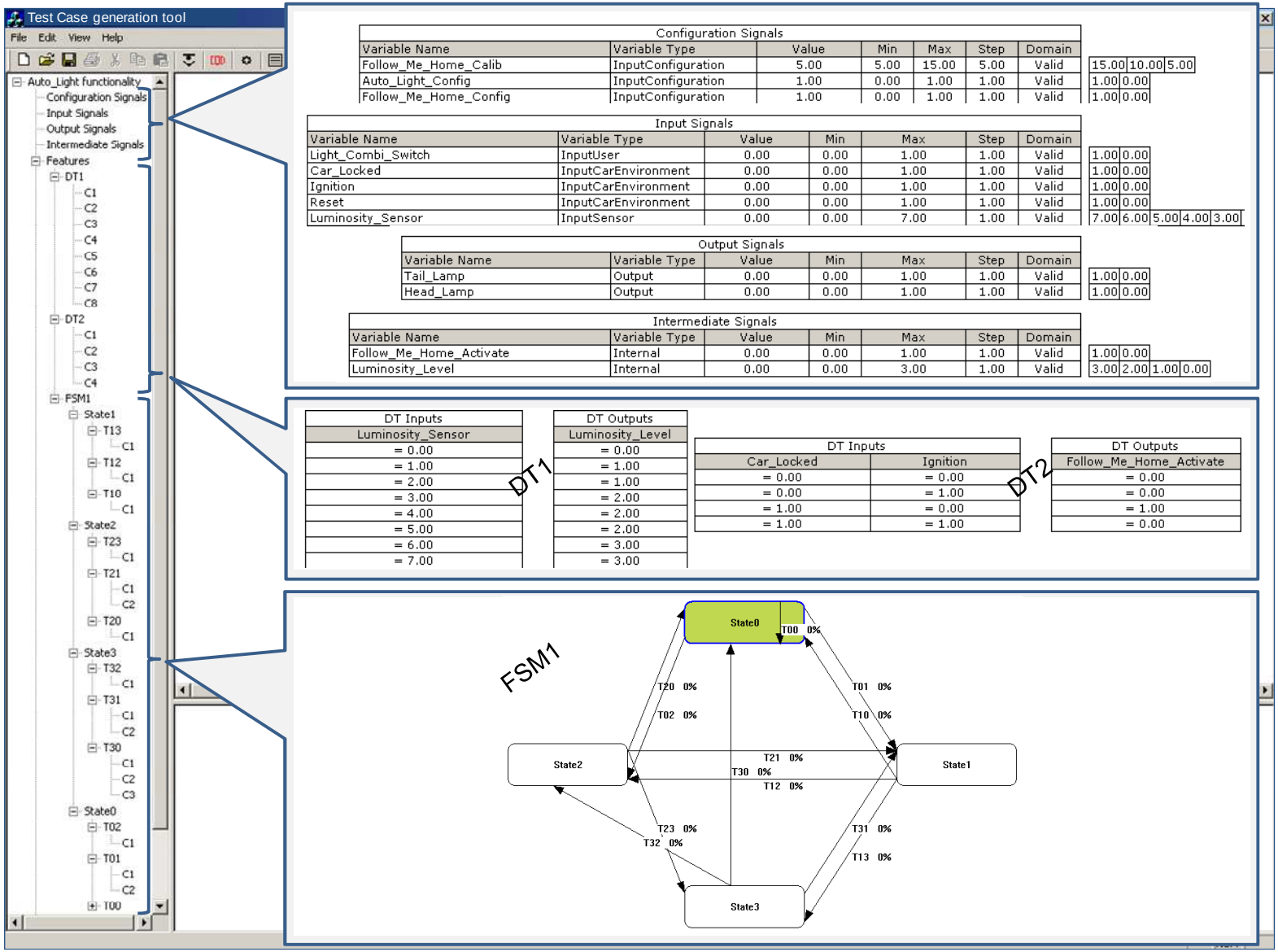
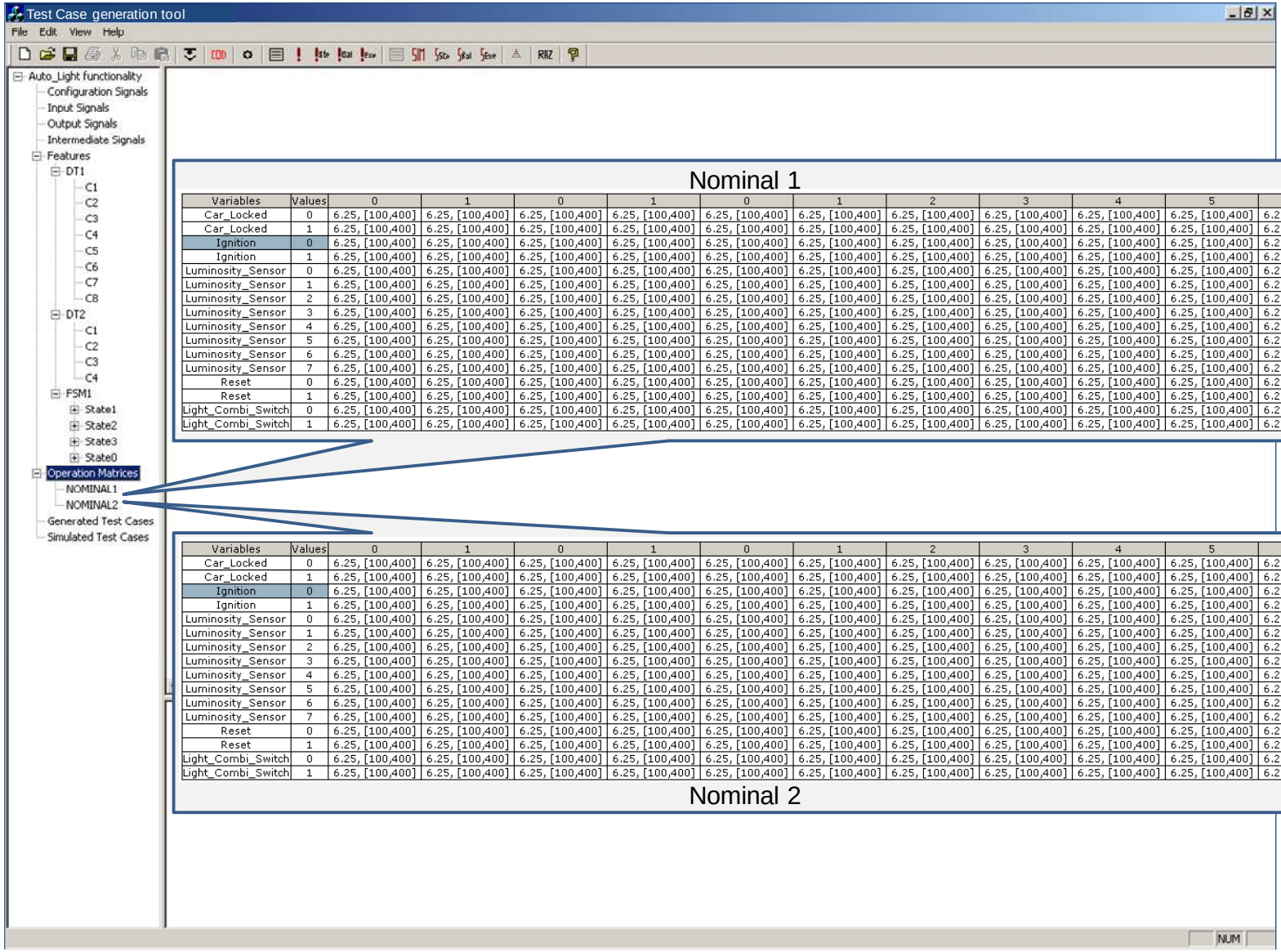


Figure 9.7 – A screenshot of the tool after computerizing the requirements model of the *Chapter 5 – Section 5* example

B. Generate Nominal “operation matrices” automatically

After computerizing the requirements model, one can generate automatically the two *Nominal “operation matrices”* (*Nominal 1* and *Nominal 2*, Cf. *Chapter 7 – Section 2*). One can also customize these matrices by modifying some succession probabilities or *inter-operation time* interval *(Cf. Chapter 7 – Section 2).*

A screenshot of the tool after generating the *Nominal “operation matrices”* of the *Chapter 5 – Section 5* example is illustrated in *Figure 9.8*.



Variables	Values	0	1	0	1	0	1	2	3	4	5	6
Car_Locked	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Car_Locked	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Ignition	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Ignition	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	2	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	3	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	4	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	5	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	6	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	7	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Reset	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Reset	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Light_Combi_Switch	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Light_Combi_Switch	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]

Variables	Values	0	1	0	1	0	1	2	3	4	5	6
Car_Locked	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Car_Locked	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Ignition	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Ignition	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	2	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	3	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	4	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	5	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	6	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Luminosity_Sensor	7	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Reset	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Reset	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Light_Combi_Switch	0	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]
Light_Combi_Switch	1	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]	6.25, [100,400]

Figure 9.8 – A screenshot of the tool after generating the Nominal "operation matrices" of the *Ch. 5 – Section 5* example

C. Import "operation matrices"

One can import *"operation matrices"*. These matrices can be the results of the *bugs* and *test cases reuse processes*. They can also be designed manually by an inner engineer. A screenshot of the tool after importing *"operation matrices"* is illustrated in *Figure 9.9*.

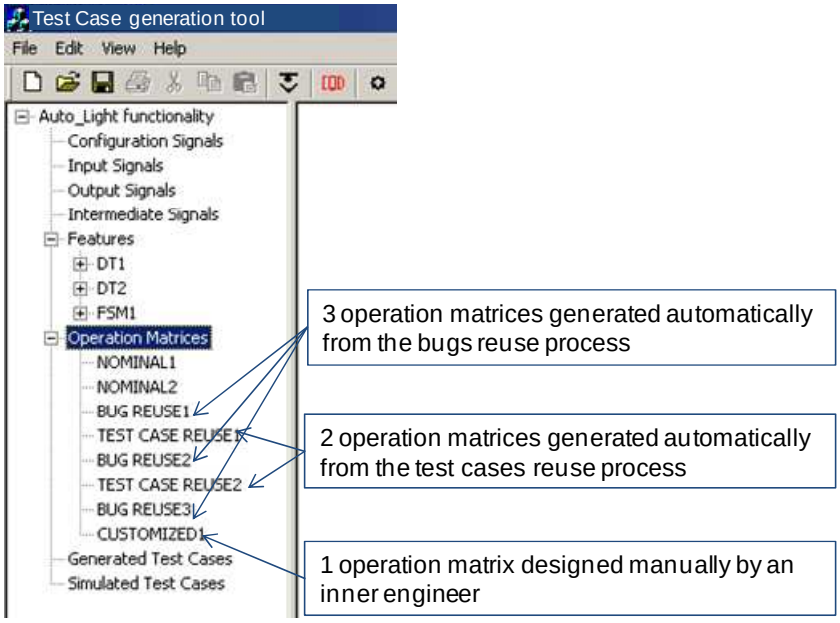


Figure 9.9 – A screenshot of the tool after importing "operation matrices"

D. Set constraints on the input signals of a requirements model and generate automatically the corresponding "operation matrix"

We develop a computer language that experts can use in order to specify their constraints on the *input signals*. Four types of constraints have been proposed in *Chapter 8 – Section 4* (*Logical*, *Conditional*, *Succession* and *Timing constraints*). The *Test Case Generation tool* analyzes these constraints and generates automatically the corresponding *Driver Profile "operation matrix"*. The generation of *test cases* from this *"operation matrix"* fulfills the predefined constraints on *input signals*. An excerpt on how experts can set constraints on the *input signals* of a requirements model is shown in *Figure 9.10*.

Manual Design of constraints

Constraints description	Constraints definition using the computer language that we developed
The *"Ignition"* signal is Cyclic	*// Constraint definition* Constraint1(Cyclic); *// Set the constraint to an input signal* Ignition(Constraint1);
The *"Ignition"* signal can be different from 2 only if *"Light_Combi_Switch"* is equal to 0	*// Constraint definition* Constraint2(NEQUAL, 2, "Light_Combi_Switch", EQUAL, 0); *// Set the constraint to an input signal* Ignition(Constraint2);
Once *"Car_Locked"* is set to 0, the *"Light_Combi_Switch"* signal is set to 1 with a probability of 0.5	*// Constraint definition* Constraint3(1, "Car_Locked", 0, 50); *// Set the constraint to an input signal* Light_Combi_Switch(Constraint3);
Once *"Car_Locked"* is set to 0, the *"Light_Combi_Switch"* signal is set to 0 with a time interval of [13000;13000]	*// Constraint definition* Constraint4(0, "Car_Locked", 0, 13000, 13000); *// Set the constraint to an input signal* Light_Combi_Switch(Constraint4);

Export the constraints to the Test Case generation tool

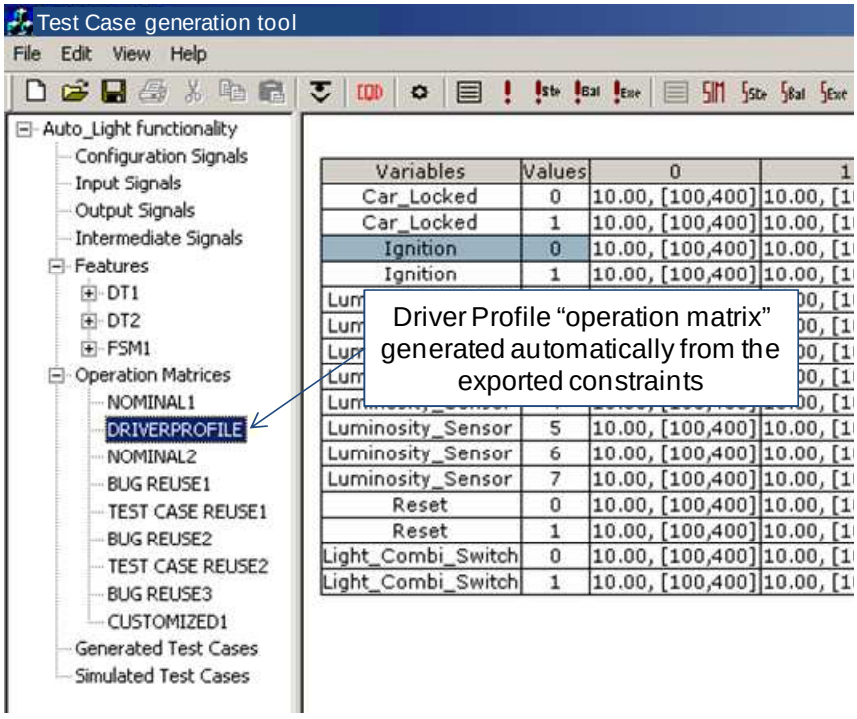
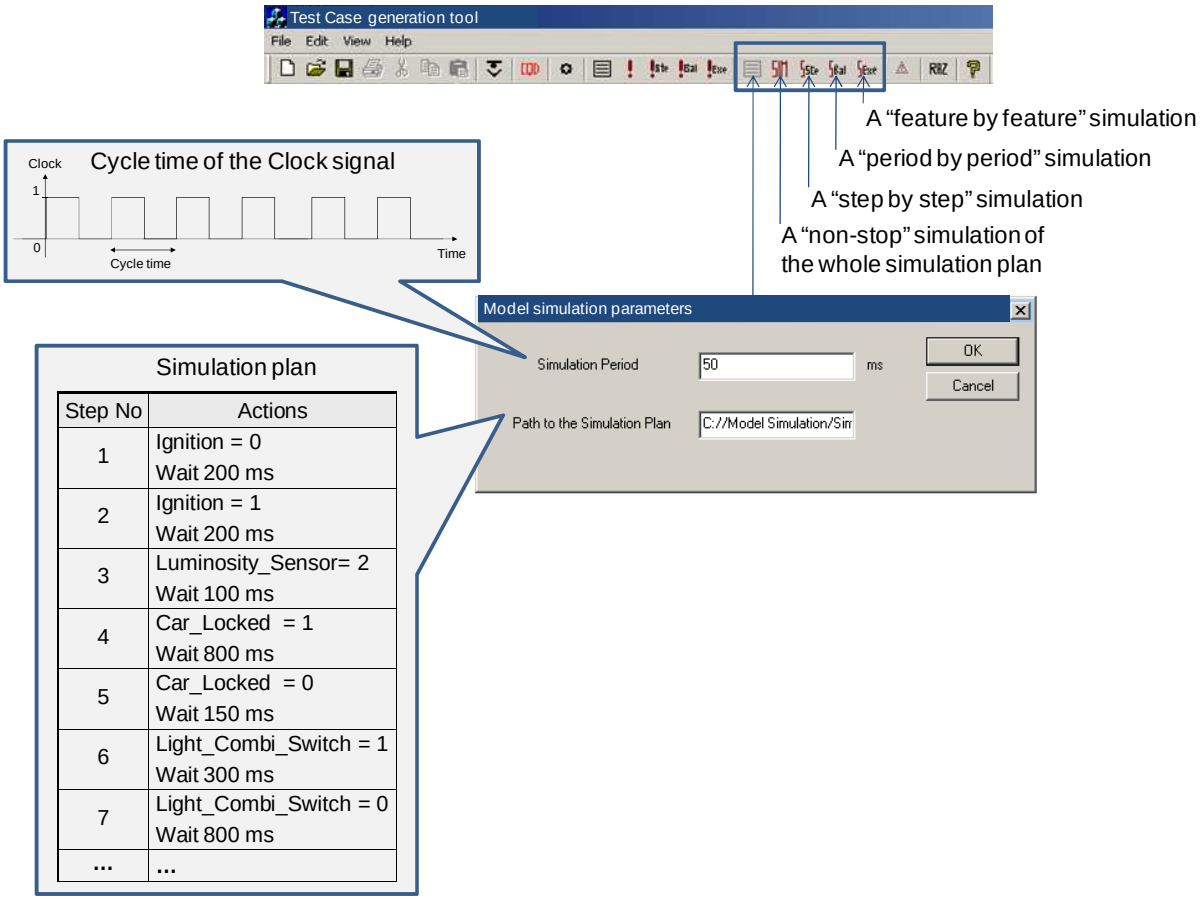


Figure 9.10 – An excerpt on how experts can set constraints on the input signals of a requirements model

E. Simulate a requirements model

Once the requirements model is computerized, one can simulate it. Modeler has to define a simulation period (the *cycle time* of the *Clock signal*, Cf. *Chapter 5 – Section 3*). Modeler has also to specify the path where the *simulation plan* is stored. In fact, a *simulation plan* consists of a finite number of steps. In each step, at most one *operation* on the *input signals* is performed and an *inter-operation time* is defined. The result of a simulation is the behavior of

the *output signals* of the requirements model after each step of the *simulation plan*. The output data of a model simulation are stored in an *Excel* file. In *Figure 9.11*, we illustrate the simulation parameters and the four modes of simulating a requirements model. The *"non-stop"* mode aims to simulate the whole *simulation plan* nonstop. The *"step by step"* mode consists of simulating the *simulation plan* in a step by step manner. Indeed, after each step's simulation, the simulation process is stopped. The *"period by period"* mode stops the simulation at each period of the *Clock signal*. Finally, the *"feature by feature"* mode stops the simulation after each *feature* simulation.



Step No	Actions
1	Ignition = 0 Wait 200 ms
2	Ignition = 1 Wait 200 ms
3	Luminosity_Sensor= 2 Wait 100 ms
4	Car_Locked = 1 Wait 800 ms
5	Car_Locked = 0 Wait 150 ms
6	Light_Combi_Switch = 1 Wait 300 ms
7	Light_Combi_Switch = 0 Wait 800 ms
...	...

Figure 9.11 – The simulation toolbox of the Test Case Generation tool

F. Generate test cases automatically

The main functionality of the *Test Case Generation tool* is to generate *test cases* automatically. In *Chapter 7 – Section 4*, we developed a set of test generation objectives and constraints. A panel interface to allow the test engineer to set precise targets on these objectives and constraints is presented in *Figure 7.13*. In *Chapter 7 – Section 6*, we have developed a heuristic algorithm to optimize the generation of *test cases* while fulfilling the predefined objectives and constraints. A list of 8 parameters that a test engineer should set before start generating *test cases* is also introduced.

Through the *Test Case Generation tool*, one can set targets on the test generation objectives and constraints and parameterize the test generation algorithm. The generation of *test cases* is automatic. Each generated test case and its reached objectives and constraints are stored in an *Excel* file. In *Figure 9.12*, we illustrate the panels where a test engineer can calibrate the generation of *test cases*. We also identify the four modes of generating *test cases*. The *"non-stop"* mode aims to generate the set of required *test cases* nonstop. The *"step by step"* mode

consists of generating each *test case* in a *test step* by *test step* manner. Indeed, after each *test step* generation, the test generating process is stopped. When designing a *test step* and after choosing an *operation* and an *inter-operation time*, the *"period by period"* mode stops the model simulation (in order to assess the expected outputs) at each period of the *Clock signal*. Finally, the *"feature by feature"* mode stops the simulation after each *feature* simulation. We are conscious of the number of parameters (*coverage* objectives, constraints, optimization parameters …) required to set our approach. In *Chapter 10 – Section 8*, we propose two strategies to help test engineers parameterizing the generation of *test cases*.

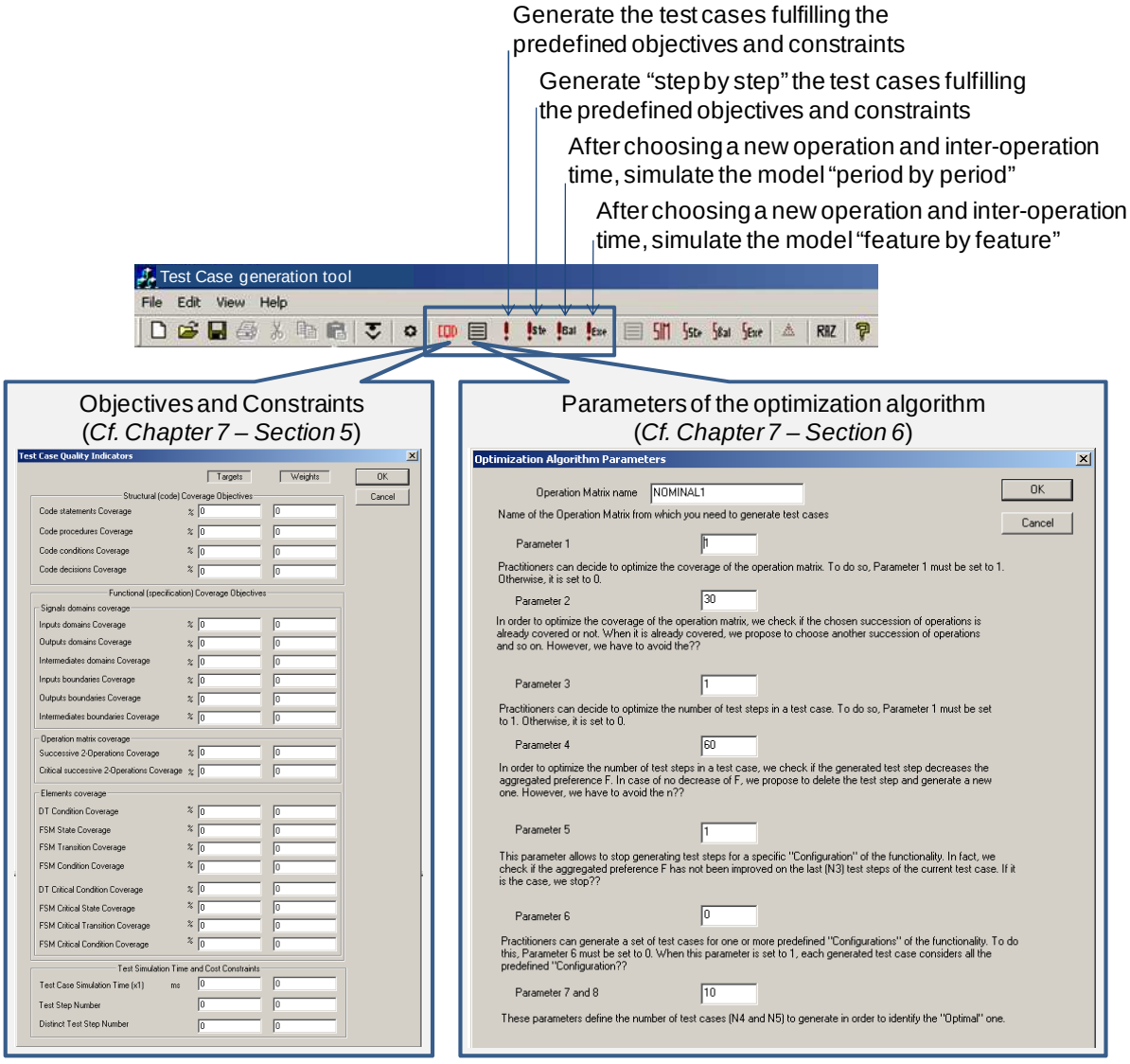


Figure 9.12 – The test generation toolbox of the Test Case Generation tool

Let us consider the example of the *Chapter 5 – Section 5*. After computerizing and verifying the requirements model (Cf. *Figure 9.7*), one decides to generate the *Nominal 1* and *2 "operation matrices"* (Cf. *Figure 9.8*). For a specific "*Configuration*" of the functionality "Auto_Light" under test (*Parameter 6* = 0, *Cf. Chapter 7 – Section 6*), one decides to generate *test cases* that covers at 100% the domain of all the *input*, *output* and *intermediate signals* (*coverage* objectives, Cf. *Chapter 7 – Section 4 and 5*). Nevertheless, the length of these *test cases* must not exceed 50 *test steps* (time and cost constraints, Cf. *Chapter 7 – Section 4 and 5*). The *test cases* must be generated from the *Nominal 2 "operation matrix"*. When generating the *test cases*, one decides to avoid already covered successions of *operations* (*Parameter 1* = 1 and *Parameter 2* = 30, Cf. *Chapter 7 – Section 6*). One also decides to

optimize the number of *test steps* by keeping only the ones which contribute to the objectives fulfillment (*Parameter 3* = 1, *Parameter 4* = 10, Cf. *Chapter 7 – Section 6*). After 10 generated *test steps* with no improvement in the objectives fulfillment, the corresponding *test case* must be ended (*Parameter 5* = 10, *Cf. Chapter 7 – Section 6*). And finally, one decides to generate 5 separate *test cases* in order to choose the "optimal" one (*Parameter 8* = 5, *Cf. Chapter 7 – Section 6*). Since *Parameter 6* is equal to 0, *Parameter 7* has not to be defined *(Cf. Chapter 7 – Section 6.B)*. A screenshot of the *Test Case Generation tool* after generating the *test cases* for the previous exercise is presented in *Figure 9.13*. The generated *test cases* and their reached objectives are stored in an *Excel* file. In case the execution of the *test cases* on the software product under test is automatic, the generated *test cases* can be translated into a computer language understandable by the *test execution platform* (Cf. *Appendix C*). Moreover, while simulating a *simulation plan* on a requirements model or generating *test cases* from a requirements model, one can visualize in real time the covered zones of the model (Cf. *Chapter 7 – Section 4.B*). In fact, the *Test Case Generation tool* highlights the covered zone of the model (*Signals domain*, *Conditions* of *Decision Tables*, *States*, *Transitions* and *Conditions* of *Finite State Machines* and *Operation matrices*). After generating a set of *test cases* from the computerized requirements model presented in *Figure 9.7* the covered zones of this model are illustrated in *Figure 9.14*.

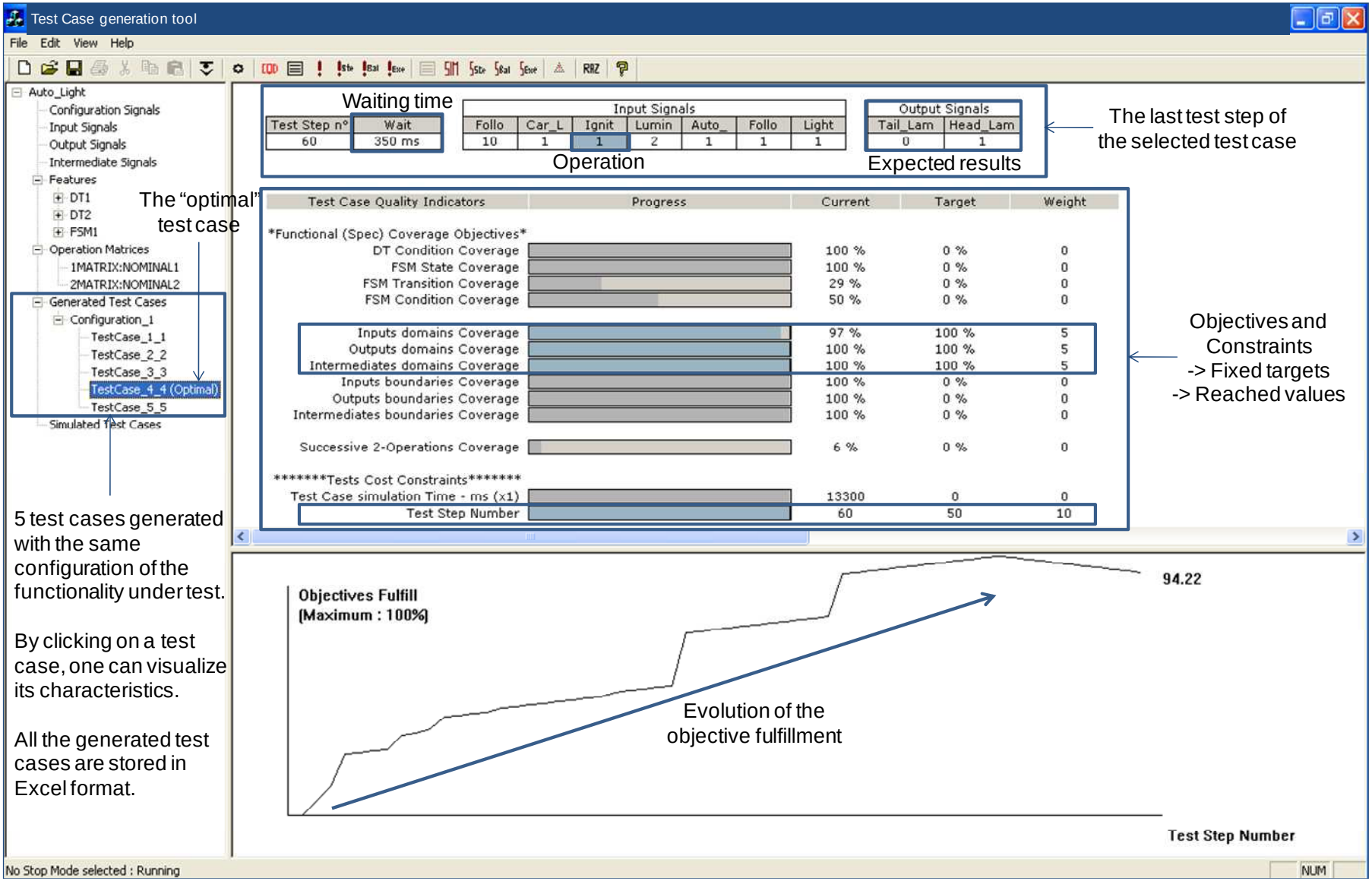


Figure 9.13 – A screenshot of the tool after generating test cases for the *Chapter 5 – Section 5* example

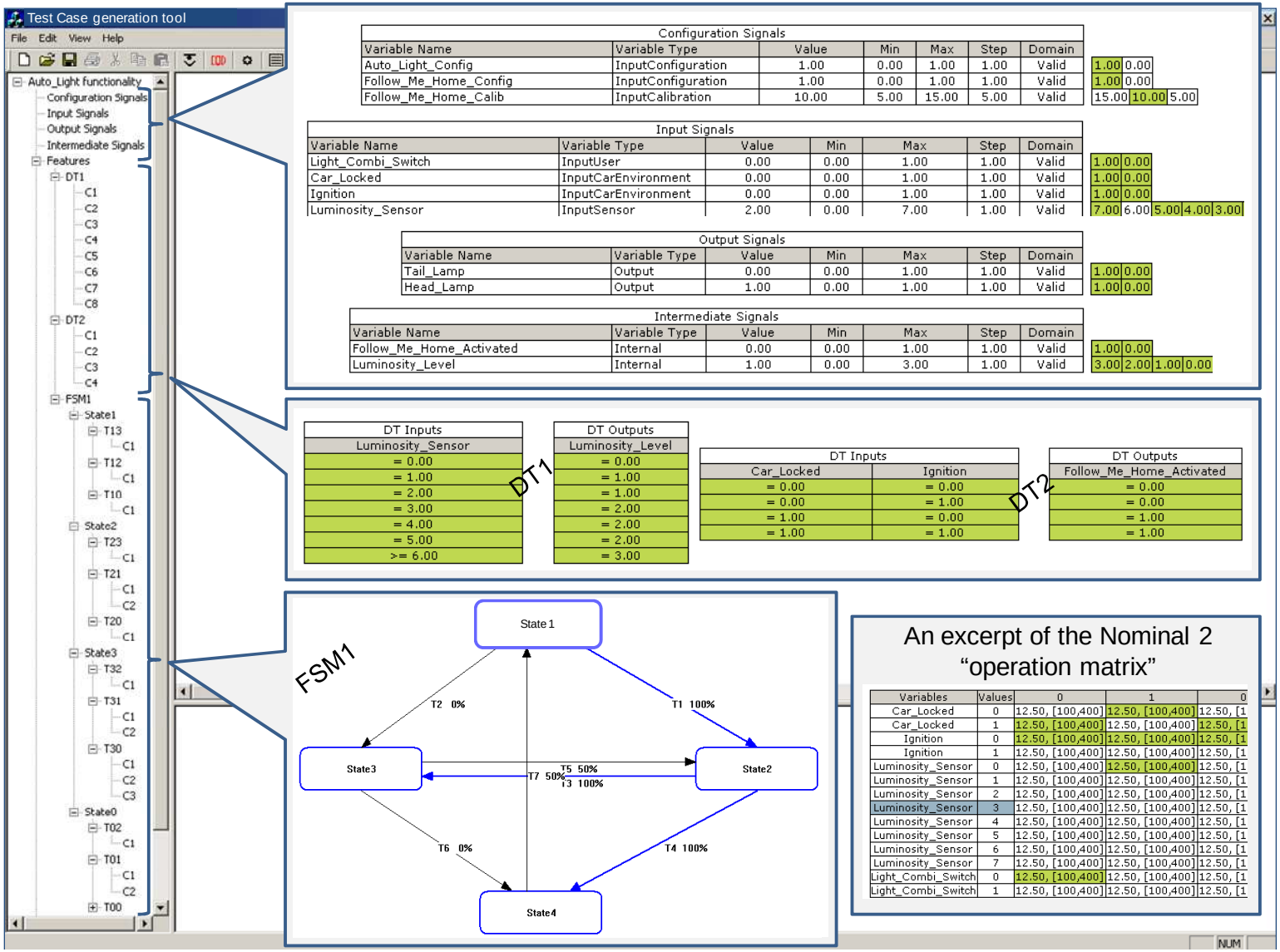


Figure 9.14 – A screenshot of the tool while highlighting the covered zones of a requirements model

V. Conclusion

The prototype presented in this chapter takes into account the impacts of our approach on the *processes*, *roles* and *tools* of the *software testing skill* within the Johnson Controls organization. A new process map for generating automatically *test cases* from the functional requirements of a functionality is presented. New *roles* and *skills* for software engineers in charge of designing *test cases* using our approach are developed. And finally, *computer tools* automating up to 70% of our approach are described. The development of these tools is not presently entirely completed. Some improvements can be done and especially on the *Graphical User Interfaces*.

In the following chapter, we analyze the results of using this prototype on two industrial case studies of practical size. We model, simulate and generate *test cases* for two software functionalities of a car.

CHAPTER 10. MODELING AND SIMULATING TWO INDUSTRIAL CASE STUDIES

I. Introduction

In order to validate our integrated framework to generate *test cases* automatically for a software module or product, we consider, at Johnson Controls, two case studies with historical data. Through these case studies, we highlight the benefits of using our approach in the *unit test* of software modules. In each case study, we consider one functionality that has already been developed and validated in the past using the software *Verification and Validation* (*V&V*) techniques currently used in Johnson Controls (Cf. *Chapter 2 – Section 5*). For each carmaker delivery (Cf. *Chapter 2 – Section 3*), historical data on the time spent to verify and validate these functionalities and on the bugs' detection by Johnson Controls and by the carmakers are available. We consider the first version of the two software modules (corresponding to the two functionalities) as it was delivered for the first time by the development team to the validation team. We also consider the version of the software functional requirements of these functionalities when delivering the software modules for the first time to the carmaker. We model, verify, validate and simulate the software functional requirements and then generate *test cases* automatically for the *unit test* of each software module. These test cases are executed on the first version of the two modules.

Our process to choose the two functionalities under experiment is described in *Section 2*. A characterization of the carmakers' requirements related to the software of these two functionalities is performed in *Section 3*. Modeling, verification and validation activities of the requirements models are respectively presented in *Section 4, 5* and *6*. A set of *"operation matrices"* for each functionality is designed in *Section 7*. Strategies to tune the generation of *test cases* are developed in *Section 8*. The generation and execution of test cases for the two functionalities are specified in *Section 9*. Finally, a deep analysis of the execution results is illustrated in *Section 10*.

II. Characterization of the two case studies

We experiment our proposals on two software functionalities of automotive electronic products developed within Johnson Controls. The choice of these functionalities has been delicate. Many criteria have guided our choices:

- C1: Recent products
- C2: Two different products
- C3: Two different carmakers
- C4: Two different management teams
- C5: Two different development teams
- C6: Two different validation teams
- C7: Two different levels of complexity (from experts point of view)
- C8: Two different types of software functional requirements
- C9: Functionalities already verified and validated using the tradition process
- C10: Functionalities well documented (historical data)
- C11: Functionalities' experts still in the company (historical data)

Based on these criteria, we choose two functionalities. The first one is the "*front wiper management*" functionality. This functionality is implemented with other functionalities in an automotive electronic product, named *body controller module*. The second one is the "*fuel gauge management*" functionality. It is implemented with other functionalities in another automotive electronic product, named *dashboard* or *cluster*. The compliance of the chosen functionalities with the predefined criteria is illustrated in *Table 10.1*.

Selection criteria	Front wiper functionality	Fuel gauge functionality
C1	Project starts in 2005. Start of serial production in 2007	Project starts in 2005. Start of serial production in 2007
C2	Body controller module of a car	Car Dashboard or Cluster
C3	Same carmaker	
C4	Two different management teams	
C5	Two different development teams	
C6	Two different validation teams	
C7	Quite complex	Very complex
C8	Formal	Informal
C9	Functionalities already verified and validated using the traditional process	
C10	Historical data are available	
C11	At least, one expert of these functionalities is still in the company	

Table 10.1 – Criteria for selecting the two functionalities

These two functionalities have already been developed and validated using the Johnson Controls present process. Some characteristics of the two software modules developed respectively for these two functionalities are given in *Table 10.2*.

	Front wiper functionality	Fuel gauge functionality
Number of input signals of the software module	**18**	**35**
Number of output signals of the software module	**9**	**25**
Size of the software module (Lines Of Code - without comments and blanks)	**1229**	**1500**

Table 10.2 – Characteristics of the two software modules developed respectively for the two functionalities under experiment

Each software module delivered to the *validation team* (first version) is considered to be verified and validated independently from its environment (other software modules). This means that a *code review*, a *static* and *dynamic analysis* and a *unit test* (Cf. *Chapter 2 – Section 5*) have been performed on each module delivered to the *validation team*. Unfortunately, at Johnson Controls, bugs detected during these *V&V* phases are often not capitalized in the *problems' database* (Cf. *Chapter 2 – Section 7.A*). Once a bug is detected, it is corrected immediately by the person who detects it. Moreover, in Johnson Controls, these phases mainly focus on answering the question: "Are we building the product RIGHT?" and not on: "Are we building the RIGHT product". In other words, the compliance with the carmaker requirements are not verified on the software modules delivered to the *validation team*. Presently, when testing unitarily a software module, the main purpose of a test engineer is to cover at 100% the *source code* of the module (Cf. *Diagnosis 8*). The *validation team* integrates the set of modules (already tested unitarily) planned for the carmaker delivery and

performs *validation tests* (Cf. *Chapter 2 – Section 5.C and 5.D*). During the *validation test*, test engineers design manually *test cases* in order to demonstrate the compliance of the whole software product (integration of at least two software modules) with the carmaker requirements (Cf. *Chapter 2 – Section 6*). Bugs detected by inner engineers during the *validate test* stage (before the delivery) and by the carmaker engineers (after the delivery) are capitalized. The distributions of bugs related to the internal behavior of the two functionalities are illustrated in *Figure 10.1*. Until the last carmaker delivery, 22 bugs were detected on the software module of the *front wiper* functionality and 23 bugs on the one of the *fuel gauge* functionality. These bugs were detected, on the two functionalities, before (*validation test*) and after (*carmaker test*) the carmaker deliveries. Unfortunately, we do not have any information on the bugs detected during the other Johnson Controls *V&V* activities (code review, unit test …). In fact, after analyzing the total number of the bugs of *Figure 10.1*, we came up to the conclusion that all these bugs could be detected earlier in the process (during the *unit test* stage). In fact, during the *unit test* of a software module and in addition to a 100% *code coverage*, test engineers should verify the compliance of each software module with the carmaker requirements. We call this the ***functional unit test***. In order to comment the *Figure 10.1*, let us consider the *front wiper* functionality example. 17 bugs were detected by the Johnson Controls *validation test* and 5 bugs by the carmaker after intermediate delivery. It must be noted that, after developing the software module of the *front wiper* functionality for the first time, only 12 bugs were detected during the first validation stage. Therefore, a delivery was performed and the carmaker immediately detected 2 more bugs. In the meantime and before the second carmaker delivery, test engineers tried to improve their *test cases* and design some new *test cases*. In consequence, they have been able to detect one more bug and after the second intermediate carmaker delivery, no new bug was detected by the carmaker. For the 4th intermediate delivery, no new *test cases* have been developed. The complete scenario of bugs' detection until the last carmaker delivery of the two functionalities is summarized in the histogram of *Figure 10.1*.

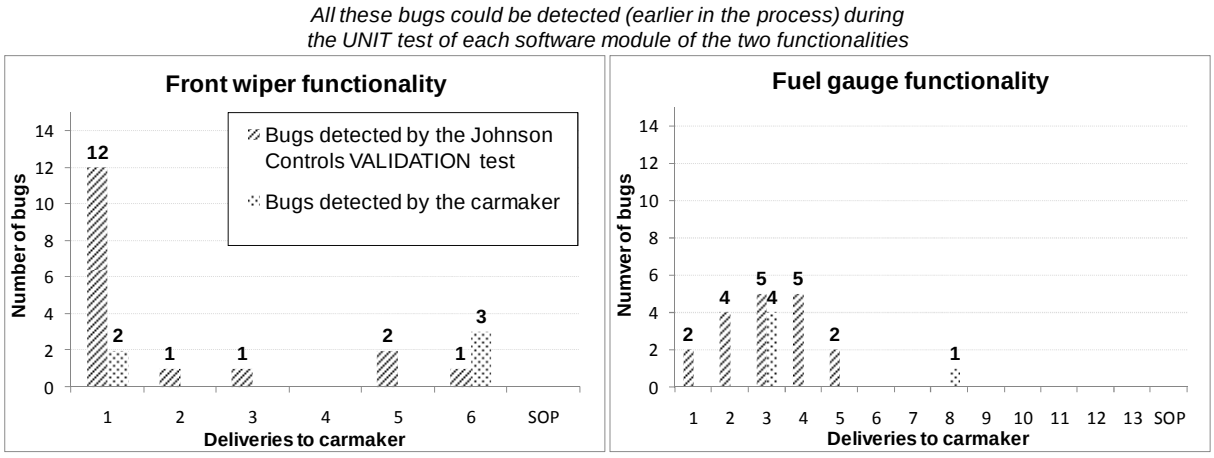


Figure 10.1 – Distribution across the carmakers' deliveries of the bugs detected on the two functionalities

Among the bugs detected on the two functionalities, some of them are considered to be more critical than others. *Severity* and *Occurrence* are two *attributes* of the current bug model and are filled at 99% for each capitalized bug (Cf. *Chapter 2 – Section 7.B*). These *attributes* are not free fields. Indeed, a set of predefined values for each attribute has been defined by Johnson Controls software experts (Cf. *Table 10.3*).

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

Table 10.3 – Severity and Occurrence levels as it was defined by Johnson Controls software experts (Johnson Controls source)

Despite the predefined values and according to experts, the attribution of a *severity* and an *occurrence* for a bug detected internally remains a *subjective* question. In fact, most of the test engineers do not have a global view of the system in order to assess the impact of the detected bug on the *end-user*. However, the *severity* and *occurrence* of bugs detected by the carmakers are more relevant since they are set by the carmaker itself. The distribution of bugs, detected on the two functionalities, across the couple (*Severity*, *Occurrence*) is presented in *Figure 10.2*. For the *front wiper* functionality, up to 76% of the bugs are (*Minor*, *Systematic*) and for the *fuel gauge* functionality, up to 72% of the bugs are (*Major*, *Systematic*). These results could be explained from two different points of view. *The first one* confirms the notion of subjectivity in defining a criticity level for a bug. In fact, the bugs of these two functionalities were described by two different teams in two different countries. *The second one* is related to the fact that the functionality of managing the *fuel level* in a car is more critical than the one managing the *wipers*. As a consequence, bugs on the *fuel gauge* functionality are considered to be more critical than the ones of the *front wiper* functionality. Moreover, bugs detected by the carmakers are often considered as critical.

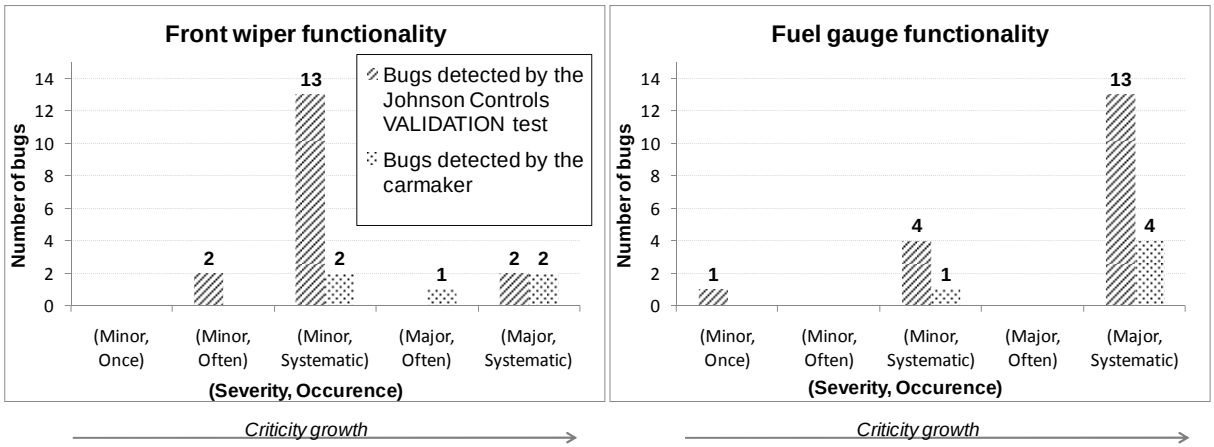


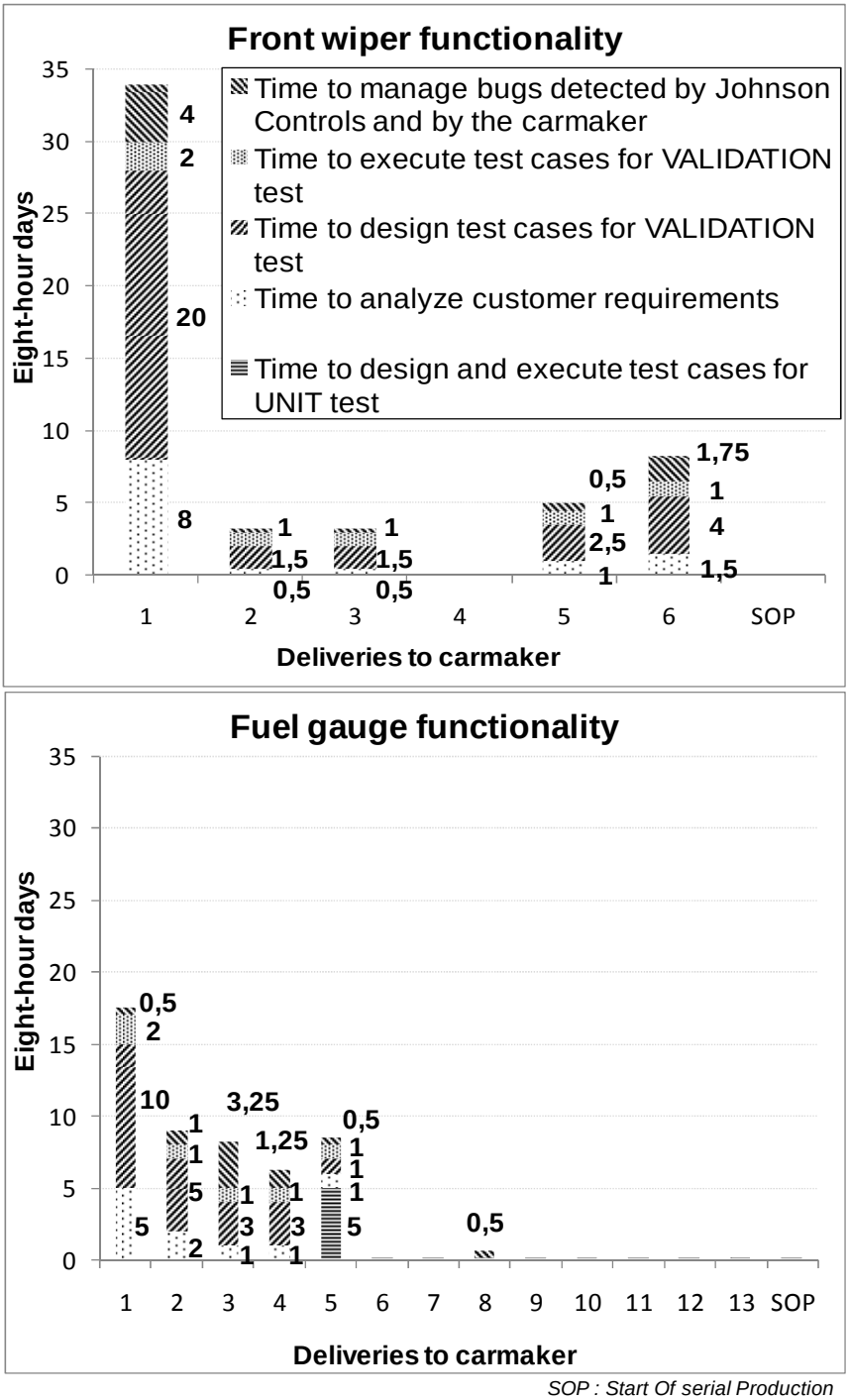
Figure 10.2 – Distribution across the couple (Severity, Occurrence) of the bugs detected on the two functionalities

In *Figure 10.3*, we illustrate the time spent by the project team in order to debug the software modules of the two functionalities using the conventional testing techniques (*unit test* and *validation test*, Cf. *Chapter 2 – Section 5*). The main activities done are:

- Design and execute *test cases* for the *unit test* of each software module (*Unit test*).
- Analyze the carmaker requirements in order to design *validation test cases.*
- Design and execute test cases for the *validation test* of each functionality.
- Manage the bugs detected internally and by the carmaker.

We note that up to 50% and 10% of the total time spent in verifying and validating a software functionality were respectively spent to manually design the *test cases* and manage the bugs detected by the carmakers. Using the current Johnson Controls testing practices,

approximately 54 *eight-hour days* were spent to test the *front wiper* functionality and 50 *eight-hour days* for the *fuel gauge* functionality.



SOP : Start Of serial Production

Figure 10.3 – An estimate of the time spent during each delivery to test the two functionalities using the conventional testing techniques

In our experiment, we propose to perform a *functional unit test* on the software modules of the two functionalities. In other words, we plan to verify unitarily the compliance of each software module with its functional requirements. To do this, we use our new approach to design *test cases* (Cf. *Chapter 5, 6, 7, 8* and *9*).

III. Characteristics of the software functional requirements of the two functionalities

The core of our approach is the modeling of software functional requirements. Therefore, one important criterion while choosing the functionalities of the two case studies was the diversity of the software functional requirements. In fact, we want to prove that whatever the formalism used by the carmaker to express the requirements related to software, one can use our approach to generate *test cases* automatically. In *Chapter 2 – Section 4.A*, we present the result of a study that we performed on the diversity, typology and evolution of these requirements within Johnson Controls. Moreover, in *Chapter 4 – Section 5.C*, we identify three formalisms of software functional requirements (*Informal*, *Semi-formal* and *Formal*). Some characteristics of the software functional requirements of the chosen functionalities are presented in *Table 10.4*.

	Front wiper functionality	Fuel gauge functionality
Formalism (Cf. *Chapter 4 – Section 4.D.1*)	**Formal** (Statechart)	**Informal** (Natural language specifications)
Size of the software functional requirements document (Number of pages in Microsoft Word format)	**30**	**30**

Table 10.4 – Characteristics of the software functional requirements of the two functionalities

IV. Modeling the software functional requirements of the two functionalities

Three stages have been necessary for modeling the software functional requirements. *The first one* consists of analyzing and understanding the requirements with our modeling language. A loop process was initiated with inner experts in order to well understand and clarify the requirements. *The second one* consists of sketching “on paper” the requirements models. We identify the *input*, *output* and *intermediate signals* and the *elements* (*Decision Tables* or *Finite State Machines*) of each functionality. Then, we develop each *element* by identifying all the *states*, *transitions* and *conditions* of the *elements* (Cf. *Chapter 5*). *The third and last stage* has been the computerization of the requirements models using the *Test Case Generation tool* that we developed (Cf. *Chapter 9 – Section 4.A*). A comparison of the time spent in each of these stages for the two functionalities is illustrated in *Table 10.5*.

Time in eight-hour days	Front wiper functionality	Fuel gauge functionality
Time spent to **analyze** the requirements before starting modeling task	**3**	**3**
Time spent to **design “on paper”** the requirements model	**5**	**7**
Time spent to **computerize** the paper requirements model	**12**	**6**
TOTAL	20	16

Table 10.5 – Time spent to design the requirements model of the two functionalities

In fact, it was more difficult and time consuming to design the requirements of the second case study (7 *eight-hour days*) than the first one (5 *eight-hour days*). The main reason is that the requirements of the second case study are expressed *informally*. However, we spent more time in computerizing the first case study (12 *eight-hour days*) than the second one (*6 eight-hour days*). Indeed, the requirements model of the first case study is bigger than the second one in terms of number of *signals*, *elements*, *states*, *transitions* and *conditions*. In *Table 10.6*, we illustrate the characteristics of the requirements models of the two case studies. The requirements model of the "*front wiper*" functionality has 19 Decision Tables and 5 *Finite State Machines*, while the one of the "*fuel gauge*" functionality has 2 *Decision Tables* and 4 *Finite State Machines*.

	Front wiper functionality	Fuel gauge functionality
# of input signals	18	6
# of output signals	9	8
# of intermediate signals	24	31
# of Decision Tables	19	2
# of Conditions in DT	289	110
# of Finite State Machines	5	4
# of States in FSM	36	53
# of Transitions in FSM	119	158
# of Conditions in FSM	154	197

Table 10.6 – Characteristics of the requirements models of the two functionalities

V. Verifying the requirements models of the two functionalities

In *Chapter 6 – Section 4*, we developed a set of *integrity rules* to be checked on each requirements model in order to verify its correctness. The verification of the requirements models of the two functionalities was performed manually and automatically. In fact, when sketching "on paper" the requirements model, we verify manually the fulfillment of the *integrity rules*. Moreover and after computerizing the model, the *Test Case Generation tool* (*Cf. Chapter 9 – Section 4.A*) allows to check automatically these rules. Therefore, the time spent in verifying these models is integrated to the time of sketching "on paper" and computerizing the models (Cf. *Table 10.5*). We stop verifying a requirements model when all the *integrity rules* are checked OK on the model. After verifying the developed requirements models (manually and automatically), around 30 rules violations are detected on each model. The distribution of these violations over the set of *integrity rules* is presented in *Figure 10.4*. We first model the "*front wiper*" functionality then the "*fuel gauge*" one. As a consequence, violations in *Rules 1*, *14*, *15* and *18* were detected on the first case study and not on the second. In fact, when modeling the "*fuel gauge*" functionality, we focus on respecting the rules already violated on the previous case study. Moreover, up to 80% of the violations on the first case study (*Rule 8*) are related to the fact that the domains of the functionality's *input*, *output* and *intermediate signals* are not covered by *conditions* and *actions* in *elements*. Since the requirements model of the second case study is smaller than the one of the first case study (Cf. *Table 10.6*), it capitalizes up to 60% of *Rule 8* violation. The remaining 40% is shared out between the *Rules 4*, *5*, *6* and *7*. This is due to the fact that the requirements of the "*fuel gauge*" functionality are expressed *informally* (Natural language).

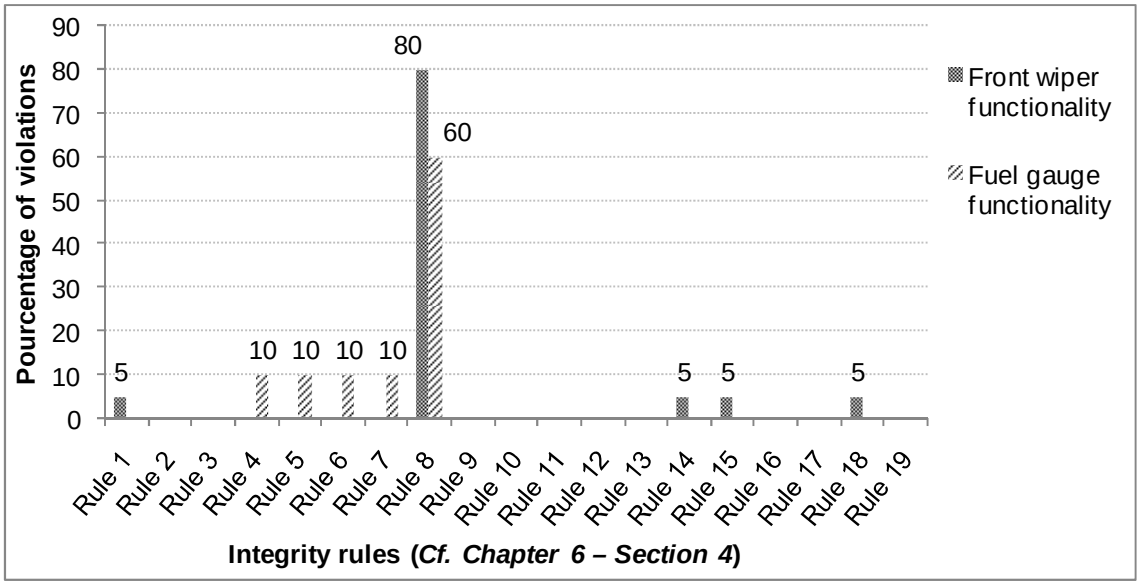


Figure 10.4 – Distribution of violations over the integrity rules

VI. Validating the requirements models of the two functionalities

Once the requirements models are designed, computerized and verified, we validate these models. In other words, we verify that the developed requirements models are compliant with the carmaker requirements related to the software domain. In *Chapter 6 – Section 5*, we propose three scenarios to validate a requirements model:

- *First scenario:* Animate our requirements model
- *Second scenario:* Simulate *test cases* delivered by the carmaker on our requirements model
- *Third scenario:* Compare our requirements model to another valid model of the requirements

These scenarios can be used concurrently or separately. However, it is mandatory to have the data necessary for performing one scenario or another. For instance, in case of the first case study, the carmaker has delivered a simulation model of the software functional requirements and one *test case* (about 10000 *test steps*). Therefore, the three scenarios are applicable. On the contrary, the carmaker requirements of the second case study cannot be simulated and no carmaker *test cases* are available. For that reason, only the first scenario can be applied in the second case study. Because of time constraints, we only apply the first and second scenarios for the first case study and the first scenario for the second case study. One main question is: *When to stop validating a model?* In fact, we consider a tradeoff between the quality of the model and the resources (time, people, and cost) spent in validation (Cf. *Chapter 6 – Section 2.B*). *On the one hand*, we spent *5 eight-hour days* validating the requirements model of the first case study and we detect around 15 nonconformities between the model and the requirements as it was delivered by the carmaker. *On the other hand*, we spent 20 *eight-hour days* validating the second case study and we detect around 50 nonconformities. Even if the requirements model of the first case study is bigger than the one of the second case study (Cf. *Table 10.6)*, we spent more time and detected more nonconformities in validating the second case study. Indeed, the main reason of this result is that the requirements delivered by the carmaker for the second case study are *informal*, while the ones for the first case study are *formal*. In the following, we detail the validation process of the two requirements models.

In case of the first case study, we first simulate on our requirements model the *test case* (about 10000 *test steps*) delivered by the carmaker. Once a nonconformity is detected, the requirements model is corrected before restarting the simulation of the *test case*. The cumulated number of nonconformities detected on the first case study is presented in *Figure 10.5*. After the 2000th *test step*, no more nonconformities are detected on the model. Afterwards and in order to increase the confidence in our model, we propose to animate it by an expert. Two *simulation plans* of 100 steps each (*operations* and *inter-operation times)* have been designed by an expert and simulated "*step by step*" on the model (Cf. *Chapter 8 – Section 4.E*). No nonconformities have been detected. In consequence, we decide to stop validating the model.

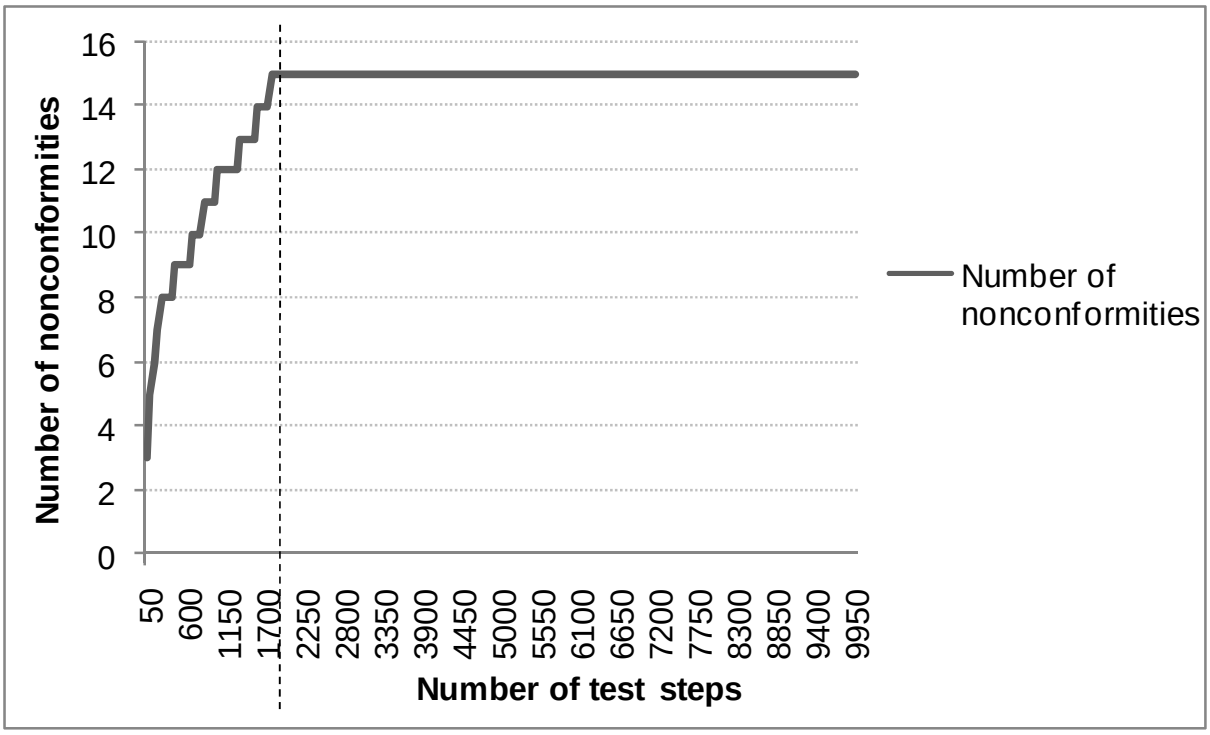


Figure 10.5 – Cumulated number of nonconformities on the first case study (Second scenario)

In case of the second case study, we animate the model by an expert. Three *simulation plans* of 300 steps each (*operations* and *inter-operation times*) have been designed and simulated successively "step by step" on the model (Cf. *Chapter 8 – Section 4.E*). Once a nonconformity is detected, the requirements model is corrected before restarting the simulation. The cumulated number of nonconformities detected on the second case study is presented in *Figure 10.6*. Through the first set of 300 steps, we detect and correct 27 nonconformities. The second one allows to detect and correct 14 nonconformities. The third one detects 8 nonconformities. The main question was: Are there other nonconformities in the model? To answer this question, we design a new *simulation plan* of 50 steps and we simulate it on the model. In fact, this *simulation plan* allows to detect new nonconformities. At this moment, we realize the difficulty of validating at 100% a model and we decide to consider a tradeoff between the quality of the model and the time spent in validation. In fact, through the three *simulation plans*, we spent up to 20 *eight-hour days* simulating and debugging our requirements model and we cover at 90% the requirements model (*signals* and *elements*). Therefore, we decide to stop validating the model.

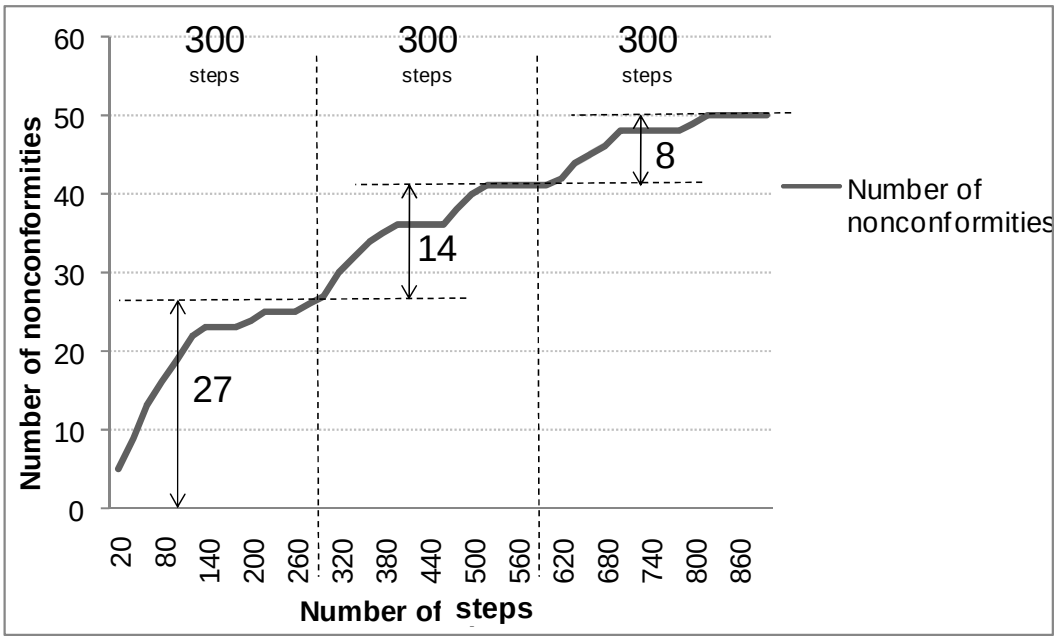


Figure 10.6 – Cumulated number of nonconformities on the second case study (First scenario)

VII. Designing "operation matrices" for the two case studies

The generation of *test cases* is performed based on the concept of *"operation matrix"* (Cf. *Chapter 7 – Section 2*). Through an *"operation matrix"*, inner engineers can enrich the requirements model with knowledge on the user (driver) recurrent operations (Cf. *Chapter 8 – Section 4*) and the test engineers' experience (Cf. *Chapter 8 – Section 5 and 6*). However, one major question is: How engineers can design an *"operation matrix"*? Five possible scenarios are identified (Cf. *Chapter 9 – Section 3 and 4*):

1. Design manually one or more *"operation matrices"* and export them to the *Test Case Generation tool.*
2. Generate the *Nominal "operation matrices"* (*Nominal 1* and *Nominal 2*) automatically via the *Test Case Generation tool.*
3. Design manually a set of *constraints* on the *input signals* of the requirements model, export them to the *Test Case Generation tool* and generate a *Driver profile "operation matrix"* automatically.
4. Generate via the *Bugs Reuse tool* one or more *Bug "operation matrices"* from one or more capitalized bugs and export them to the *Test Case Generation tool.*
5. Generate via the *Test Cases Reuse tool* one or more *Test Case "operation matrices"* from one or more capitalized *test cases* and export them to the *Test Case Generation tool.*

The number of cases of an *"operation matrix"* for the *"front wiper"* functionality is 9604 (98x98, 98 is the number of possible *operations* on the functionality). 7921 (89x89) is the one for the "*fuel gauge*" functionality. Therefore, it was ridiculous to think of manually designing *"operation matrices"* for these functionalities (Cf. *Chapter 7 – Section 2*).

As a basic solution, we generate, via the *Test Case Generation tool*, the **two *Nominal "operation matrices"*** for the two functionalities (Cf. *Chapter 7 – Section 2*, Cf. *Chapter 9 – Section 4.B*). According to experts, we define one standard *time interval* and we affect it to all successive *operations*.

Moreover, experts design manually a set of *constraints* on the *input signals* of each functionality (Cf. *Chapter 8 – Section 4*) that we export to the *Test Case Generation tool*. Based on these *constraints*, the *Test Case Generation tool* generates **a *Driver Profile "operation matrix"*** for each functionality (Cf. *Chapter 9 – Section 4.D*). The number and type of the designed *constraints* is illustrated in *Table 10.7*.

Type of constraints (Cf. *Chapter 8 – Section 4*)	Front wiper functionality	Fuel gauge functionality
Logical	**1**	**1**
Conditional	**3**	**1**
Succession	**2**	**1**
Timing	**1**	**1**

Table 10.7 – Constraints designed for the two functionalities

Unfortunately, we do not have enough time to analyze bugs and *test cases* capitalized on previous projects implementing the "*fuel gauge*" functionality. In fact, we decide to focus our effort on the "*front wiper*" functionality. In *Chapter 2 – Section 7.C*, we perform a study on the bugs detected on the "*front wiper*" functionality through 5 different projects since 1997 and till 2007. Excluding the last project (*Project 5*) which is the one on which we carry out our experiments, 55 bugs were detected on this functionality since 1997. In *Chapter 8 – Section 5*, we propose two strategies to reuse capitalized bugs. One strategy (Cf. *Chapter 8 – Section 5.A*) consists of representing the "*Problem description*" of bugs in a specific format in order to generate a *Bug "operation matrix"* for each bug. One difficult task was to represent the "*Problem description*" of the 55 identified bugs into our specific format. Based on the experts' advices, we only consider the 10 most critical bugs providing that there exists enough information related to the "*Problem description*" of the bug. Afterwards, we generate, via the *Bugs Reuse tool* (*Cf. Chapter 9 – Section 3.C.1*), the corresponding **10 *Bug "operation matrices"*** that we export to the *Test Case Generation tool*. A glossary of the *input signals* names on the previous and current projects was necessary.

Over the 4 projects implementing the "*front wiper*" functionality (Cf. *Chapter 2 – Section 7.C*), only one project *P* has adopted the *test case format* presented in *Definition 2.11*. This format of *test cases* is required for generating a *Test Case "operation matrix"* automatically for each *test case* (Cf. *Chapter 8 – Section 6*). Within *P*, test engineers have designed one *test case* (about 2000 *test steps*) in order to test the "*front wiper*" functionality. Based on this *test case*, we generate, via the *Test Cases Reuse tool* (*Cf. Chapter 9 – Section 3.C.2*), **one *Test Case "operation matrix"*** that we export to the *Test Case Generation tool*. A glossary of the *input signals* names on the previous and current projects was necessary.

A summary of the *"operation matrices"* designed for the two functionalities is illustrated in *Table 10.8*. We also estimate the time spent in designing these *"operation matrices"*. For the *front wiper* functionality, we spent 2 *eight-hour days* and for the *fuel gauge* functionality, 0,5 *eight-hour days*. In fact, identifying and preparing the capitalized bugs and *test cases* have taken about 1,5 *eight-hour days*.

Front wiper functionality	Fuel gauge functionality
2 *Nominal* « operation matrices » (*Nominal 1* and *Nominal 2*)	2 *Nominal* « operation matrices » (*Nominal 1* and *Nominal 2*)
1 *Driver Profile* « operation matrix »	1 *Driver Profile* « operation matrix »
10 *Bug* « operation matrices »	-
1 *Test Case* « operation matrix »	-

Table 10.8 – "Operation matrices" designed for the two functionalities

VIII. How to tune the generation of test cases?

Three questions have been raised at this stage of the experiment:

- From which *"operation matrix"* do we start generating *test cases*?
- How to tune the *coverage* objectives and the time and cost constraints?
- How to tune the test generation algorithm?

In order to answer the first question, we propose to generate *test cases* from the *"operation matrices"* according to the order presented in *Figure 10.7*. *Firstly*, we generate *test cases* from the *Bug "operation matrices"*. At least, we ensure that our software module is free from bugs similar to the ones already detected in the past. *Secondly*, we generate *test cases* from the *Test Case "operation matrices"*. These *test cases* are suitable to detect bugs since they are based on the test engineers' experience. *Thirdly*, we generate *test cases* from the *Driver Profile "operation matrix"*. This aims to check that the software module fulfills the *end-user* (driver) expectations. *Finally*, we generate *test cases* from the *Nominal "operation matrices"*. Improbable successions of *operations* are generated in order to check the robustness of the software module. In the previous section, we note that no *Bug* or *Test Case "operation matrices"* have been designed for the second case study. Moreover and according to experts, simulating random *operations* (*Nominal "operation matrices"*) on the *fuel gauge* functionality does not make real sense. Therefore, for this case study, we only generate test cases from the *Driver Profile "operation matrix"*.

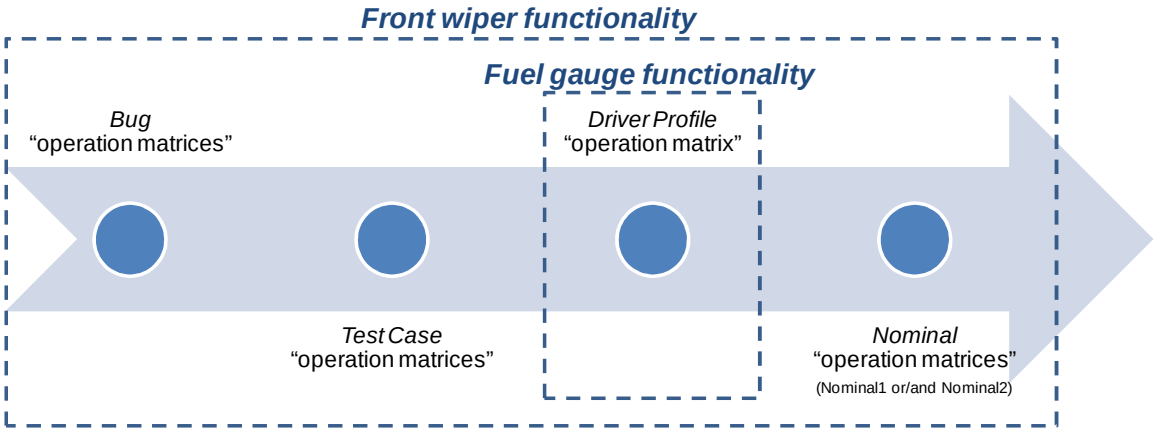


Figure 10.7 – Our strategy of generating test cases from the "operation matrices"

Before generating *test cases* from an *"operation matrix"*, we have to define the objectives and constraints of the generation (Cf. *Chapter 7 – Section 5*, Cf. *Chapter 9 – Section 4.F*) but also the parameters of the optimization algorithm (Cf. *Chapter 7 – Section 6*, Cf. *Chapter 9 – Section 4.F*). In our case studies, we tune these factors based on a *try-and-test* protocol and on the experts' knowledge.

According to the type of the *"operation matrix"*, we propose guidelines for defining the *coverage* objectives and the time and cost constraints (Cf. *Table 10.9*). These guidelines have been defined based on our analysis of the different *"operation matrix"* modes. For instance, in case of a *Bug* or *Test Case "operation matrix"* mode, the knowledge extracted from capitalized bugs or *test cases* is incorporated in the *"operation matrix"*. Therefore, it is necessary to cover at least all the successions of *operations* of a *Bug* or *Test Case "operation matrix"*. The constraints' values depend on the context (budget, planning, and resources) of the project.

"Operation matrix" from which the test case has to be generated	Objectives guidelines	Constraints guidelines
Bug « operation matrix »	At least, cover at 100% the « operation 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
Test Case « operation matrix »	A least, cover at 100% the « operation matrix »	
Driver Profile « operation matrix »	A least, cover at 100% the domains of the input signals	
Nominal « operation matrix »	At most, cover the requirements model and the « operation matrix »	

Table 10.9 – Guidelines for defining the objectives and constraints of a test case generation

Based on these guidelines, we set the objectives and constraints of the *test cases* generation for the two functionalities (Cf. *Table 10.10*). Because of technical reasons and based on the assumption that covering at 100% the requirements model involves that the *source code* is covered at 100%, we decide to only set objectives in terms of functional coverage. From the opposite direction, this assumption is rarely true (Cf. *Chapter 7 – Section 4*). Moreover, we do not consider the criticity of the requirements (*Critical Successive 2-Operations coverage*, *DT Critical Conditions coverage* and so on – Cf. *Chapter 7 – Section 4.B*). Finally, no constraints are set in terms of number of *test steps* and execution time of the generated *test cases*.

	Front wiper functionality				Fuel gauge functionality			
"Operation matrices"	Bug	Test Case	Driver Profile	Nominal2	Bug	Test Case	Driver Profile	Nominal2
Objectives								
Functional coverage								
Inputs domains	-	-	**100-10**	**100-10**	-	-	**100-10**	-
Outputs domains	-	-	-	**100-10**	-	-	-	-
Intermediates domains	-	-	-	**100-10**	-	-	-	-
Inputs boundaries	-	-	-	**100-10**	-	-	-	-
Outputs boundaries	-	**Target - Weight**	-	**100-10**	-	-	-	-
Intermediates boundaries	-	-	-	**100-10**	-	-	-	-
Successive 2-Operations	**100 - 10**	**100-10**	-	**100-10**	-	-	-	-
DT Conditions	-	-	-	**100-10**	-	-	-	-
FSM States	-	-	-	**100-10**	-	-	-	-
FSM Transitions	-	-	-	**100-10**	-	-	-	-
FSM Conditions	-	-	-	**100-10**	-	-	-	-
Constraints								
Test execution time and cost								
Test Execution Time (en ms)	-	-	-	-	-	-	-	-
Test Step Number	-	-	-	-	-	-	-	-

Table 10.10 – Objectives and constraints when generating test cases for the two functionalities

Once defining objectives and constraints, we have to tune the optimization algorithm of the *test case* generation. In *Chapter 7 – Section 6*, we describe the optimization algorithm and its parameters. Eight parameters have been identified. In our case studies, we tune these parameters based on the traditional *try-and-test* protocol. In fact, we first set the objectives and constraints of the *test case* generation, then we set specific values on the optimization algorithm parameters and finally we generate *test cases*. Based respectively on the fulfillment and respect of the objectives and constraints, we adjust the optimization parameters. The purpose is to better fulfill and respect the *coverage* objectives and the time and cost constraints. For each case study and after 10 trials (approximate), the "optimal" values for the optimization algorithm parameters are identified in *Table 10.11*. We spent 1 *eight-hour day* in adjusting these parameters for the two case studies. Let us consider the first case study. We have to generate *test cases* from a *Bug "operation matrix"*. According to *Table 10.9*, we first set the objectives and constraints of the *test case* generation (Cover at 100% the *Bug "operation matrix"*). Afterwards, we tune the parameters of the optimization algorithm (Cf. *Table 10.11*). In fact, we decide to optimize the *coverage* of the *Bug "operation matrix"* (*Parameter 1* = 1). When choosing a new *operation* in the *"operation matrix"*, the optimization algorithm checks if the corresponding succession of *operations* is already covered or not. If it is the case, another *operation* is chosen until a non-covered succession of *operations* is selected. The maximum number of unsatisfied trials, before the algorithm exits the loop, is 30 (*Parameter 2* = 30). Even if the designed *test step* does not improve the objectives fulfillment, we decide to add it to the *test case* under construction (*Parameter 3* = 0 and *Parameter 4* = 0). After 30 *test steps* generated without an improvement in the objectives fulfillment, we decide to stop designing *test steps* for the corresponding *test case* (*Parameter 5* = 30). According to experts, we decide to generate *test cases* for only one *"Configuration"* of the *front wiper* functionality (*Parameter 6* = 0 and *Parameter 7* not defined). In fact, we choose the basic (by default in a car) *"Configuration"* of the functionality. Finally, only one *test case* has to be generated (*Parameter 8* = 1).

	Front wiper functionality				Fuel gauge functionality			
"Operation matrices"	Bug	Test Case	Driver Profile	Nominal	Bug	Test Case	Driver Profile	Nominal
Parameter 1	1				-	-	1	
Parameter 2	30				-	-	90	
Parameter 3	0				-	-	0	
Parameter 4	0				-	-	0	
Parameter 5	30				-	-	30	
Parameter 6	0				-	-	0	
Parameter 7	-				-	-	-	
Parameter 8	1	6	6	6	-	-	3	

Table 10.11 – Optimization parameters when generating test cases for the two functionalities

IX. Generation and execution of the test cases on the software modules of the two functionalities

In *Section 7*, we develop the *"operation matrices"* designed for the two case studies. In *Section 8*, objectives, constraints and optimization parameters for the generation of *test cases* for the two functionalities are defined. In this section, we describe the generation and execution of *test cases*. The number and characteristics of the generated *test cases* are illustrated in *Figure 10.8* and *10.9*. In fact, the generation of *test cases* has been carried out automatically via the *Test Case Generation tool* (Cf. *Chapter 9 – Section 4.F*).

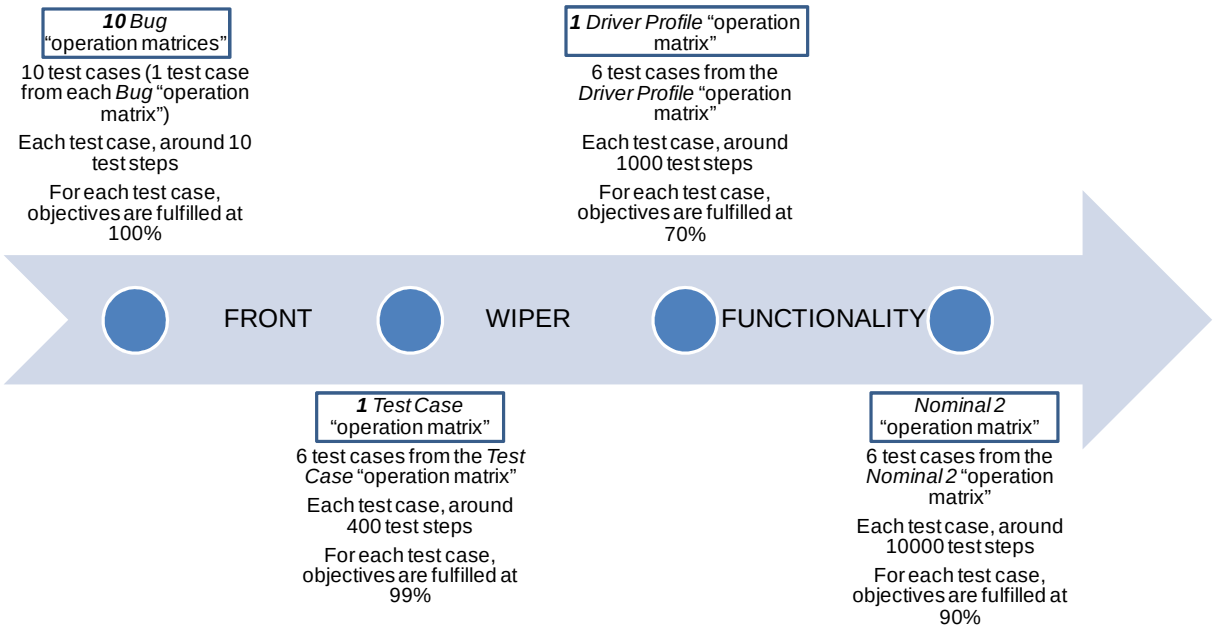


Figure 10.8 – Order of generating and executing test cases for the front wiper functionality

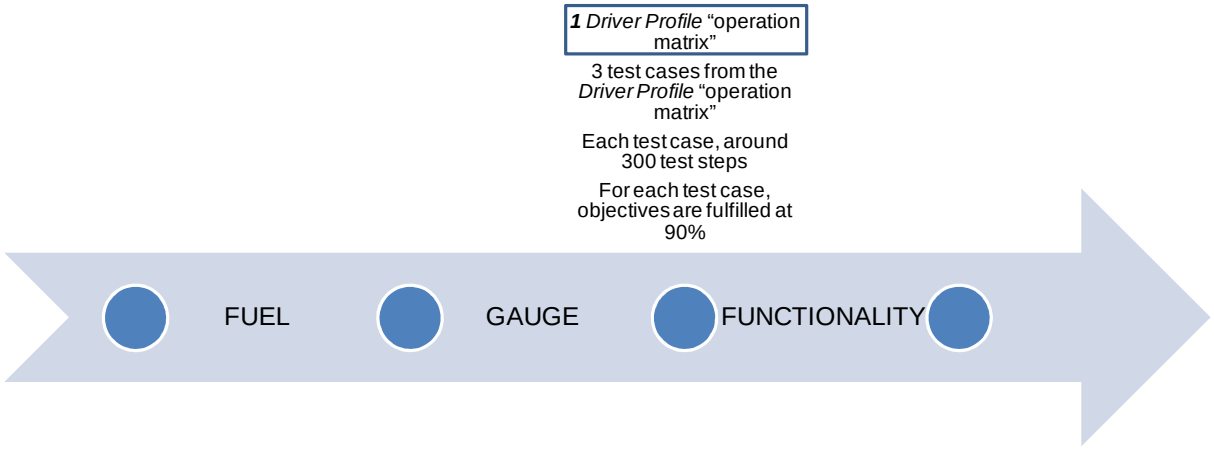


Figure 10.9 – Order of generating and executing test cases for the fuel gauge functionality

As we plan to do a ***functional unit test*** of the two functionalities (Cf. *Section 2*), we execute the generated *test cases* on the first version of the two corresponding software modules (as it was delivered for the first time by the *development team* to the *validation team*). In other words, we isolate the first version of the software module which fulfils the *front wiper* functionality and the one of the *fuel gauge* functionality and we test them through the generated *test cases*. To do this, we first translate these *test cases*, via an inner tool, into the *unit test language*. It is a computer language understandable by the *unit test execution platform* (Cf. *Appendix B* and *C*). The execution is performed following the order defined in *Figure 10.8* and *10.9*. Once an anomaly is detected, we analyze it in order to identify its origin. The origin can be:

- A *bug in the requirements model*,
- A *known bug in the software module*. It is a bug that the *validation test* of Johnson Controls or the carmaker has already detected (Cf. *Figure 10.1*),
- An *unknown bug in the software module*. It is a bug which is not yet detected neither by the *validation test* of Johnson Controls nor by the carmaker.

Whatever the origin, the bug is corrected before restarting the execution of the *test cases*.

It is important to note that the time to generate *test cases* via the *Test Case Generation tool* and the time to execution *test cases* via the *unit test execution platform* are both trivial from the automotive industry point of view. It can be respectively estimated to 500 and 1000 *test steps* per minute. These estimations are given for reference only because they depend on many factors (*CPU*[41], *inter-operations times* of the *test steps*, parameters of the optimization algorithm and so on).

X. Analysis of the results of the two case studies

A. Detect bugs earlier in the software life cycle

Once executing all the generated *test cases* on the software modules of the two functionalities, a total of 29 anomalies were detected on the first case study and 35 anomalies on the second one. In fact, it is important to assess the *accuracy* of the results delivered by our measurement system. Therefore, we measure (Cf. *Figure 10.10*):

- The ratio between the number of *"false"* bugs (bugs in the requirements models) detected by our approach and the total number of detected anomalies. The *"false"* bugs are the anomalies that are not related to bugs in the software module under test but to bugs in the requirements model itself. As said in *Section 6*, it is impossible to validate at 100% a requirements model and therefore bugs in this model could be detected later when executing the generated *test cases* on the software under test.
- The ratio between the number of *"true"* bugs (known bugs in the software modules) detected by our approach and the total number of detected anomalies.
- The ratio between the number of *"new"* bugs (unknown bugs in the software modules) detected by our approach and the total number of bugs in the software module under test.

About 17% (5 over 29) of the anomalies detected on the *front wiper* functionality were related to bugs in the requirements model and up to 49% (17 over 35) on the *fuel gauge* functionality. This could be explained by the fact that the requirements models of the two functionalities could not be exhaustively validated (Cf. *Section 6*). More especially, the one of the *fuel gauge* functionality because of the *informal* formalism of the carmaker requirements. Around 65% (19 over 29) of the anomalies detected on the *front wiper* functionality were related to *known bugs* in the software module and up to 51% (18 over 35) on the *fuel gauge* functionality. We also detect 5 "*minor*" bugs ("*minor*" from experts' point of view) that neither the conventional *validation test* of Johnson Controls nor the carmaker test has detected on the *front wiper* functionality. According to experts, these bugs have no impact on the *end-user* (driver). It represents 19% (5 over (22+5)) of the total number of bugs in the functionality (22+5).

From another point of view, we were able to detect 86% (19 over 22) of the bugs already detected by the conventional *validation test* on the first case study and 78% (18 over 23) on the second one. **These results prove that many of the bugs detected later in the software life cycle (*Validation test*) could be detected earlier (*Unit test*) via our *functional unit test*.**

[41] CPU : Central Processing Unit

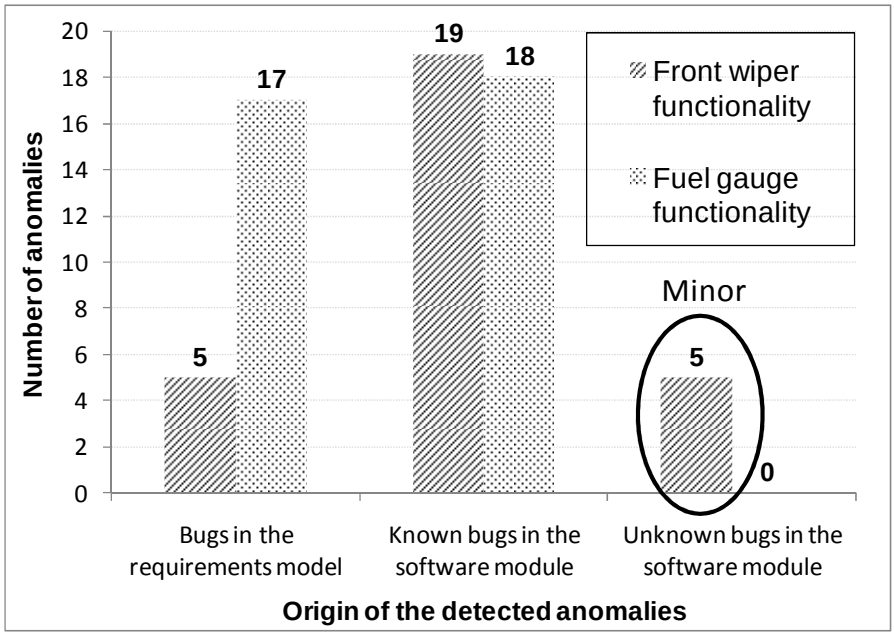


Figure 10.10 – Origin of the anomalies detected when executing the generated test cases on the two functionalities

After analyzing the remaining 3 (22-19) and 5 (23-18) *known bugs* not detected respectively on the first and second case studies, we come up to the conclusion that these bugs could be detected by our platform since we reach a 100% of the functional *coverage* (which is not the case, Cf. *Section 9*). These non-detected bugs are related to some specific functional requirements that weren't covered by our generated *test cases*. Indeed, when generating *test cases* from a *Nominal "operation matrix"*, our computational algorithms didn't succeed to reach 100% of the functional *coverage* (maximum of 90%). To overcome this lack, we have to improve our computational algorithm in order to focus on covering the non-covered zones of the requirements model. In *Figure 10.11*, we identify across the carmakers' deliveries the *known bugs* (Cf. *Figure 10.1*) not detected by our approach. In *Figure 10.12*, we illustrate the criticity (*Severity*, *Occurrence*) of the non-detected bugs as it was filled in the *problems' database* (Cf. *Figure 10.2*).

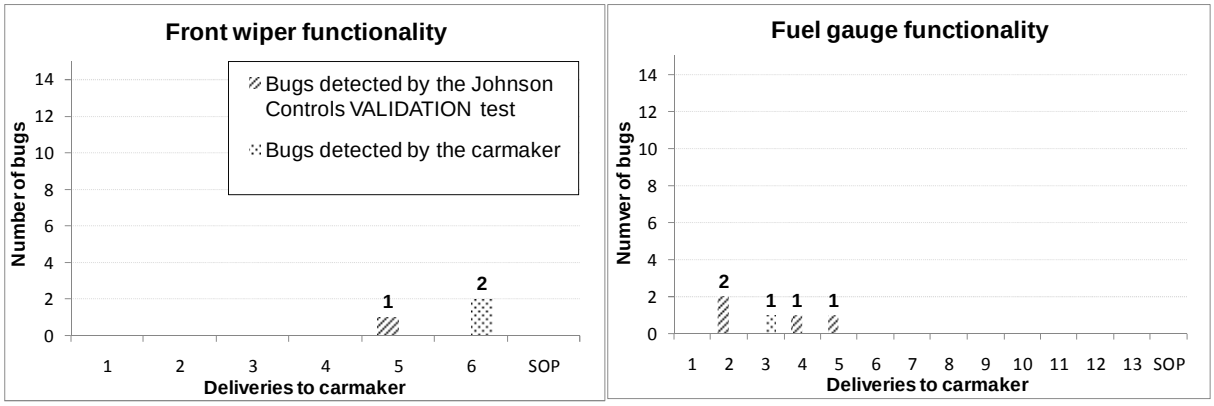


Figure 10.11 – Distribution according to the carmakers' deliveries of the known bugs not detected by our approach

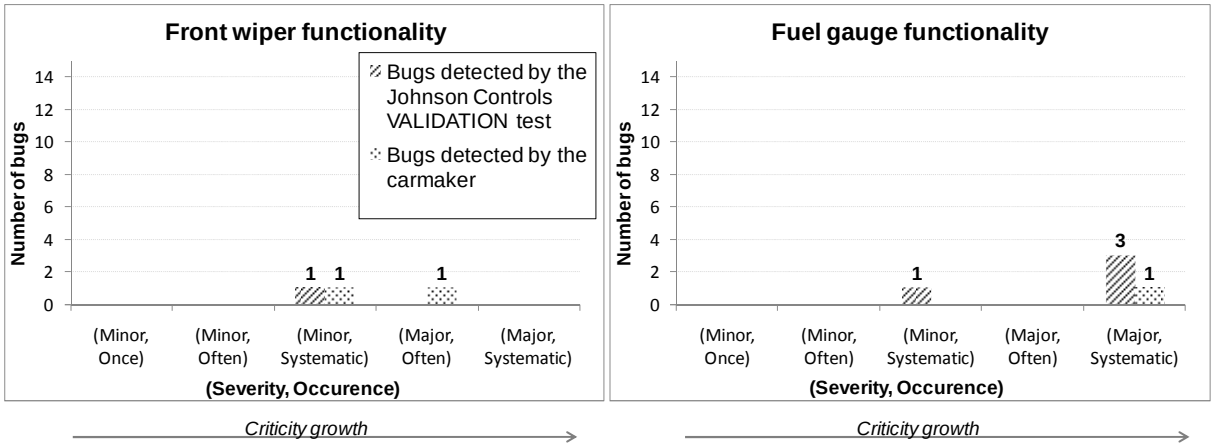


Figure 10.12 – Distribution across the couple (Severity, Occurrence) of the known bugs not detected by our approach

Among the *known bugs* detected by our approach, some of them are bugs already detected by the conventional Johnson Controls *validation test* and others by the carmaker (Cf. *Figure 10.13*). For the *front wiper* functionality, we detect 60% (3 over 5) of the bugs detected by the carmaker and 94% (16 over 17) of the bugs detected by the conventional *validation test*. For the *fuel gauge* functionality, we detect 80% (4 over 5) of the bugs detected by the carmaker and 78% (14 over 18) of the bugs detected by the conventional *validation test*.

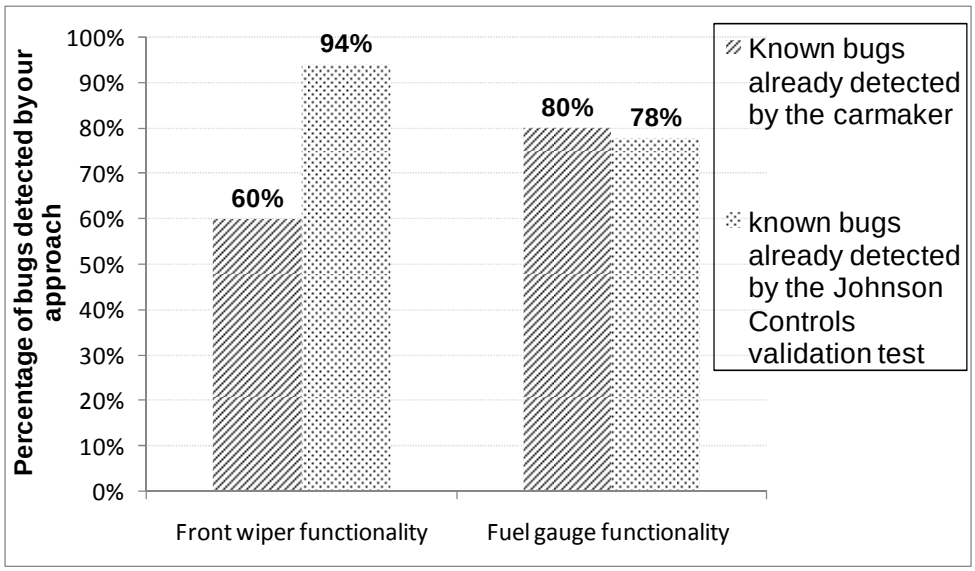


Figure 10.13 – Origin of the known bugs detected by our approach on the two functionalities

In the case of the front wiper functionality, the evolution of the cumulated number of *known* and *unknown bugs* that we detect through our approach is illustrated in *Figure 10.14*. The evolution is drawn according to the execution order of the generated *test cases* defined in *Figure 10.8*. Once a bug is detected, it is corrected before restarting the execution. Through the *test cases* generated from the *Bug "operation matrices"*, we detect 2 bugs out of the 17 bugs detected by the Johnson Controls software testing processes. The *test cases* generated from the *Test Case "operation matrix"* enable to detect 6 bugs out of the 17 bugs detected by Johnson Controls, 1 bug out of the 5 bugs detected by the carmaker after intermediate delivery and 2 new "minor" bugs that were neither detected by Johnson Controls nor by the carmaker. The *test cases* generated from the *Driver Profile "operation matrix"* enable to detect 1 bug out of the 17 bugs detected by Johnson Controls. And finally, the *test cases* generated from the *Nominal 2 "operation matrix"* enable to detect 7 bugs out of the 17 bugs

detected by Johnson Controls, 2 bugs out of the 5 bugs detected by the carmaker and 3 new "minor" bugs that were neither detected by Johnson Controls nor by the carmaker. As conclusions on the bugs' detection flow:

- All new detected bugs (5 bugs) have occurred in the *Test Case* and *Nominal 2* test stages. This could be explained by the fact that these bugs are related to specific successions of *operations*, illogical from a use point of view but could probably occur in the serial life of the software product.
- At the end of each test stage, the number of the detected bugs tends to stabilize.

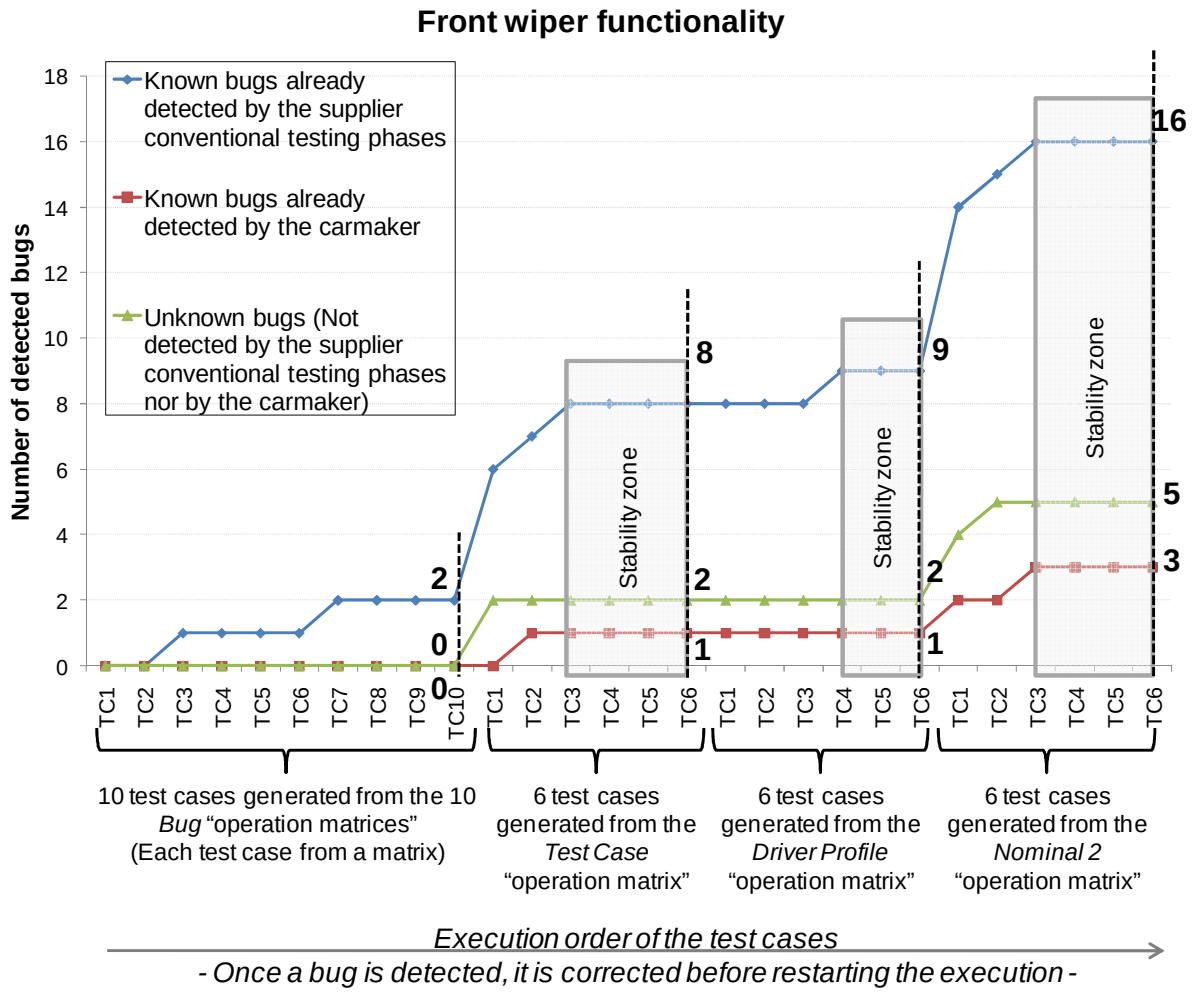


Figure 10.14 – Evolution of the cumulated number of bugs detected by our approach on the front wiper functionality

We also execute independently on the first version of the *front wiper* software module all the *test cases* generated from each mode of *"operation matrix"*. The result of this experiment is illustrated in *Figure 10.15*. We identify the number and type of bugs that can be detected by one or more modes of *"operation matrix"*. As a conclusion:

- One mode of the *"operation matrix"* wasn't able to detect all the bugs already detected by the present Johnson Controls testing processes and by the carmaker.
- Each mode has at least one bug that can only be detected via this mode.
- The *Nominal 2* mode *"operation matrix"* detects the maximum number of bugs. This could be explained by the fact that we generate 60000 *test steps* from this *"operation matrix"* and we cover at 90% the requirements model.

- The *Test Case* mode *"operation matrix"* detects up to 80% of the bugs that the *Drive Profile* mode can detect. In fact, the capitalized *test cases* have been designed with an *end-user* point of view.

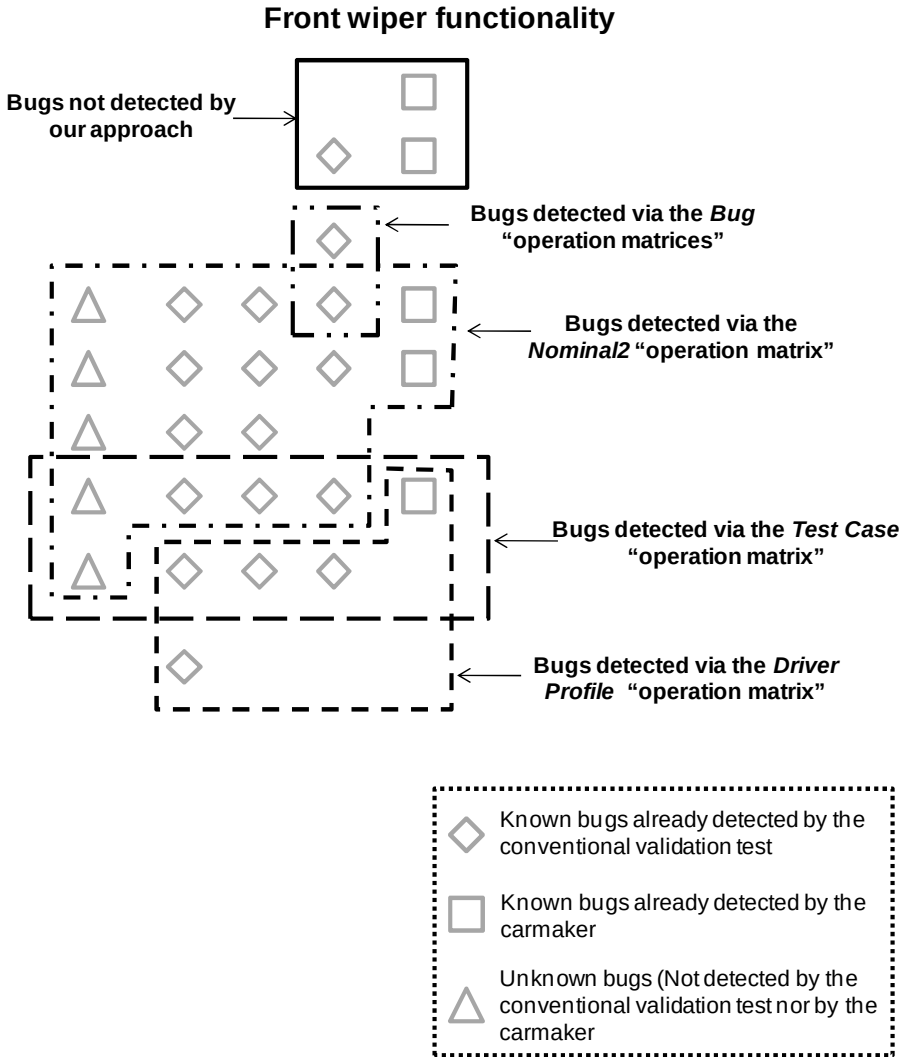


Figure 10.15 – Number and type of bugs detected via each "operation matrix" mode

In the case of the fuel gauge functionality, the evolution of the cumulated number of *known* and *unknown bugs* that we detect through our approach is illustrated in *Figure 10.16*. Through the *test cases* generated from the sole *Driver Profile "operation matrix"* (Cf. *Section 8*), we detect 14 bugs out of the 18 bugs detected by Johnson Controls and 4 bugs out of the 5 bugs detected by the carmaker. No new bugs have been detected. Comparing to the first case study, we were able through the *Driver Profile* mode to detect most of the known bugs. This could be explained by two facts:

- We cover 90% of the *input signals* domains in comparison with 70% in the first case study.
- According to experts, simulating random *operations* (*Nominal "operation matrices"*) on the *fuel gauge* functionality does not make real sense.

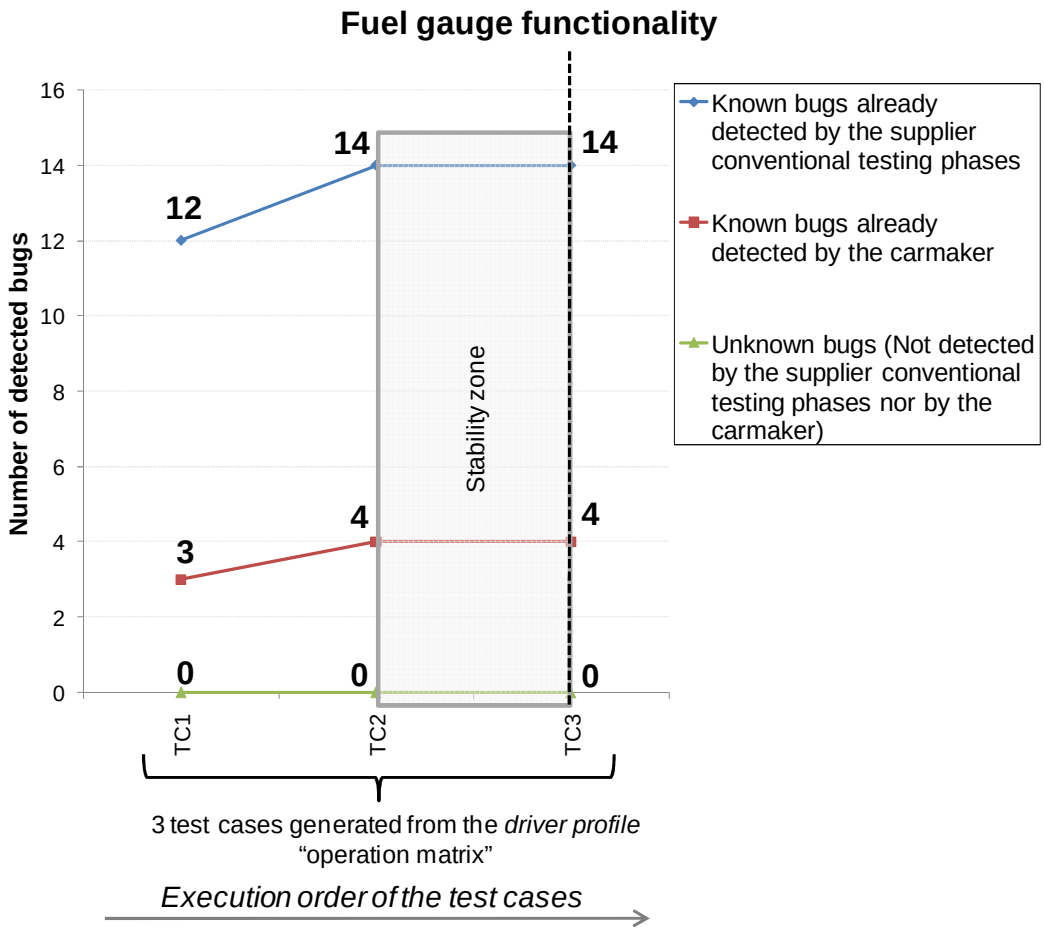


Figure 10.16 – Evolution of the cumulated number of bugs detected by our approach on the fuel gauge functionality

B. Decrease the time spent in testing a functionality

On the one hand, **we detect bugs earlier in the software life cycle**. *On the other hand*, **we lower the time spent in testing a functionality**. Thanks to historical data, the total time spent in testing conventionally the two functionalities is illustrated in *Figure 10.17*; e.g. 53.75 eight-hour days for the *front-wiper* and 50 days for the *fuel gauge*. For the *front wiper* functionality, no *unit test* has been performed. During the *validation test* stages, test engineers spent 11,5 *eight-hour days* analyzing the carmaker requirements before start designing manually *test cases* (29,5 e*ight-hour days*). 6 *eight-hour days* were spent executing the designed *test cases* and analyzing the results. Finally, we estimate at 6,75 *eight-hour days* the time spent in managing the bugs detected later in the process. For the *fuel gauge* functionality, 5 *eight-hour days* were spent testing unitarily the functionality. During the *validation test* stages, test engineers spent 10 *eight-hour days* analyzing the carmaker requirements before start designing manually *test cases* (22 *eight-hour days*). 6 *eight-hour days* were spent executing the designed *test cases* and analyzing the results. Finally, we estimate at 7 *eight-hour days* the time spent in managing the bugs detected later in the process.

As stated in *Section 2*, up to 50% and 10% of the total time spent in verifying and validating a software functionality were respectively spent to manually design the *test cases* and manage the bugs detected later in the process.

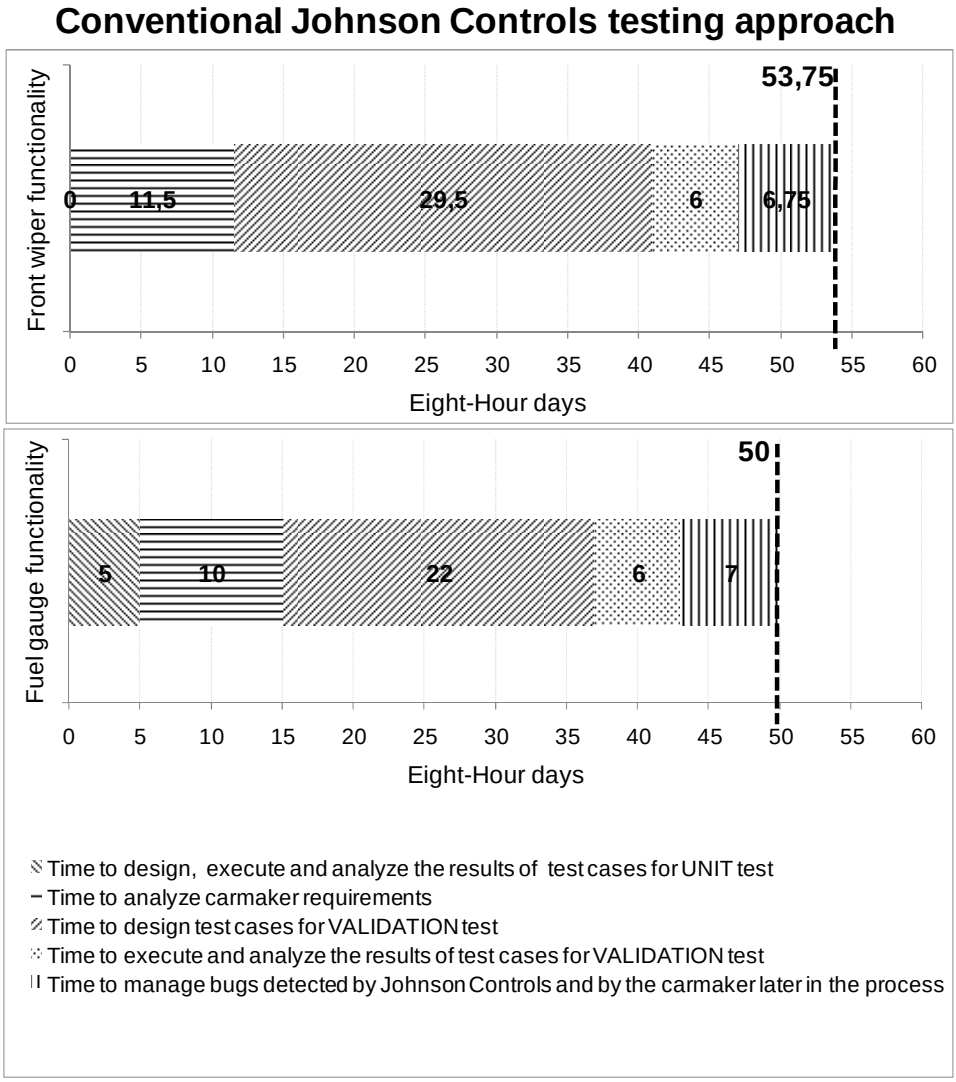


Figure 10.17 – An estimate of the total time spent in testing conventionally the two functionalities

The total time that we spent in testing unitarily the two functionalities using our approach is presented in *Figure 10.18*. It has been approximately spent 39 and 41,5 *eight-hour days* testing respectively the *front wiper* and *fuel gauge* functionalities. In *Section 4*, *5*, *6*, *7* and *9*, we estimate and comment the time spent analyzing the carmaker requirements, modeling, computerizing, verifying and validating the requirements model, designing the *"operation matrices"* and finally generating and executing automatically the *test cases*. After executing the generated *test cases*, we estimate to 10 and 2 *eight-hour days* the time respectively spent in analyzing the execution results. It consists, once an anomaly is detected, of answering the question: "Is it a bug in the requirements model or a bug in the software module?". The correction of anomalies is instantaneous. The time spent in analyzing the execution results is proportional to the number of executed test steps (*front wiper*: 68500 *test steps*, *fuel gauge*: 900 *test steps*). In fact, the task of manually designing the *test cases* disappears in favor of designing, verifying and validating the requirements model. Once the model is developed, the test design activity is automated but more efforts are necessary to analyze the results of the tests execution. Indeed, test engineers have to understand the generated *test cases* in order to confirm or not a bug. Moreover, we do not detect 3 and 5 *known bugs* respectively on the first and second case studies. In *Section 10.A*, we come up to the conclusion that these bugs could be detected by our platform since we reach a 100% of the functional *coverage* (now, it is not

the case, Cf. *Section 9*). Based on the assumption that our computational algorithm was improved (to be able to reach the 100% functional *coverage*), we estimate the time required to detect the remaining bugs on the two case studies. This time take into account the time to generate and execute the *test cases* and analyze the results. For the first case study, we already cover 90% of the requirements model and 3 bugs are remaining. Therefore, we estimate to 2 *eight-hour days* the time to detect these bugs. For the first case study, we already cover 70% of the requirements model and 5 bugs are remaining. Therefore, we estimate to 3 *eight-hour days* the time to detect these bugs. These estimations could be explained by the fact that:

- The requirements model of the first case study is bigger than the one of the second case study (Cf. *Table 10.6*).
- Analyzing the execution results of the second case study takes more time that the one of the first case study. In fact, the requirements model of the second case study (*Natural language*) is less reliable that the one of the first case study (Cf. *Section 3* and *6*).

Globally, we spent approximately 39 and 41,5 *eight-hour days* testing respectively the *front wiper* and *fuel gauge* functionalities. In this estimation, we do not consider the time spent in configuring our test platform using the *try-and-test* protocol (Cf. *Chapter 8.A*). In conclusion, we lower by 27% (39 instead of 53,75 *eight-hour days*) and 17% (41,5 instead of 50 *eight-hour days*) respectively the time spent in testing the *front wiper* and *fuel gauge* functionalities.

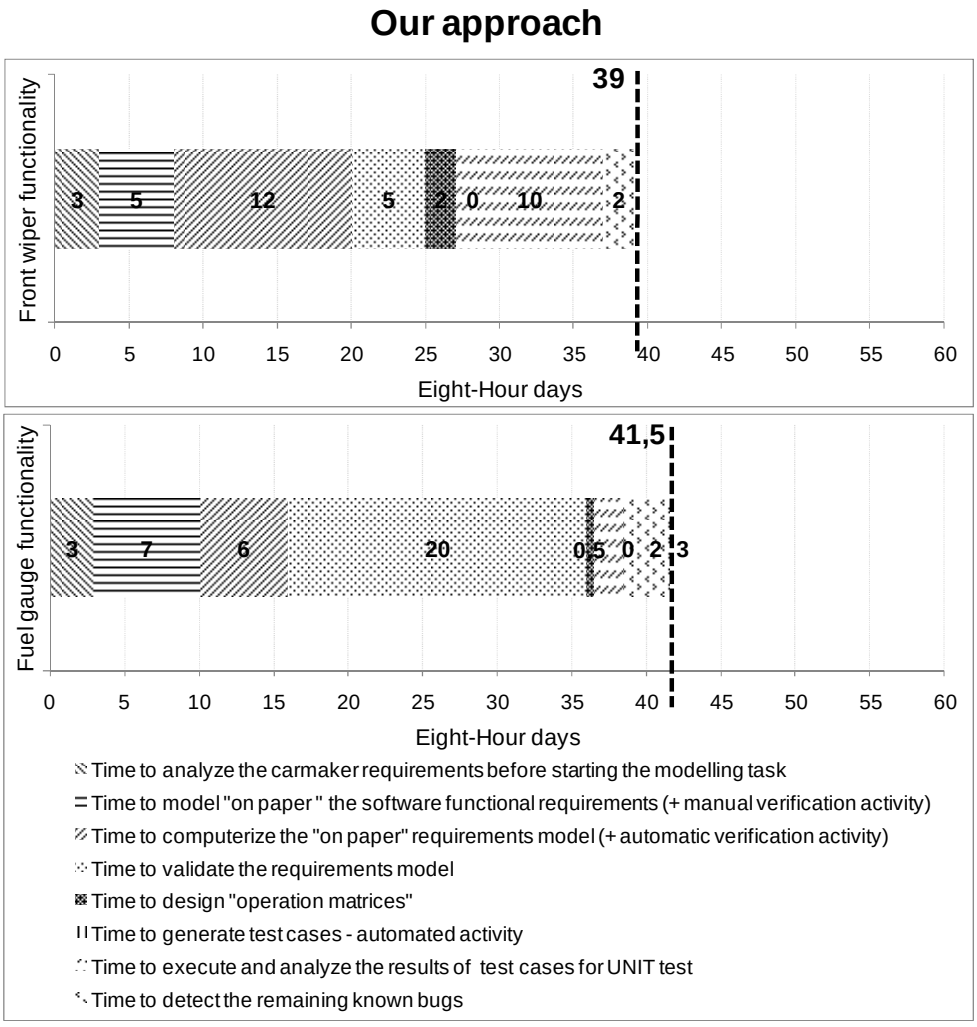


Figure 10.18 – An estimate of the total time spent in testing unitarily the two functionalities using our approach

After the first carmaker delivery and for each new delivery, we estimate that an average of 1 *eight-hour days* can be enough to review and update the *test cases* of the functionality under test. In fact, as carmaker requirements is suitable to evolve along the different deliveries (Cf. *Chapter 2 – Section 4.A*), it will be easier to test engineers to update the requirements model and generate automatically a new set of *test cases* than to update manually the design of *test cases*.

C. Quantitative results' overview: earlier detection of bugs and time saving

Performing a *functional unit test*, for each functionality (software module), using our approach to generate *test cases* automatically leads to **notably improved results**. In *Table 10.12*, we summarize the results of the two case studies in terms of detecting bugs earlier in the software life cycle.

	Front wiper functionality	Fuel gauge functionality
Increase the number of bugs detected since the first testing phase	100% (from 12 to 24)	800% (from 2 to 18)
Decrease the number of bugs detected by the carmaker	60% (from 5 to 2)	80% (from 5 to 1)
Increase the number of bugs detected by Johnson Controls	41% (from 17 to 24)	22% (from 18 to 22)
New bugs detected	18% (5 out of 27)	0% (0 out of 23)

Table 10.12 – A summary of the results of the two case studies

Moreover, we lower by 27% and 17% respectively the time spent in testing the *front wiper* and *fuel gauge* functionalities (Cf. *Figure 10.19*).

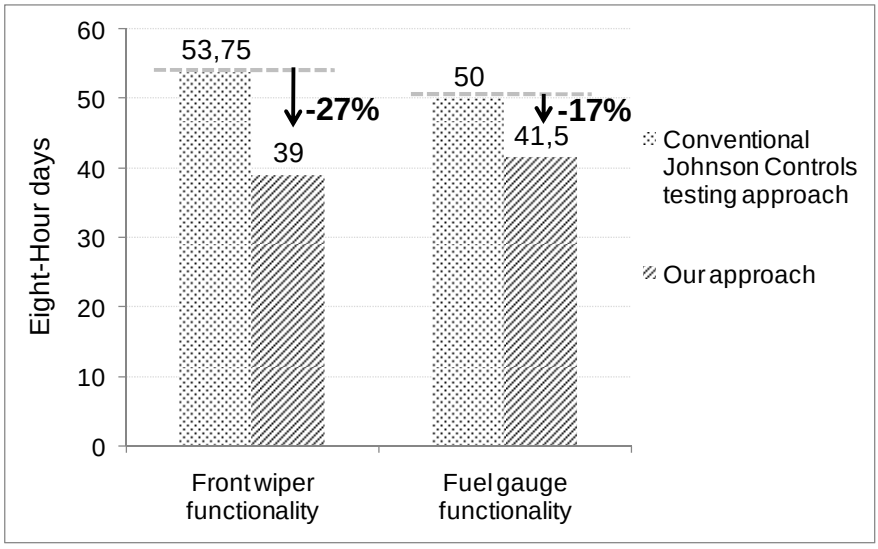


Figure 10.19 – Reducing the time spent in testing the two functionalities

XI. Conclusion

In this chapter, we have experimented our new *testing methodology* through two typical case studies on historical data. Potential benefits (*quantitative* and *qualitative*) have been quantified. We reduce by 70% the number of bugs detected by the carmakers and by 9% the ones detected by the end-users. Moreover, we reduce by 22% the time spent in testing a software product. We also propose to deliver to the carmaker *formal* quality indicators (*coverage*) on the delivered software. All these results contribute to an improvement of the

customer satisfaction and as a direct impact; the number of tenders will grow. Unfortunately, estimating the cost of software bugs in an organization is a delicate, strategic and confidential question and therefore we were not allowed to communicate the numbers on the bugs' costs savings via the use of our approach.

In the following chapter, we give an overview on the contributions, impacts and perspectives of our approach.

GENERAL CONCLUSION

I. Contributions' review

In this research project, we were asked by an automotive electronic supplier, namely Johnson Controls, to improve the performance of its software *V&V* activities. Their main purpose is to improve the quality of their products and therefore better satisfy the requirements and expectations of their clients. We went through this problem with a *systemic approach* in order to identify levers in any domains from which we might be able to improve the *global performance* of the software *V&V* activities. The major added value of the present work is to globally solve the quality issue of the *software testing* process. Hereafter, we summarize the main ten contributions of our research:

Contribution 1: *A list of anomalies and lacks in the software verification and validation (V&V) practices in automotive industry.*
Through an *industrial audit*, we analyze the current software practices in automotive industry. The audit is divided into four parts: 1) the process of managing the carmakers' requirements related to the software domain, 2) the processes of verifying and validating software products, 3) the process of managing and reusing capitalized bugs and finally 4) the process of managing and reusing capitalized *test cases*. For each of these parts, we make our analysis in two stages: 1) a snapshot of the current software practices in automotive industry (*process*, *tool*, *people*) and 2) an analysis and identification of issues and lacks (diagnoses) in these practices. Our approach to perform the audit can be divided into 7 activities: 1) analyze the documents delivered by the carmakers to their electronic suppliers, 2) analyze the main activities of an engineer when designing *test cases* for a software product, 3) audit engineers when designing *test cases*, 4) intervention on the design of *test cases* for four software projects, 5) interview managers on the expectations of the carmakers at each stage of the software development life cycle, 6) interview all types of engineers that can be involved in a software project and finally 7) analyze data on the *software testing* practices of carmakers. The result of the audit is a list of anomalies and lacks (diagnoses) in the current software *V&V* activities in automotive industry.

Contribution 2: *A formal specification language to represent and simulate software functional requirements in automotive industry*
Managing the software functional requirements is considered as one of the key issues in the software development process. In fact, these requirements are the main input for the design and implementation processes of the software product but also for the *verification and validation* processes. Ten years ago, *formal* methods were rarely used in automotive industry, contrarily to medical, avionics and railways industries. Now, in automotive industry, *semi-formal* and *formal* methods are more and more used to specify software functional requirements. However, there is a lack of a standard *formal*ism shared between carmakers and suppliers. In fact, for each project, the supplier must adapt its processes to the *formal*ism used by the carmaker. In this context, we develop a new *formal* and *simulation* language to model software functional requirements. A *simulation model* of these requirements can help to avoid ambiguity, incompleteness and inconsistency in customers' requirements. Development and validation teams can communicate more easily with the customer and fix specification's problems. Moreover, through a *simulation model*, one can automate the assessment process of all the expected outputs values of a software product. In fact, when designing test cases, test engineers can perform the selected *operation* on the *requirements model* and automatically assess the expected output values by simulating the model. We then name "*operation*" the fact that an *input signal* of the software product is set to a given value. Finally, one can now *formally* measure the *coverage* of the requirements model, which bring new valuable quality

indicators in addition of the sole *code coverage* for better monitoring the *software testing* process.

Contribution 3: *An automatic process to design test cases for software products*
In industry, the activity of manually designing *test cases* for software products becomes more and more laborious and time consuming. Despite the considerable time and money spent in testing a software product and after each delivery to the customer, some bugs are still detected by the customer. Since the late 90's, the automation of the *test case design process* has become a hot topic and industrials are still looking for a relevant automation of this process. In this context, we develop a strategy to automatically design *test cases* with simulations from our *formal* model. A *test case* is a *series of operations* whose selection is performed based on a *Monte Carlo simulation* on an "*operation matrix*". Probabilities are expressed for choosing a next *operation* and for defining the time interval between both successive *operations*. Therefore, we build a matrix that we name "*operation matrix*" with all possible *operations* in columns and in rows; this "*operation matrix"* becomes central to our *test case generation* algorithm. All along the *test case generation*, the expected values on the *output signals* of the functionality are assessed through a simulation of the requirements model.

Contribution 4: *An objective function for optimizing the design of test cases for software products*
Testing software exhaustively remains a major problem from the computing point of view. Therefore, *software testing* must often be based on specific assumptions and objectives which help test engineers and managers to decide when to stop the testing protocol. In order to monitor our automatic design of *test cases*, we propose an *objective function* based on a *formal structural* (software code) and *functional* (customer requirement specification) *coverage* and the execution time and cost of designed *test cases*. In software engineering, the term "*coverage*" means the degree, expressed as a percentage, to which a specified item (code or requirement) has been exercised by a *test case*. In addition, we define an exponential set of weights that test engineers can associate for each defined *coverage*, *time* or *cost* target: 0 (to be ignored), 1 (not very important), 5 (important), 10 (very important).

Contribution 5: *A hybrid heuristic algorithm for optimizing the design of test cases for software products*
When testing a software product, test engineers have to execute the designed *test cases* on the software under test. The execution could be *manual* or *automatic* and is often time and resource consuming. The main purpose of a test engineer is to detect the maximum number of bugs in minimum laps of time. Therefore, optimizing the number and length of *test cases* while fulfilling predefined objectives and constraints is critical to reach the quality, schedule and cost goals of a software project. To overcome this problem, we propose a heuristic algorithm in charge of optimizing the design of *test cases* while fulfilling quality objectives and time constraints. In this algorithm, we implement two types of optimization strategies: *Look Back* and *Look Ahead*. In fact, when designing *test cases*, we avoid similar and repetitive operations or successions of operations (*Look Back*) and we focus on the ones which improve the objective fulfillment (*Look Ahead*).

Contribution 6: *A software bug classification model and a detailed typology of software problems*
Each software organization uses a *problems' tracking tool* in order to manage and store problems detected during a software project. Moreover, the *tracking tool* has a database where all the problems are stored. Such databases hold thousands of software bugs and are difficult to be analyzed. In fact, when describing a bug in the problems' tracking tool, there are often

too many fields to fill in, a lot of free fields and a lack of relevant predefined fields. Moreover and as the detection of bugs comes later in the process, engineers do not have enough time to fill in all the fields of a bug. Therefore, analyzing these databases in order to pinpoint issues in the development processes and propose improvement actions is a complicated task. In this context, we propose a new *bug classification model*. The aim of this model is to be able to identify process improvement actions for the development and *V&V* processes. In other words, the new *bug classification model* answers the question of "which types of software problems are injected and detected in which process phase?" To do this, we propose a detailed software problem typology taking the industrial context into account. In addition, identifying recurrent type of software problems allows test engineers to focus the design of *test cases* on detecting these problems.

Contribution 7: *A process to define software users' profiles in order to design test cases that simulate the real use of a software product*

There is no better way to test a product other than testing it in the way that it will be used. The major number of bugs detected by the *end-users* of a software product is related to specific *operations* or successions of *operations* recurrently performed on the software in real use. Therefore, testing a software product with an *end-user* point of view seems beneficial. We propose to define an *end-user behavior's profile* for each software under test. This profile can be used by test engineers when designing *test cases*. In fact, we define four types of *constraints* that test engineers can affect to each *input signal* of a software product in order to eliminate or favor specific successive *operations*. Each *input signal* can have one or more *constraints*. These *constraints* aim to lower the number of possible combinations on *input signals* and to more thoroughly pinpoint which ones are frequently set once the product is launched on the market. These four *constraints* are: *logical constraint, conditional constraint, succession constraint* and *timing constraint*.

Contribution 8: *An automatic and formal process to use capitalized software bugs in the design of test cases suitable to detect similar bugs on a new software development*

Only exhaustive testing can show that a software product is free from bugs. However, exhaustive testing of a software product is not practical because variable input values and variable sequencing of inputs result in too many possible combinations to test. So it is useful to concentrate the test on the areas associated with the greatest risks and priorities. In this context, we propose to design *test cases* which have a high probability to detect software bugs. Therefore, we specify a new format for the "*Problem description*" attribute of a bug capitalized in the problems' database. This format consists of describing the initial conditions and the successive *operations* that lead to the capitalized bug. Based on this new format, we propose an automated process able to design one or more *test cases* from each capitalized bug. These *test cases* are suitable to detect bugs recurrently done by test engineers on specific software functionalities.

Contribution 9: *An automatic and formal process to reuse capitalized test cases for one project to another*

Reusing capitalized *test cases* from one project to another seems to be beneficial in an industrial context. In other words, when *testing* a software functionality that has already been implemented in the past on another project, it is judicious to reuse existing *test cases*. But unfortunately, *test cases* are not often reused from one project to another. Two potential main reasons are: 1) the use of different formats when designing manually *test cases*. Sometimes, test engineers write the *test cases* immediately in a computer language (*C language* …) understandable by the *test execution platform*. Others use a more high level language. 2) the lack of an automated process to reuse the *test cases*. To overcome this problem, we propose to

use one specific format as the standard format to represent a *test case*. Based on this new format, we develop an automated process able to design one or more *test cases* from each capitalized *test case*. In fact, the designed *test cases* focus on test scenarios based on the returns of experience from previous projects.

Contribution 10: *Promising results of the experiment of our testing methodology on two typical case studies within an automotive electronic supplier*
Through our research project, we propose *a new systemic approach to automate efficiently the design of test cases for software products*. Apart from the computational aspects of software testing, the approach takes into account organizational matters (Cf. *Contributions* 2, *3*, *4*, *5*, *6*, *7*, *8* and *9*) such as *functional simulation*, *knowledge management*, *competency management* and *project management*. Our *testing methodology* has been implemented in a computer platform and experimented on two typical case studies of Johnson Controls for which historical data are available. Consequently, we *reduce by 70%* of the number of bugs detected by the carmakers and *by 9%* the ones detected by the end-users. Moreover, we *reduce by 22%* the time spent in testing a software product. In fact, we detect the bugs *earlier* in the software development process and *closer to their origin*. We also propose to deliver to the carmaker *formal quality indicators* on the delivered software. All these results contribute to an improvement of the *customer satisfaction* and as a direct impact; the number of tenders will grow. Unfortunately, estimating the cost of software bugs in an organization is a delicate, strategic and confidential question and therefore we have not been allowed to communicate the numbers on the bugs' costs savings via the use of our *methodology*.

Contribution 11: *A patent on our approach to design test cases for software products*
The promising results of the deployment of our *testing methodology* within the industrial context have motivated the automotive electronic supplier Johnson Controls (who grants this *PhD*) to *patent this approach*. Presently, the company patent experts are assessing the economical profit of patenting our approach. In the meantime, a *worldwide Quick Patent*[42] (for a preliminary protection of the idea) has been submitted by the company.

II. Impact of our testing methodology in the company organization

Estimating the cost of bugs in a software organization is a delicate, strategic and confidential question. In 2002, the *National Institute of Standards and Technology* (*NIST*) has estimated that software bugs cost U.S. economy 59,5 billion dollars annually[43]. In Johnson Controls, there is no model to estimate the cost of software bugs. Unfortunately, these data are confidential. However, the number of software bugs detected by the carmakers during intermediate deliveries or by the *end-users* after the *Start Of Production* (*SOP*) is estimated each month. As the automotive market becomes more and more competing, decreasing the development time of outsourced parts and decreasing the number of problems detected later in the process becomes of major importance for carmakers and consequently a major quality indicator for automotive suppliers. Indeed, the carmakers' process for assigning new projects to suppliers is mainly based on feedbacks from previous projects. Through our *testing methodology* (Cf. *Table 10.12*), we **reduce by 70%** ((60+80)/2, 60% and 80% respectively on the first and second case studies) the number of software bugs detected by the carmakers after intermediate deliveries. Making the assumption that the new "minor" bugs that we detect

[42] In France, we associate a *Quick Patent* to an "Enveloppe Soleau" (http://www.inpi.fr/fr/services-et-prestations/enveloppe-soleau.html, Consulted on November 2008).
[43] http://www.nist.gov/public_affairs/releases/n02-10.htm (Consulted on November 2008)

through our *methodology* and which were neither detected by Johnson Controls nor by the carmaker have been detected by an *end-user*, we can state that we **reduce by 9%** ((18+0)/2, 18% and 0% respectively on the first and second case studies) the number of software bugs detected by the *end-users* once the product is launched on the market. Moreover, we propose to deliver to the carmaker **quality indicators** related to **code coverage** (already done in the industry) but also **formal requirements coverage**, which may increase its confidence about the quality of the software products. Presently, the measurement of *requirements coverage* is *informal* (Cf. *Chapter 2 – Section 6.B.1*). In conclusion, across our *testing methodology*, the **image of the company** (Johnson Controls) in front of its customers (carmakers) will be improved and as a direct impact of the **customer satisfaction**, the number of tenders will grow.

Moreover, the *validation test* stage accounts for more than 50% of the project time and resources (Cf. *Chapter 1 – Section 5.C.2*). In fact, bugs related to the internal behavior of one software module could be detected in *unit test* stage **(earlier in the process)**. Unfortunately, it is not the case and such bugs are detected later in the *validation test* stage. Of course, analyzing the origin of a bug in *validation test* stage (all the software modules are integrated together) is more difficult and time consuming than analyzing the bug's origin in a specific software module. Through our *testing methodology* (Cf. *Figure 10.20*), we **reduce by 22%** ((27+17)/2, 27% and 17% respectively on the first and second case studies) the time spent in testing a functionality. While lowering the number of bugs detected by carmakers and end-users, we **lower the resources required for testing a software product**.

However, we are conscious of the **impact of our testing methodology (model and design platform) on the current software organization in case of an industrial deployment**. Indeed, an **investment** but also **personal commitments** of all the software players within the company are mandatory for the success of such change of practices. In *Chapter 9*, we develop a "*process-people-tool*" view of our *testing methodology*. Based on this view, we identify three streams of actions necessary for integrating our methodology within the current software organization of the company:

- Integrate the **processes** of our methodology (Cf. *Figure 9.2*) within the global software process map of the company (Cf. *Figure 2.2*).
- Train the software **engineers** to the new *testing methodology*. *Test automation* has broad impacts on an organization such as the skills needed to design and implement automated tests, automation tools, and automation environments. The test engineers' practices, roles and competencies change when automation is installed. These impacts have negative aspects that must be considered. When introducing a new methodology and tool to the testing program, mentors and trainings are very important. Even with training, automation skills take time and experience to acquire. The best automation tool in the world will not help the test efforts if the test team resists using it. The test engineers may feel that 1) their manual process works fine, and they don't want to bother with the additional setup work for introducing an automation tool and 2) they may lose their *know how* in designing manually test cases for software products. Indeed, test engineers' technical skills will have to switch from a manual design to a high level modeling of the test scenarios and objectives in using in a flexible manner our design platform. Nevertheless, based on the literature (Bunse 2007), model-based software development approaches are slowly superseding traditional ways of developing software products and software engineers' required skills tend toward modeling and automation tool monitoring.
- Improve the *Man Machine Interfaces* of the computer **tools** that we developed to support our *testing methodology* (Cf. *Chapter 8 – Section 3.C and 4*). In fact,

ergonomic user interfaces play a major role in the practitioners' use and perception of a computer tool.

III. Research perspectives

The open perspectives of this research project are listed by topic.

Perspective 1: *Related to the formal language to specify software functional requirements*
The perspectives concerning our formal language to specify software functional requirements are:

- Perform a broad survey on the carmakers' specification of the software functional requirements. The purpose is to fill out our *formal* specification language in order to be able to specify any carmakers' software functional requirement.
- Develop a list of *rules* and *recommendations* to help *modelers* using efficiently our specification language and therefore develop consistent requirements model at the first attempt.
- Develop more efficient strategies to validate the compliance of a requirements model developed using our specification language with the (original) carmaker requirements. One solution could be to validate the model by the carmaker itself.
- Develop an *editor tool* to support *modelers* in designing a requirements model using our specification language. For instance, when designing a *DT element*, designers can not consider all the possible *conditions* on the *input signals*. In fact, in an industrial context, the number of the *DT input signals* can exceed 10 and the domain length of one signal can exceed 100 (for instance, when sampling the "vehicle speed" signal). In that case, its remains a very difficult task to identify manually all the possible *conditions* and their corresponding *actions*. Therefore, an automatic generation of all the possible *conditions* on the *input signals* of a *DT* could be judicious. The *editor tool* could perform such functionality.

Perspective 2: *Related to the knowledge management in terms of capitalized bugs and test cases*
On the one hand, we propose to reuse capitalized bugs in order to verify the nonexistence of recurrent bugs. To do this, we develop a new *bug classification model* with a detailed typology of software problems and a specific format to describe the initial conditions and the successive *operations* that lead to detect a bug. We propose to generate automatically *test cases* that verify the nonexistence of recurrent (capitalized) bugs on each software functionality (for instance, *front wiper*) of a new development. To do this, for each software functionality of a product family, a glossary of the functionality's *input signals* names on previous and new projects are necessary. A family of product is defined by a *customer* (for instance, *Renault*), a *type of product* (for instance, a *body controller module*) and a *car platform* (for instance, *Laguna platform*). We experiment these proposals on two industrial case studies with historical data. However, it could be judicious to experiment our *bug classification model* (software problems typology and description formalism of a bug) and the inputs glossary on new software projects. Therefore, we could adjust our proposals in order to take practical considerations into account.
On the other hand, we propose to reuse *test cases* from one project to another. To do this, we define a new formalism to represent a *test case* and based on this formalism, we develop an automatic process to generate one or more *test cases* that focus on *operations* or successions of *operations* regularly done in a capitalized *test case*. In fact, we propose to reuse *test cases* when testing a software functionality that we already tested in the past. Therefore, a *test cases* library should be specified in order to capitalize the *test cases* by software functionality and

family of product. Moreover, for each software functionality of a product family, a glossary of the functionality's *input signals* names on previous and new projects are necessary.

Perspective 3: *Related to the test case generation algorithm*

Through our experiment, we show that our computational algorithm does not successfully reach 100% of functional *coverage* (the maximum was 90%). Consequently, we were not able to detect bugs related to the non-covered functional requirements. To overcome this deficiency, we plan to develop a new *test case generation* algorithm that focuses on covering non-covered zones of a requirements model. In fact and instead of selecting *operations* via a *Monte Carlo simulation* on the *input signals* of a model, we propose to synthesize the *operations* that lead in covering a specific item (for instance, a *state* of an *FSM*, a *condition* of a *DT* …) of the model. In other words, one has to select the item that should be covered and the algorithm will propose a list of successive *operations* to be performed on the model in order to cover this item. We already start a global design of this algorithm but unfortunately, we had not enough time to implement it in our approach.

Perspective 4: *Related to the strategy for tuning the generation of test cases*

We are conscious of the variability or subjectivity of our current strategy (*try-and-test*) to set *coverage* objectives and optimization parameters when generating *test cases*. In fact, there are a lot of parameters to set. As a consequence, we plan to propose a new strategy to help test engineers to parameterize the generation of *test cases*. In fact, when testing a software product, the main purpose of a test engineer is to detect the maximum number of bugs in minimum laps of time. Therefore, we have to identify the *correlations* between the optimization algorithm parameters, the functional *coverage*, the execution time of the generated *test cases* and the number and type of detected bugs. Based on these correlations, we might define *rules* and *recommendations* to help test engineers parameterizing the generation of *test cases*. Moreover, we plan to develop *parameterization profiles* that test engineers could adopt according to the test stage objectives. A *parameterization profile* consists of a set of predefined optimization parameters, coverage objectives and time constraints. To do this, we plan to perform a *Design of Experiments* (*DoE*) on our approach (Cf. *Figure Conclusion.1).* We set all the functional *coverage* objectives to 100% with no time or cost constraints. We decide to generate *test cases* for only one *"Configuration"* of the functionality under test (*Parameter 6* = 0, *Parameter 7* not to be defined). We plan to generate one *test case* for each combination of the parameters (*Parameter 8* = 1). The five remaining parameters of the optimization algorithm (*Parameter 1*, *2*, *3*, *4* and *5*) represent the factors of the *DoE*. Two factors (*Parameter 1* and *3*) have two levels (0, 1) and three factors (*Parameter 2*, *4* and *5*) have *n* levels (*n* integer). Based on our experience, we sample the domain of these factors into four levels (*30, 60, 90 and 120*). Consequently, a *complete DoE* accounts for 256 combinations and a *partial one* for 16 combinations. We decide to perform the partial *DoE*. After generating one or more *test cases* for each combination, we have to measure the reached functional *coverage*. And after executing independently each *test case* on the software module under test, we have to assess the number and type of detected bugs and the time spent to execute the *test case*. Once all the combinations of the *DoE* are achieved, the experiment results must be analyzed and correlations identified. In fact, we expect that the results of the *DoE* can help test engineers to configure the test platform in a short time (around 1 *hour*) instead of 1 *eight-hour day* using a *try-and-test* strategy (Cf. *Chapter 10 – Section 8*). We start performing the *DoE* but unfortunately, we had not enough time to complete it.

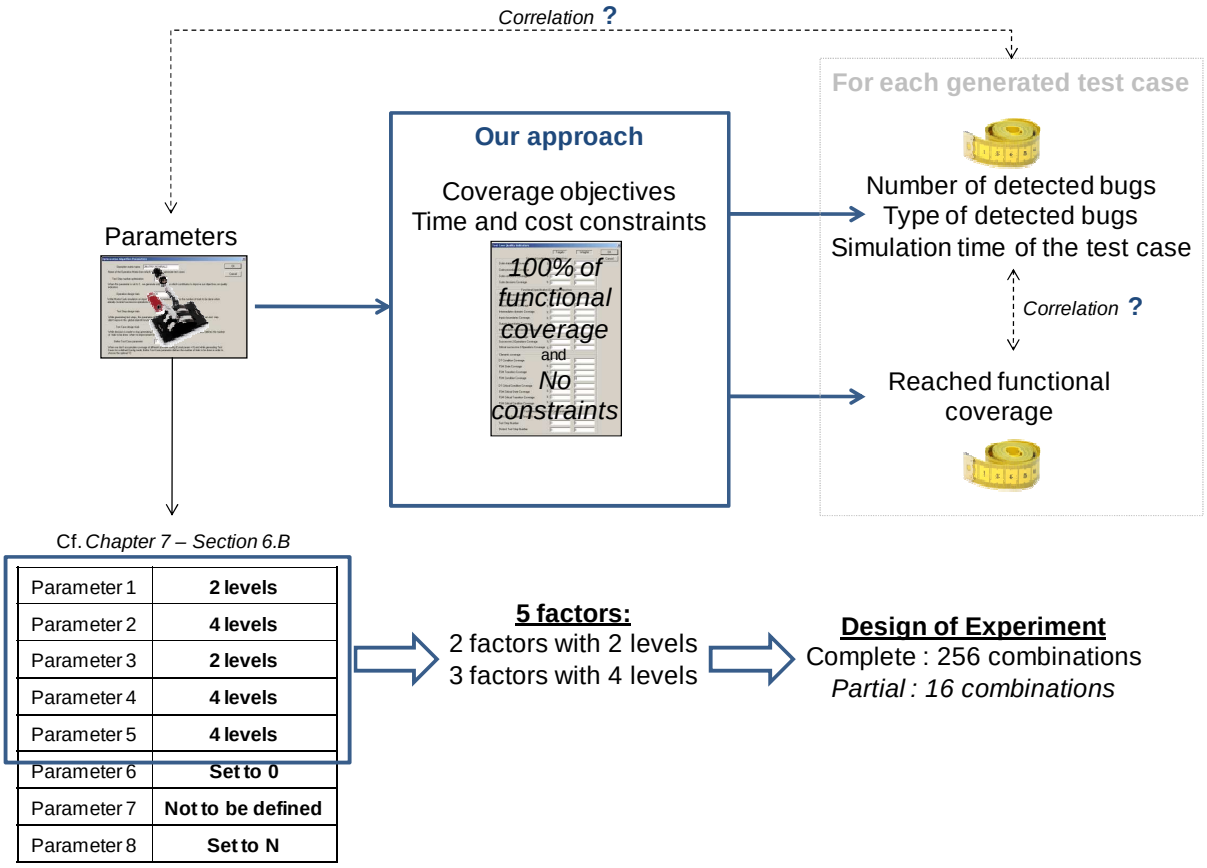


Figure Conclusion.1 – A Design of Experiments to identify the correlations between the parameters of our approach and the detection of bugs

Perspective 5: *Related to the consistency and reliability of our experiment's results*

Our approach to design *test cases* for software products can be identified to a *measurement system* which has to measure the number of bugs in a software product. As a consequence, we have to check the statistical properties of a reliable *measurement system*: *repeatability* and *reproducibility*. We start performing this task but unfortunately, we had not enough time to complete the experiments.

- *Reproducibility:* In our *testing methodology*, the two main activities depend on the *operator* (e.g. human intervention). The first one is the design of the requirements model and the second one is the definition of a set of targets and weights for the *test case generation*. Therefore, the *reproducibility* of our experiment results must be verified. In fact, two *operators* must independently model the same carmaker requirements. *Rules* and *recommendations* have to be defined in order to help *operators* configure the generation of *test cases*. Each *operator* has to generate automatically a set of *N test cases* fulfilling the predefined targets. After executing independently each set of *N test cases*, one has to assess the ratio of bugs simultaneously detected by the two sets of *test cases*.
- *Repeatability:* Since our generation of *test cases* is partly based on a *stochastic process*, the *repeatability* must be verified. Consequently, we propose to generate two or more sets of *N test cases* from the same requirements model and with the same objectives, constraints and optimization parameters. After executing independently each set of *N test cases*, one has to assess the ratio of bugs simultaneously detected by two or more sets of *test cases*.

Perspective 6: *Related to the monitoring of our test case design process*
We also plan to monitor the quality of our new testing process. To do so, it seems that within the *Design for Six Sigma* (*DFSS*) framework, the *Define*, *Measure*, *Analyze*, *Design*, *Optimize*, and *Verify* (*DMADOV*) methodology is the appropriate approach. This will allow us to put the proper focus on the up front design of the testing process. Therefore, we need to establish the set of measurable, customer-oriented attributes, which can be defined, measured, analyzed, optimized and verified (*DMADOV*) in the *software testing* process. These attributes need to be directly built into the testing process so that it is specifically geared to producing pre-defined quality limits. This means embedding specific design intent within the *software testing* algorithm to meet specific and understood, customer-facing performance metrics. Below, we identify two types of *critical-to-customer* metrics concerning the *software testing* process. We plan to assess the following metrics on each software project that undergoes testing:

- *Critical-to-Quality* (*CTQ*) metrics:
  Y1. The capacity to reduce the number of bugs detected by the carmaker: the ratio between the number of bugs detected by carmakers and the total number of bugs
  Y2. The capacity to reduce the number of bugs detected by the *end-user*: the ratio between the number of bugs detected by the *end-users* and the total number of bugs
- *Critical-to-delivery* (*CTD*) metrics:
  Y3. The number of versions of each software module or product
  Y4. The capacity to deliver software free of bugs since the first delivery: the ratio between the number of bugs detected in the first testing phase and the total number of bugs

Since we place a high premium on reducing the number of bugs detected by carmakers and *end-users* (*Y1* and *Y2*), one solution could be to increase the structural and functional *coverage*. But, experiments reveal that some bugs cannot be detected even if our requirements model and *source code* are covered at 100%. This leads to the realization that we need to refine our functional *coverage* model. Typically, we can consider the *coverage rate* of the succession of two *transitions* in a *FSM* element.

IV. General discussions

In this section, we discuss three major topics related to the *deployment* and the *durability* of our *testing methodology* within an industrial context.

Since 2003, carmakers, suppliers and other companies from the electronics, semiconductor and software industry have been working on the development and introduction of an open, standardized software architecture for the automotive industry (*AUTOSAR - AUTomotive Open System ARchitecture*). One of the key features of this consortium is the *modularity* and *configurability* of automotive software products. This leads to increase the reuse of software components from one project to another. As a consequence, reused software components would reach a high reliability degree and do not require to be tested unitarily after each reuse. *Integration* and *validation test* will be of major importance. Nevertheless, the *unit test* of software components will remain necessary since 1) the reused software components represent around 50% of the total components of an automotive software product and 2) these reused component will evolve continuously (new functionalities and features) and therefore need to be tested unitarily.

Presently, many researches and industrial projects deal with the automatic generation of the *source code* of a software product. The main purposes of these actions are to 1) reduce the

software development time and 2) avoid some software problems injected by the software engineer when designing and coding the software product. As for the automatic generation of *test cases*, a formal representation of the software specification is required. Most of the *formal* specification languages found in the literature attempt to be useful for the *code* and *test case generation*. Therefore, it could be useful to explore the automatic generation of *source code* from our functional requirements model of a software product. Considering the following two assumptions 1) the requirements model is validated at 100% and 2) the generation of the source code is reliable at 100%, the generated *source code* of a software product does not need to be tested. Unfortunately, it is not the case and a software product needs always to be tested (verified and validated).

Although our *testing methodology* has been customized to software embedded in cars (carmaker requirements formalisms, automotive constraints …), the use of this approach in industries such as aeronautic, railway, medical, telecommunication … seems beneficial. In these industries, software products properties and architectures are similar to the automotive industry. However, software requirements formalisms and priorities in testing software products could be different. For instance, contrarily to automotive industry, in aeronautic industry, constraints on software project planning and budget are less important than software quality objectives. This could be explained by the fact that avionics software requires being highly reliable, since failures in this kind of products may very likely lead to deathly consequences. One more point is the applicability or adaptability of our *testing methodology* to *computers applications*; for instance, testing software products such as the *Microsoft Word* software.

BIBLIOGRAPHY

My publications

International Conferences

(Yannou 2005) Yannou B., *Awedikian R.*, 2005, "A Plug-And-Contract Mechanism for a Robust Assessment of Design Concepts." Proceedings of the ASME Design Engineering Technical Conferences / Design Automation Conference - DETC/DAC 2005, Long Beach, CA, USA, DETC2005/85457.

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

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

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

(Awedikian 2008c) *Awedikian R.*, Yannou B., 2008, "An objective function for optimizing the generation of test cases for automotive software product." Proceedings of IDMME - Virtual Concept 2008 - IDMME 2008, Beijing, China.

(Yannou 2008) Yannou B., Dihlmann M., *Awedikian R.*, 2008, "Evolutive design of car silhouettes." Proceedings of the ASME Design Engineering Technical Conferences / Design Automation Conference - DETC/DAC 2008, New York City, NY, USA, DETC2008/49439.

International Journals

(Awedikian 2009a) *Awedikian R.*, Yannou B., 2009. "A formal language to simulate the software functional requirements in automotive industry." Submitted on January 2009 in the "Computers In Industry" Journal.

(Awedikian 2009b) *Awedikian R.*, Yannou B., 2009. "Automatic generation of relevant test cases: A practical model-based testing approach." Submitted on January 2009 in the "Software Testing, Verification and Reliability" Journal.

(Adrion 1986) Adrion, W. R., M. A. Branstad, and J. C. Cheriavsky, 1986, "Validation, Verification and Testing of Computer Software." In Software Validation, Verification, Testing, and Documentation, S. J. Andriole, ed. Princeton, N. J.: Petrocelli, 81-123.

(Apfelbaum 1997) Larry Apfelbaum and J. Doyle, 1997, "Model-based testing." Proceedings of the 10th International Software Quality Week (QW 97).

(Apostolov 2007) Zhivko Apostolov, 2007, "Types of software tests." Internal document, Johnson Controls.

(Ayewah 2008) N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, 2008. "Experiences using static analysis to find bugs." Software, IEEE, 25(5).

(Balci 1984) Balci, O. and R. G. Sargent, 1984, "A Bibliography on the Credibility Assessment and Validation of Simulation and Mathematical Models." Simuletter, 15, 3, pp. 15–27.

(Balci 1989) Balci, O., 1989, "How to Assess the Acceptability and Credibility of Simulation Results." Winter Simulation Conference 1989, pp. 62–71.

(Balci 1997) Osman Balci, 1997, "Verification, Validation and Accreditation of Simulation Models." Winter Simulation Conference 1997: 135-141.

(Barezi 2006) Luciano Baresi, Mauro Pezzè, 2006, "An Introduction to Software Testing." Electronic. Notes Theoretical. Computer Science 148(1): 89-111.

(Basanieri 2002) Basanieri, F., Bertolino, A., Marchetti, E., 2002, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects." Proceedings 5th International Conference UML 2002, Dresden, Germany. LNCS 2460 383-397.

(Beizer 1984) Boris Beizer, 1984, "Software System Testing and Quality Assurance." Van Nostrand Reinhold.

(Beizer 1990) B. Beizer, 1990, "Software Testing Techniques." Van Nostrand Reinhold, 2nd edition.

(Beizer 1995) Boris Beizer, 1995. "Black-Box Testing: Techniques for Functional Testing of Software and Systems." John Wiley & Sons, ISBN 0471120944.

(Bernot 1991) Bernot, G., Gaudel M. C., Marre, B., 1991, "Software Testing Based On Formal Specifications: a Theory and a Tool." Software Engineering Journal 6 387-405.

(Bertolino 1997) Bertolino, A., Marré, M., 1997, "A General Path Generation Algorithm for Coverage Testing." Proceedings 10th International Software Quality Week, San Francisco, Ca. paper 2T1.

(Bertolino 2002) Bertolino, A., Polini, A., 2002, "Re-thinking the Development Process of Component-Based Software." ECBS 2002 Workshop on CBSE, Lund, Sweden.

(Bertolino 2003) Bertolino, A., 2003, "Software Testing Research and Practice", Invited presentation at 10th International Workshop on Abstract State Machines ASM 2003, Taormina, Italy, March 3-7, 2003, LNCS 2589, p. 1-21.

(Bertolino 2007) Bertolino, A., 2007, "Software Testing Research: Achievements, Challenges, Dreams," fose, pp.85-103, Future of Software Engineering (FOSE '07).

(Binder 1995) Robert V. Binder, 1995, "Object-oriented testing: Myth and reality." Object magazine.

(Black 2000) Rex Black, 2000, “Shoestring Manual Testing.” In 6th International Conference on Practical Software Quality Techniques (PSQT'00 South).

(Bontron 2005) Pierre BONTRON, 2005, “Les schémas de test : une abstraction pour la génération de tests de conformité et pour la mesure de couverture.” PhD dissertation, Joseph-Fourier University, Grenoble, France.

(Branaghan 1999) Branaghan, R., 1999, “Testing, one -- two -- three: Fundamentals of usability testing.”

(Brinkkemper 1990) Brinkkemper, J.N., 1990, “Formalization of Information Systems Modelling.” Katholieke Universiteit te Nijmegen, The Netherlands, doctoral dissertation published by Thesis Publishers.

(Brooks 2007) Brooks, Jr., F.P., 2007, “The Mythical Man-Month: Essays on Software Engineering.” 20th Anniversary Edition. Reading, MA: Addison-Wesley, 322 pages.

(Bunse 2007) Christian Bunse, Hans-Gerhard Gross, Christian Peper, 2007, “Applying a Model-Based Approach for Embedded System Development.” Report TUD-SERG-2007-020, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology.

(Chapman 1982) D. Chapman, 1982, “A Program Testing Assistant.” Communications of the ACM.

(Charrette 2005) Robert N. Charrette, 2005, “Why Software Fails?” IEEE Spectrum, pp. 42-49.

(Chavez 2000) Tom Chávez, 2000, “A decision-analytic stopping rule for validation of commercial software systems.” IEEE Transactions on Software Engineering, 26(9):907– 918.

(Cheng 2003) C. Cheng, A. Dumitrescu, P. Schroeder, 2003, “Generating Small Combinatorial Test Suites to Cover Input-Output Relationships.” Proceedings of 3rd Quality Software International Conference (QSIC '03) pp. 76-82.

(Chillarege 1992) Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, Man-Yuen Wong, 1992, “Orthogonal Defect Classification - A Concept for In-Process Measurements.” IEEE Transactions on Software Engineering, Vol. 18 N°11, pp.943-956.

(Chow 1978) Chow, T. S., 1978, “Testing software design modeled by finite state machines.” IEEE Transactions on Software Engineering, Vol. 4, No. 3, pp. 178-187.

(Chvalovsky 1983) Chvalovsky, V., 1983, “Decision tables.” Software: Practice and Experience, Vol. 13, No.5, pp. 423-429.

(Dalal 1988) S. R. Dalal and C. L. Mallows, 1988, “When should one stop testing software.” Journal of the American Statistical Association, 83(403):872–879.

(Dart 1987) Dart, S.A., Ellison, R.J., Feiler, P.H., and Habermann, A.N., 1987, “Software development environments.” IEEE Computer, 18-28.

(Davis 1988) Alan M. Davis, 1998, “A Comparison of Techniques for the Specification of External System Behavior.” In Communications of the ACM. ACM 31(9): 1098-1115.

(Dick 1993) Dick, J., Faivre, A., 1993, “Automating the Generation and Sequencing of Test cases From Model-Based Specifications.” Proceedings FME’93, LNCS 670 268-284.

(DoD 1996) Department of Defense, 1996, “Validation and Accreditation (VV&A) Recommended Practices Guide.” Defense Modeling and Simulation Office, Alexandria,

VAb(Coauthored by: O. Balci, P. A. Glasow, P. Muessig, E. H. Page, J. Sikora, S. Solick, and S. Youngblood).

(Dijkstra 1972) E.W. Dijkstra, 1972, “The humble programmer.” In Communications of the ACM, vol. 15, pp. 859–866. Turing Award Lecture.

(Duernberger 1996) P.M. Duernberger, 1996, “Software testing applications in a computer science curriculum.” Northcon/96, 4-6, pp. 291 – 293.

(Duphy 2000) Sophie Duphy, 2000, “Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information.” PhD dissertation, Joseph-Fourier University, Grenoble, France.

(El-Far 2001) I. K. El-Far and J. A. Whittaker, 2001, “Model-Based Software Testing.” Encyclopedia of Software Engineering (edited by J. J. Marciniak). Wiley.

(Fagan 1986) M. E. Fagan, 1986, “Advances in Software Inspections.” IEEE Transactions on Software Engineering, Vol. SE-12, No. 7.

(Fenton 1996) Fenton, N E, Pfleeger, SL, 1996, "Software Metrics: A Rigorous and Practical Approach." Book 2nd Edition, pp. 638, International Thomson Computer Press, London and Boston.

(Fenton 2000) Fenton, N. E., Ohlsson, N., 2000, “Quantitative Analysis of Faults and Failures in a Complex Software System.” IEEE Transaction on Software Engineering, 26(8) 797-814.

(Fernandez 1996) Fernandez, J.-C., Jard, C., Jeron, T., Nedelka, L., Viho, C., 1996, “Using On-the-fly Verification Techniques for the Generation of Test Suites.” Proceedings of the 8th International Conference on Computer Aided Verification.

(Fradet 2008) Frederick Fradet, 2008, “Software Validation Plan - Quality Control Checklist.” Internal document, Johnson Controls.

(Frankl 1998) Frankl, P. G., Hamlet, R. G., Littlewood, B., Strigini, L., 1998, “Evaluating Testing Methods by Delivered Reliability.” IEEE Transaction on Software Engineering, 24(8) 586-601.

(Fraser 1994) M.D. Fraser, K. Kumar, and V.K. Vaisnavi, 1994, “Strategies for Incorporating Formal Specifications in Software Development.” In Communications of the ACM, 37(10):74–86.

(Gale 1990) Gale, J.L., Tirso, J.R., and Burchfiels, C.A., 1990, “Implement the defect prevention process in the MVS interactive programming organization.” IBM Systems Journal, vol. 29, no. 1, 33-43.

(Garey 1979) Garey, M. and D. Johnson, 1979, “Computers and Intractability: A Guide to the Theory of NP-Completeness.”

(Gass 1987) Gass, S. I. and L. Joel, 1987, “Concepts of Model Confidence.” Computers and Operations Research, 8, 4, pp. 341–346.

(Gaudel 1995) Marie-Claude Gaudel, 1995, “Testing can be formal, too.” In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, TAPSOFT’95: Theory and Practice of Software Development, vol. 915 of LNCS (Lecture Notes in Computer Sciences), pp. 82–96, Aarhus, Denmark. Springer Verlag.

(Gibson 1992) R.Gibson, 1992, “Managing Computer Projects” Prentice-Hall.

(Gill 1962) Gill, A, 1962, “Introduction to the theory of finite-state machines.” McGraw Hill, NJ.

(Grady 1992) Grady, R., 1992, “Practical Software Metrics for Project Management and Process Improvement.” Prentice-Hall, Englewood Cliffs, NJ.

(Green 1998) Darryl Green and Ann DeCaterino, 1998, “A Survey of System Development Process Models.” Center for Technology in Government University at Albany / SUNY.

(Hall 1990) Hall, A., 1990, “Seven myths of formal methods.” IEEE Software (7,5), 11-19.

(Harel 1987) Harel, D., 1987, “Statecharts: a Visual Formalism for Complex Systems.” Journal of Science of Computer Programming, 8, pp. 231-274.

(Harrold 2000) Harrold M. J., 2000, “Testing: A Roadmap.” Future of Software Engineering, 22nd International Conference on Software Engineering.

(Hessel 2007) Anders Hessel and Paul Pettersson, “A Global Algorithm for Model-Based Test Suite Generation.” Third Workshop on Model-Based Testing, Braga, Portugal, Satellite workshop of ETAPS 2007.

(Hong 2000) Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae and Hasan Ural, 2000, “A Test Sequence Selection Method for Statecharts.” The Journal of Software Testing, Verification & Reliability, 10(4): 203-227.

(IEEE Std. 610-1990) IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610-1990, IEEE Standards Software Engineering, Vol. 1; The Institute of Electrical and Electronics Engineers, ISBN 0-7381-1559-2.

(IEEE Std. 1044-1993) IEEE Standard Classification for Software Anomalies, IEEE Std. 1044-1993; The Institute of Electrical and Electronics Engineers, 1993.

(IEEE Std. 829-1998) Software Test Documentation, IEEE Std. 829-1998; The Institute of Electrical and Electronics Engineers, 1998.

(IEEE Std. 830-1998) IEEE Recommended Practice for Software Requirements Specifications (SRS), IEEE Std. 830-1998; The Institute of Electrical and Electronics Engineers, 1998.

(Jorgensen 1995) P. C Jorgensen, 1995, “Software Testing a Craftsman’s Approach.” CRC Press.

(Karp 1972) R. M. Karp, 1972, “Reducibility among combinatorial problems.” In Thomas J., editor, Complexity of Computer Computations, Proceedings Symp., pp. 85–103. IBM.

(Kemeny 1976) J. G. Kemeny and J. L. Snell, 1976, “Finite Markov chains.” Springer-Verlag, New York.

(Kemmerer 1990) Kemmerer, R.A., 1990, “Integrating formal methods into the development process.” IEEE Software, 37-50.

(Sayre 2000) Kirk D. Sayre and Jesse H. Poore, 2000, “Stopping criteria for statistical testing.” Information and Software Technology, 42(12):851–857.

(Le Corre 2006) Pascal Le Corre, 2006, “AUTOCODE : C-Code generation from Statemate impact.” Internal document, Johnson Controls.

(Leen 2002) Gabriel Leen, Donal Heffernan, 2002, “Expanding Automotive Electronic Systems.” IEEE Computer 35(1): 88-93.

(Legeard 2007) Bruno Legeard, 2007, "Software testing course." Computer laboratory, Franche-Comté University, France.

(Leszak 2000) Marek Leszak, Dewayne E. Perry, Dieter Stoll, 2000, "A case study in root cause defect analysis." Proceedings of the 22nd International Conference on Software Engineering, June 4-11, pp. 428-437.

(Leveson 1990) Leveson, N.G., 1990, "Formal methods in software engineering." IEEE Transaction on Software Engineering, 16, 9, 929-931.

(Liggesmeyer 1998) P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann, 1998, "Qualitatssicherung Software-basierter Technischer Systeme--Problembereiche und Lösungs-ansätze." Informatik Spektrum, 21(5):249--258.

(Littlewood 1997) B. Littlewood and David Wright, 1997, "Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software." IEEE-TSE, Vol. 23, No.11.

(Liu 2000) Chang Liu and Debra J. Richardson, 2000, "Using application states in software testing." Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), p. 776, ACM, Cambridge, MA, USA.

(Lyu 1996) M.R Lyu, 1996, "Handbook of Software Reliability Engineering." McGraw-Hill.

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

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

(Mays 1990) Mays, R.G., Jone, C.L., Holloway, G.J., and Sudinski, D.P., 1990, "Experiences with defect prevention." IBM Systems Journal, vol. 29, no. 1, 4-32.

(McDonald 2007) McDonald, Marc; Musson, Robert; Smith, Ross, 2007, "The Practical Guide to Defect Prevention." Microsoft Press, 480. ISBN 0735622531.

(Mignen 2005) Claude Mignen, 2005, "Lessons Learned Training." Internal document, Johnson Controls.

(Mignen 2006a) Claude Mignen, 2006, "Software development work instructions." Internal document, Johnson Controls.

(Mignen 2006b) Claude Mignen, 2008, "Software role description." Internal document, Johnson Controls.

(Mignen 2008) Claude Mignen, 2008, "Software process description – Requirements specification work instruction." Internal document, Johnson Controls.

(Mellor 1992) Mellor P., 1992, "Failures, faults, and changes in dependability measurement." Information and Software Technology 1992, 34(10), pp.640-54.

(Moavenzadeh 2006) Moavenzadeh, J., 2006, "Offshoring Automotive Engineering: Globalization and Footprint Strategy in the Motor Vehicle Industry." National Academy of Engineering.

(Moret 1982) Moret, B., 1982, "Decision Trees and Diagrams." ACM Computing Surveys (CSUR), Vol.14, No.4, pp.593-623.

(Myers 1978) Myers, G. J., 1978, "A controlled experiment in program testing and code walkthroughs/inspections." Communications of the ACM, vol. 21, no. 9, pp. 760-768.

(Myers 1979) G.J. Myers, 1979, "The Art of Software Testing." John Wiley and Sons, New York.

(NIST 2002) National Institute of Standards and Technology, 2002, "The economic impacts of inadequate infrastructure for software testing: final report." Planning report 02-3. Gaithersburg, MD: NIST.

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

(Panzl 1978) D.J. Panzl, 1978, "Automatic Software Test Drivers." Computer, 11(4):44–50.

(Paulk 2000) Paulk, M.C., Goldenson, D., White, D.M., 2000, "The 1999 survey of high maturity organizations." Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2000-SR-002.

(Pezze 1998) Mauro Pezze, Michal Young, 1998, "Software Testing and Analysis: Problems and Techniques." Proceedings of the 20th International Conference on Foundations on Software Engineering (FSE'98), Orlando.

(Powell 1986) Powell, P. B, 1986, "Planning for Software Validation, Verification, and Testing." In Software Validation, Verification, Testing and Documentation, S. J. Andriole, ed. Princeton, N. J.: Petrocelli, 3-77.

(Pressman 1997) R.S. Pressman, 1997, "Software Engineering - A Practitioner's approach." McGraw-Hill, Fourth Edition.

(Robinson 1999) Harry Robinson, 1999, "Finite state model-based testing on a shoestring." Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999), Software Quality Engineering, San Jose, CA, USA.

(Sangiovanni-Vincentelli 2003) Alberto Sangiovanni-Vincentelli, 2003, "Electronic-System Design in the Automobile Industry." IEEE Micro, vol. 23, no. 3, pp. 8-18.

(Sargent 1984) Sargent, R. G., 1984, "Simulation Model Validation, Simulation and Model-Based Methodologies: An Integrative View." Edition Oren, et al., Springer-Verlag.

(Sargent 2005) Sargent, R. G., 2005, "Verification and validation of simulation models." Winter Simulation Conference 2005: 130-143.

(Seroussi 1988) Seroussi, N. H. Bshouty, 1988, "Vector sets for exhaustive testing of logic circuits." IEEE transaction on information theory, vol. 34, pp. 513-522.

(Sommerville 1997) Ian Sommerville and Pete Sawyer, 1997, "Requirements Engineering: A Good Practice." Wiley.

(So 2002) So, S. S., Cha, S. D., Shimcall, T. J., and Know, Y. R., 2002, "An empirical evaluation of six methods to detect faults in software." Software Testing, Verification and Reliability, vol. 12, no. 3, pp. 155-171.

(Spatz 2002) Julius Spatz & Peter Nunnenkamp, 2002, "Globalization of the Automobile Industry - Traditional Locations under Pressure?" Kiel Working Papers 1093, Kiel Institute for the World Economy.

(Sturgeon 2000) Sturgeon, Timothy & Florida, Richard, 2000, "Globalization and Jobs in the Automotive Industry." MIT Center for Technology, Policy, and Industrial Development.

(Walton 2000) G. H. Walton and J. H. Poore, 2000, "Measuring complexity and coverage of software specifications." Information and Software Technology, 42(12):815-824.

(Weyuker 1982) Elaine J.Weyuker, 1982, “On testing non-testable programs.” Computer Journal, 25:465–470.

(Weyuker 1991) Weyuker, E. J., Jeng, B., 1991, “Analyzing Partition Testing Strategies.” IEEE Transaction Software Engineering, 17(7) 703-711.

(Whittaker 1994) James A. Whittaker and Michael G. Thomason, 1994, “A Markov chain model for statistical software testing.” IEEE Transactions on Software Engineering, 20(10):812-824.

(Wilkes 1949) Wilkes, M. V., 1949 “Progress in High-Speed Calculating Machine Design.” Nature vol. 64 page 341.

(Wing 1990) Wing, J.M., 1990, “A specifier’s introduction to formal methods.” IEEE Computer 23, 9, 8-24.

(Wood 1986) Wood, D. O., 1986, “MIT Model Analysis Program: What We Have Learned About Policy Model Review.” Proceedings. of the 1986 Winter Simulation Conference, Washington, D.C., pp. 248–252.

(Woodward 2005) Martin R. Woodward and Michael A. Hennell, 2005, “Strategic benefits of software test management: a case study.” Journal of Engineering and Technology Management, Vol. 22, Issues 1-2, pp. 113-140.

(Yang 2006) Yang O., Jenny Li J. and Weiss D., 2006, “A Survey of Coverage Based Testing Tools.” In International workshop on Automation of Software Test, AST ’06, Shanghai, China, pp. 99-103.

APPENDICES

Appendix A: Verification and Validation static tools

In Johnson Controls, there is a document which defines *coding rules* and *recommendations* for using the *C language*[44] in the development of embedded software. These *rules* and *recommendations* are defined and updated by a committee, whose members are appointed by the *Software Engineering Process Group* (*SEPG*) of the company. The committee includes representatives of all Johnson Controls sites on which this document is deployed. An excerpt of the *coding rules* and *recommendations* is illustrated in *Table A.1*.

Rule number	Rule type	Rule description
Rule 1	General	Optimization objectives must be defined before coding. These objectives define priorities between optimization ways (memory…). Do not optimize unless it is planned. It has been demonstrated many times that the programmers spend a considerable amount of energy to optimize a piece of code that will almost never be used. Before starting to optimize always identify the exact nature of the problem.
Rule 2	Comments	Comments shall be written in US English language.
Rule 3	Code layout	Each variable must be declared on a separate line.
Rule 4	Naming rules	Never use names that differ only by uppercase/lowercase.
Rule 5	Functions	A function must never return a pointer to one if it is a local function. Doing so, would rather be a bug than just a rule break.
Rule 6	Flow control	Give all loops a fixed upper bound.
Rule 7	Variables	No multiple assignments a=b=c=d;

Table A.1 – An excerpt of coding rules and recommendations used in Johnson Controls (Johnson Controls source)

The *static analysis* is performed automatically using a computer tool such as *QAC*[45], the most used in automotive industry. The criterion to stop the *static analysis* of a *source code* is that all *QAC errors* and *warnings* are either fixed or justified. A screenshot of the *QAC* tool is illustrated in *Figure A.1*.

[44] Computer language
[45] http://www.programmingresearch.com/QAC_MAIN.html (Consulted on November 2008)

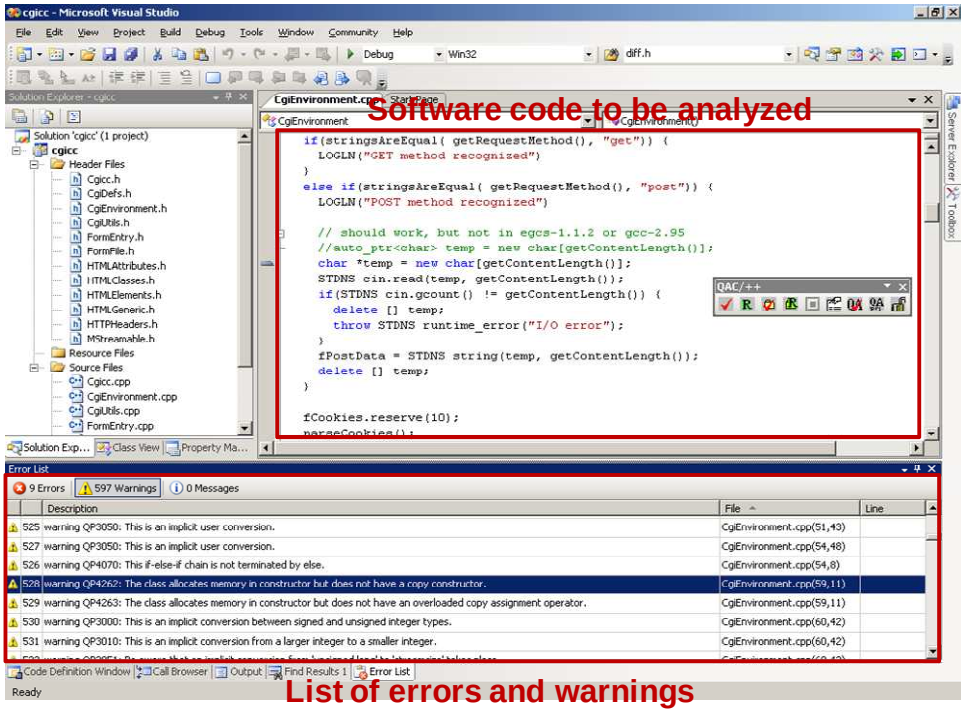


Figure A.1 – Screenshot of the static analysis tool (*QAC*)

The *dynamic analysis* is performed automatically thanks to a commercial tool *PolySpace*[46]. The criterion to stop the *dynamic analysis* of a *source code* is that all *Polyspace errors* and *warnings* are fixed or justified. A screenshot of the *Polyspace* tool is illustrated in *Figure A.2*.

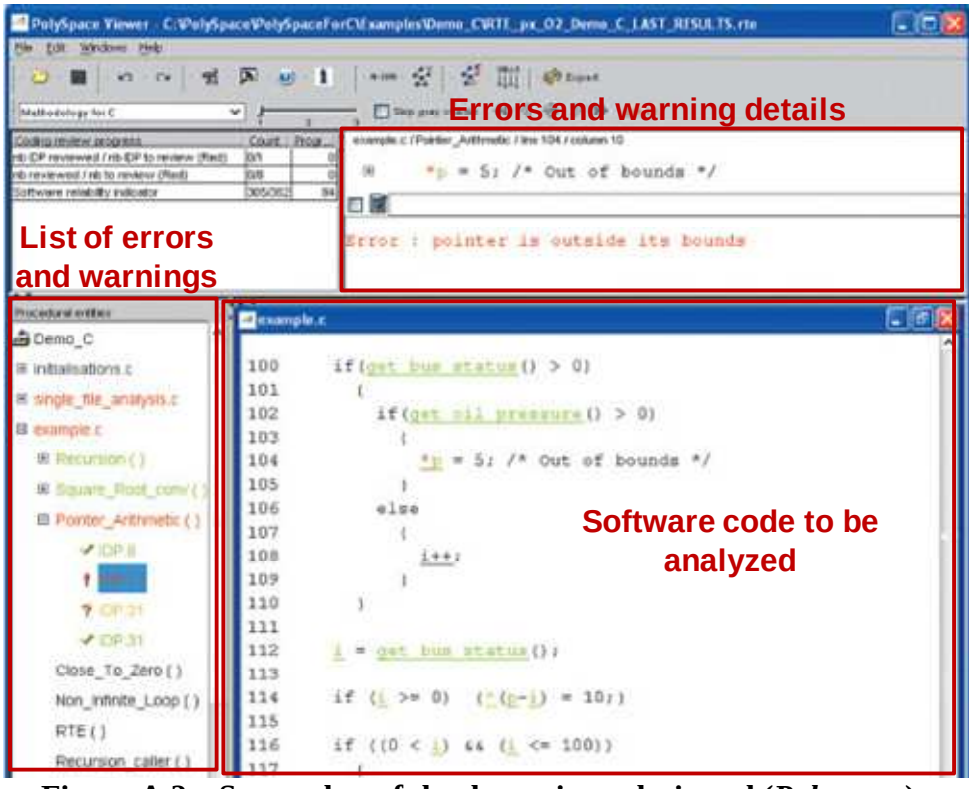


Figure A.2 – Screenshot of the dynamic analysis tool (*Polyspace*)

[46] http://www.mathworks.com/products/polyspace/index.html (Consulted on November 2008)

Appendix B: Test description languages

As developed in *Chapter 2 – Section 5*, three techniques of *software testing* are performed before a software delivery to the customer: *unit*, *integration* and *validation test*. In case of *unit test*, the execution of *test cases* is always automated using a *test execution platform* (Cf. *Appendix C*). The language used for describing *test cases* for *unit test* is the *C language*. However, the language used to design *test cases* for the *validation test* of a software product depends on the *validation test execution platform*. In case of an automatic execution of the *test cases*, one uses a *script language*. It is a Johnson Controls property language very similar to the well-known *Visual Basic*[47] language. In case of a manual execution of the test cases, *test cases* are written in *natural language*.

1. Unit test language

A standard *unit test* structure provides predefined *C functions* in order to help test engineers writing *test cases* for the unit test of a software component:

- **Function 1: TSTStartPhase(Title),** display *Title*.
  - The *test cases* should be broken down in phases to facilitate the test results interpretation.
- **Function 2: TSTWaitMs(Delay)**, wait *Delay*.
  - This function is essential because the time is only simulated when this function is called. The time is executed at the maximum speed.
  - *Delay* should be in milliseconds.
- **Function 3: mTSTCheck(Condition)**, generate an error if *Condition* is false.
  - *Condition* is a boolean.
- **Function 4: TSTTerminate()**, display *Final results* of the test
  - List the number of *checked points*.
  - List the number of bugs.
  - Indicates *Test NOK* or *Test OK* if an error has been detected or not.

An excerpt of *test cases* designed for the *unit test* of a software component is illustrated in *Figure B.1*.

[47] Computer language (http://msdn.microsoft.com/en-us/library/sh9ywfdk(vs.80).aspx, Consulted on November 2008).

```
// test case 11
Input1 = 11;
Input2 = 12;
Input3 = 14;

TSTWaitMs(5000);
mTSTCheck(Output1 == 0);
mTSTCheck(Output2 == 1);
mTSTCheck(Output3 == 0);

// test case 12
// 0x0 - essence, 0x10 - diesel
TYPCARB = 0;
Input1 = 0;
Input2 = 12;
Input3 = 14;

TSTWaitMs(1000);
mTSTCheck(Output1 == 0);
mTSTCheck(Output2 == 0);
mTSTCheck(Output3 == 0);

// test case 13
// 0x0 - essence, 0x10 - diesel
TYPCARB = 2;
Input1 = 0;
Input2 = 12;
Input3 = 20;

TSTWaitMs(500);
mTSTCheck(Output1 == 1);
mTSTCheck(Output2 == 0);
mTSTCheck(Output3 == 0);
```

Figure B.1 – An excerpt of test cases designed for the unit test of a software component (Johnson Controls source)

2. Integration and Validation test language (when automating the test case execution)

The *test script language* developed in Johnson Controls is mainly based on the universal *Visual Basic* language. A set of coding *rules* and *recommendations* to be taken into account when designing *test cases* using the *test script language* has been defined. An excerpt of these *rules* and *recommendations* is illustrated in *Table B.1*.

Rule number	Description
Rule 1	The “include” files must not have the full access path indicated. Example (in C): #include "../../h/defs.h" is OK #include "defs.h" is OK #include <defs.h> is OK #include "c:\sources/h/defs.h" is NOK
Rule 2	The validation procedures titles must be the same as the title of the SW function (functionality) which is testing Example: Odometer, Trip meter, Diagnostic, Engine Speed, Vehicle speed, Warnings, etc.
Rule 3	Any Test Step having state NOK, must refer to defect reference.
Rule 4	Random values in Test Actions and Preconditions are not allowed in any circumstances. Please note: Arbitrary values are allowed. Such test types are part of many test procedures. To develop more efficient Validation Procedures loop operators shall be used instead linear programming.

Table B.1 – An excerpt of the validation test script coding rules and recommendations (Johnson Controls source)

The test script is made of a set of *statements* organized in a *processes*, *sub-programs* and *functions* structure. The overall structure is described in *Figure B.2*.

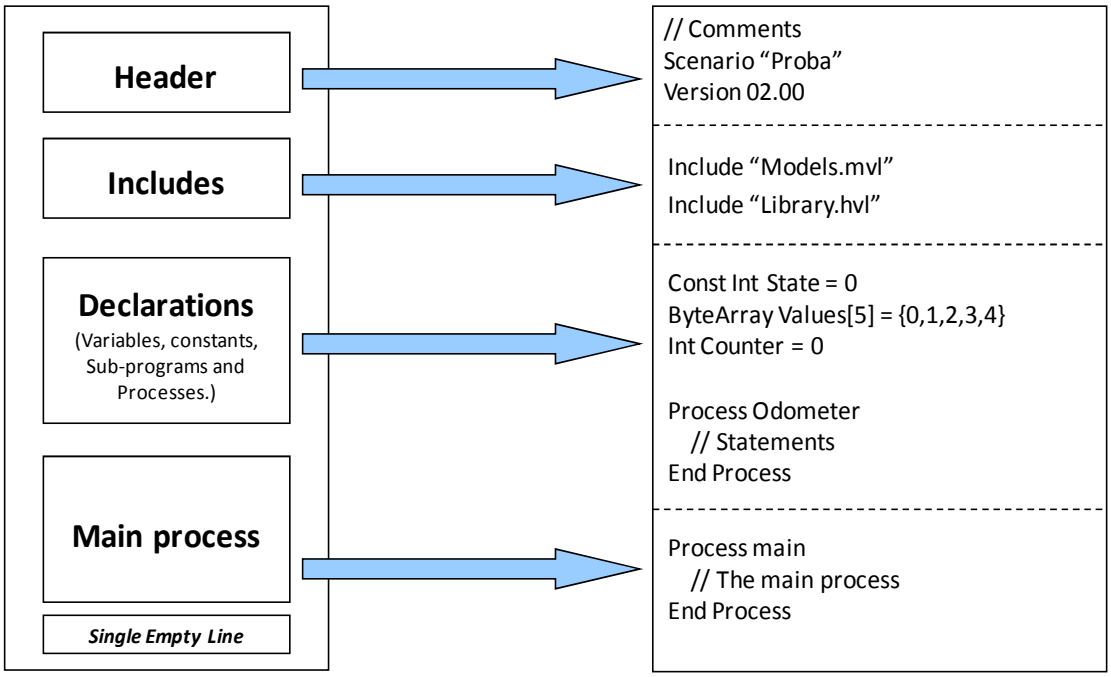


Figure B.2 – Overall structure of a test script program (Johnson Controls source)

The grammar of the test script language is developed in *Figure B.3*.

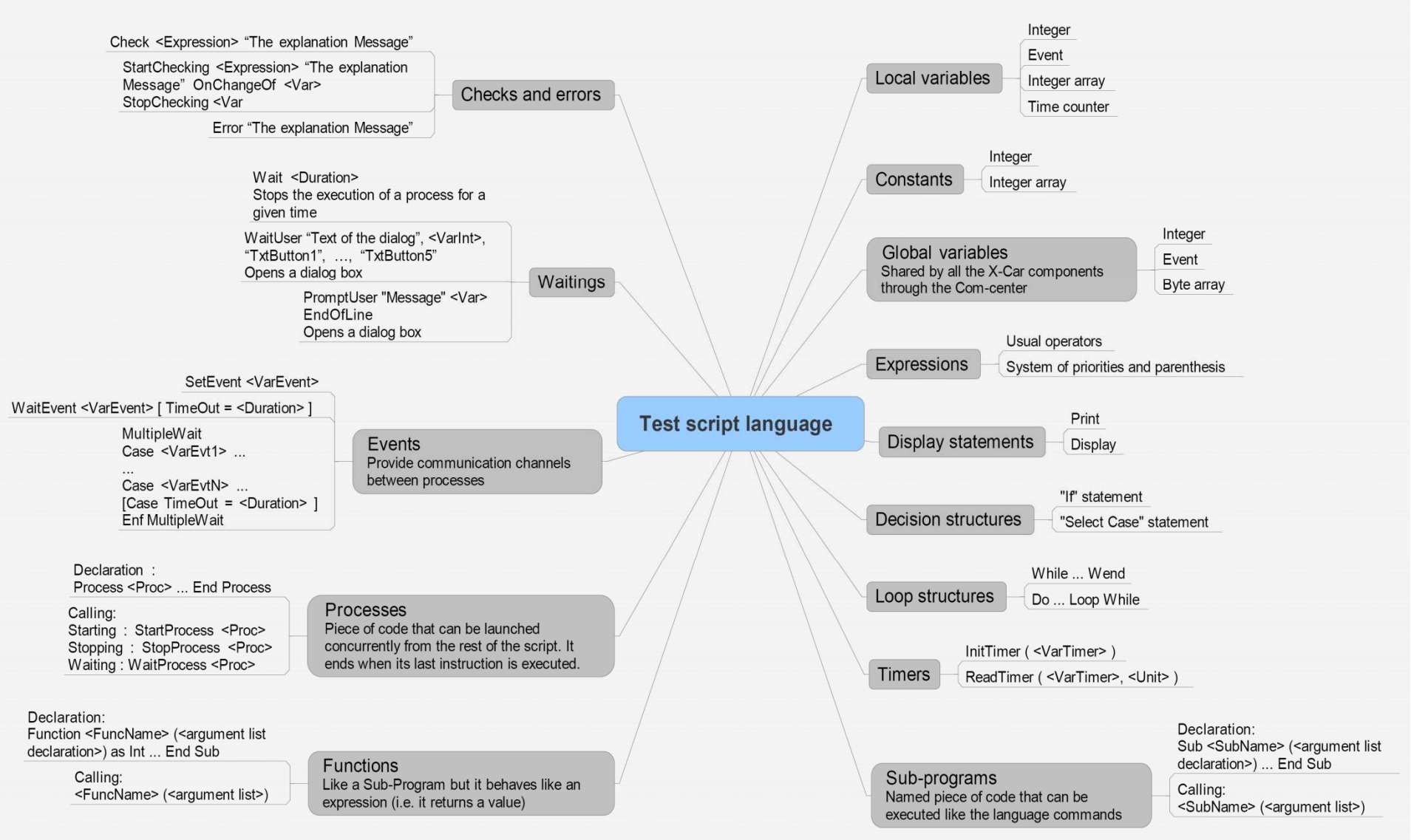


Figure B.3 – Grammar of the test script language

As developed in *Appendix C*, a computer platform (*X-Car*) has been developed in order to execute *test cases* for the integration and validation test of a software product. This platform has a *test language interpreter* tool which allows to perform initial check for *test script* correctness, run automation *test script*, handle data automatically by script and derive output for reporting. A screenshot of the *script language interpreter tool* is illustrated in *Figure B.4*.

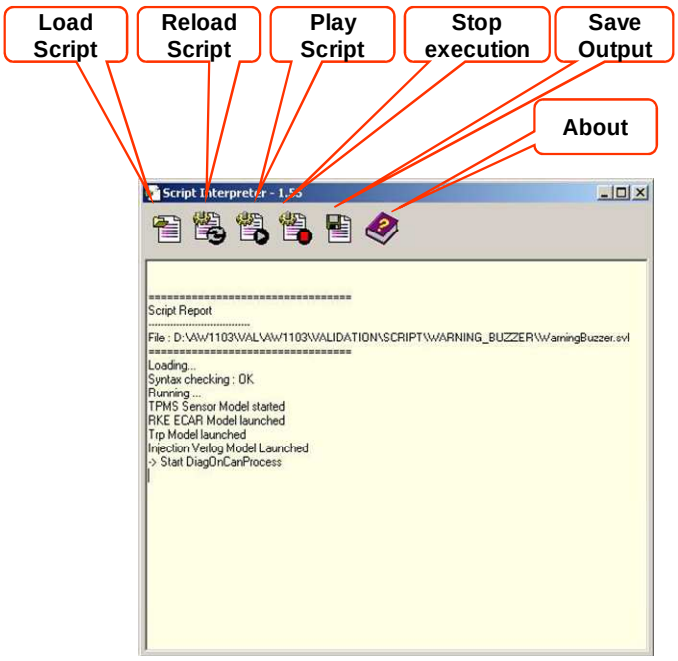


Figure B.4 – Screenshot of the test script interpreter (Johnson Controls sources)

Another tool is the *test script sequencer* which allows to manage a list of *test scripts* in order to execute them automatically and consecutively in a specified order. Each script in the list has a status and it can be activated or deactivated. Several action can be executed before each script (reset the software, reload the software, launch an initialization script). A screenshot of the *test script sequencer tool* is illustrated in *Figure B.5*.

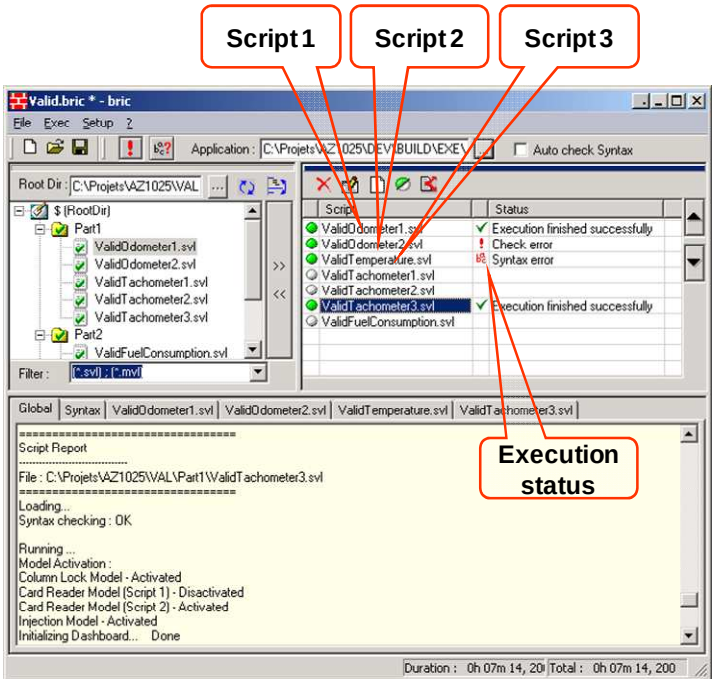


Figure B.5 – Screenshot of the test script sequencer (Johnson Controls source)

Appendix C: Test execution platforms

1. Unit test execution platform

During the *unit test* of a software component, the designed *test cases* are executed on the component automatically via the *unit test execution platform*. In fact, all the dependencies and connections between the components are *simulated on computer* in order to isolate the tested component from the project. Test results are analyzed to decide if *Component Development* activities have to be restarted, in case of failed tests. The *code coverage* is recorded and used as criteria to stop the design of *test cases*. The abstract model of the *unit test execution platform* is illustrated in *Figure C.1*.

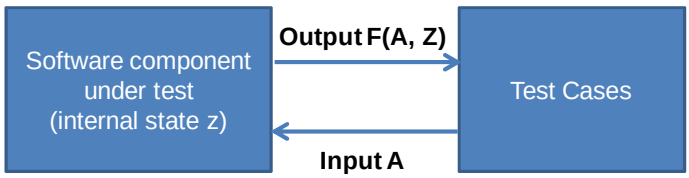


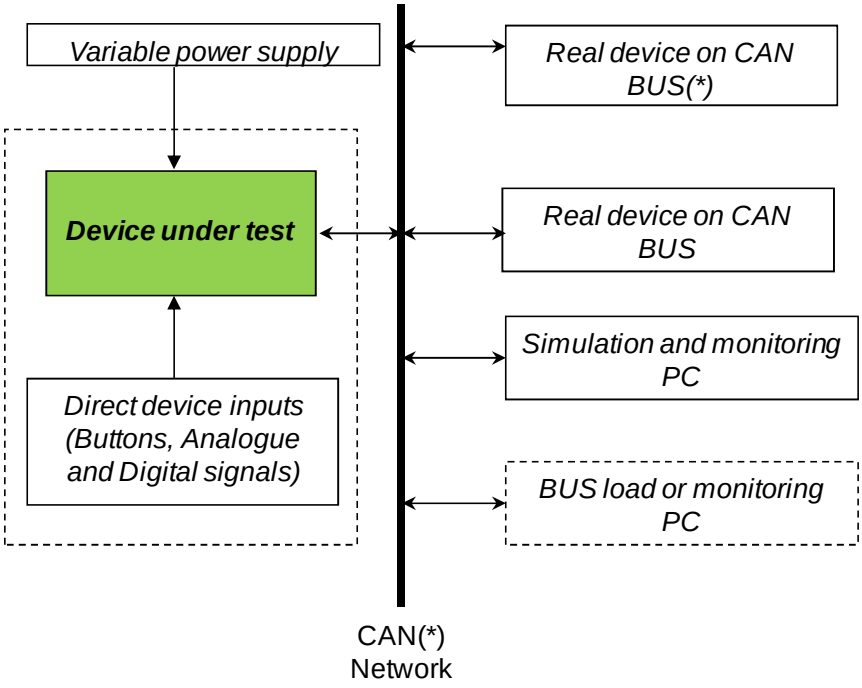
Figure C.1 – Abstract model of the unit test execution platform (Johnson Controls source)

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.*

2. Integration and Validation test execution platform

During the *integration and validation tests* of a software product, the test execution platform could be manual or automatic. For each project, managers (in close cooperation with the carmaker) decide to automate or not the execution of the *validation test cases*. In case of an automatic execution, *test cases* are designed in a *script language* (Cf. *Appendix B*). In case of a manual execution, *test cases* are written in *natural language*. The manual execution aims to perform *operations* manually on the software product via a set of switches and to check visually (by an engineer) the behavior of the output signals (lamps, actuators …). In the following, we develop the automatic *test execution platform*.

The *Software Validation Plan* (*SVP*) supports the definition of the validation *test execution platform* (Cf. *Chapter 2 – Section 5.D.1*)*:* the necessary equipments and the common and reused validation components. The functional model of a validation *test execution platform* is shown in *Figure C.2*.



(*) Controller Area Network (or CAN) is the latest communication system within the automotive world. At its simplest level, it can be thought of as a means of linking all of the electronic systems within a car together to allow them to communicate with each other (http://www.semiconductors.bosch.de/en/20/can/index.asp Consulted on November 2008)

Figure C.2 – Functional model of a validation test execution platform (Johnson Controls source)

An excerpt of a list of hardware and software tools required for the execution of *validation test case*s is illustrated in *Table C.1* and *C.2*.

ID	Tool Type	Name	Mandatory	Comment
[HW_T1]	*Power supply*	*Constant/variable/programmable*	*Y/N*	*Information on the tool configuration* *Specific inputs, outputs and features required* *Work instructions for the tool*
[HW_T2]	*<Measuring instrumentation >*	*Oscilloscope*	*Y/N*	
…	…	…	…	…

Table C.1 – An excerpt from a hardware tool list required for the execution of validation test cases

ID	Tool Type	Name	Mandatory	Comments
[SW_T1]	*Report generator to*	*Reporter*	*Y/N*	*Information for the tool configuration* *Specific input, output and feature required* *Work instruction for the tool* *If related with any HW, mention it.* *If standard guideline/work instruction for the tool is available, put it as reference document.*
[SW_T2]	*Test tool 1*	*R-car Intermat*	*Y/N*	
…	…	…	…	…

Table C.2 – An excerpt from a software tool list required for the execution of validation test cases

Finally, an excerpt of a list of reused components for the execution of *validation test cases* is illustrated in *Table C.3*.

ID	Tool Type	Name	Mandatory	Comment
[SW_V1]	*Source code*	*Library for programming power supply*	*Y/N*	*Brief description* *If related with any HW or SW, mention it.* *Path to the original Configuration management base* *Work instruction for the tool*
[SW_V2]	*Test script*	*Test cases for <Functionality i>*	*Y/N*	
…	…	…	…	…

Table C.3 – An excerpt from a reused components list required for the execution of validation test cases

The aim of the software *validation test* is to test the functional behavior of a software product in its real environment. Therefore, we need to simulate this environment (hardware, other electronic devices, network …). For that purpose, Johnson Controls has developed two types of *test execution platform*:

E-Car (*Emulated Car* – Cf. *Figure C.3*) is a *simulation on computer* of the entire electronic automotive network with all the electronic devices. This platform simulates also the hardware on which the software product under test must perform. It composes network frames on specified periods, fills them with the appropriate signals and sends them on a virtual network bus. It also simulates pressing of buttons and reaction of sensors in the car.

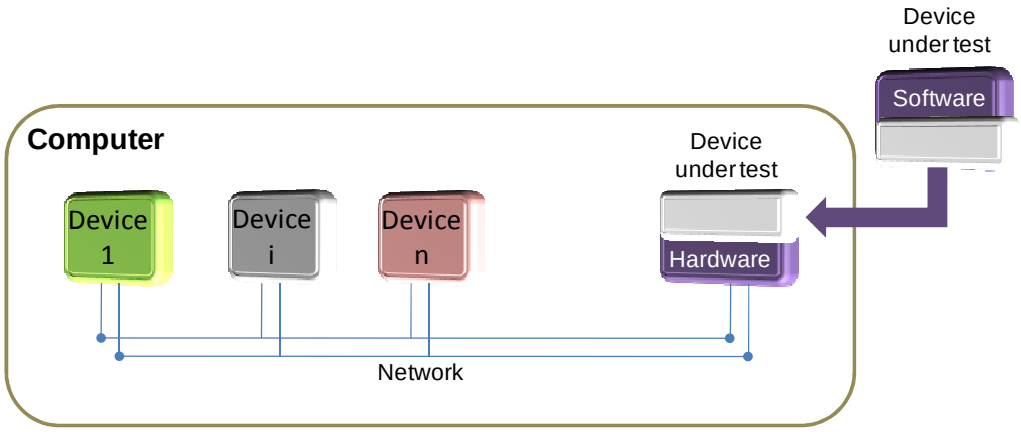


Figure C.3 – E-Car environment (Johnson Controls source)

R-Car (*Real Car* – Cf. *Figure C.4*) is a hardware - software interface used when the hardware of the software product under test is real physical target. It transforms the parametric signals into real electric signals and sends them on the specified channels in the appropriate format.

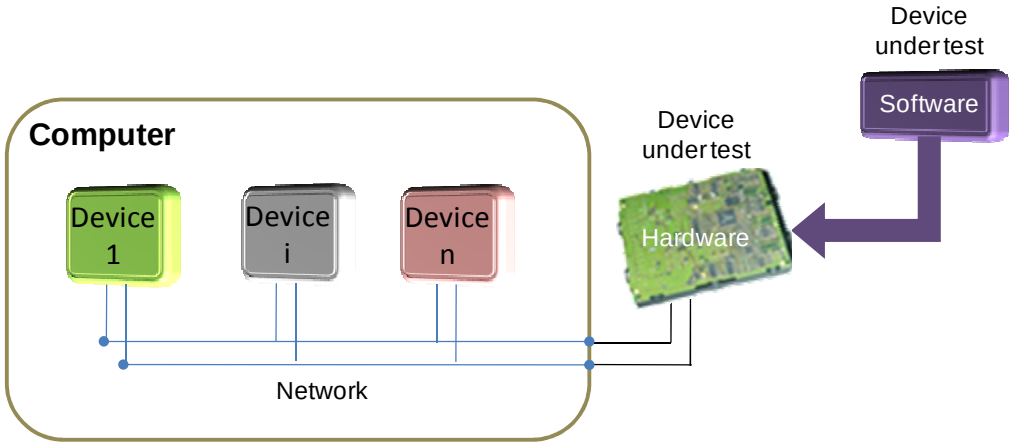


Figure C.4 – R-Car environment (Johnson Controls source)

X-Car is the base framework which allows the running of *E-Car* or *R-Car* plus some additional programs to support the *validation execution* (Cf. *Figure C.5*):

- *Network viewer:* This tool allows to trace or spy different types of data's exchange via the network.
- *Test language interpreter:* It allows to run automation *test scripts* and to perform initial check for script correctness. It handles data automatically by script and derives output for reporting.
- *Bench tool:* This tool simulates the inputs and outputs of the device under test. It shows data output state, handles data input state and shows data access status.
- *Display simulator:* It shows output state by switching between two pictograms, simulate pointer indicator on a dashboard and simulate dot matrix display.

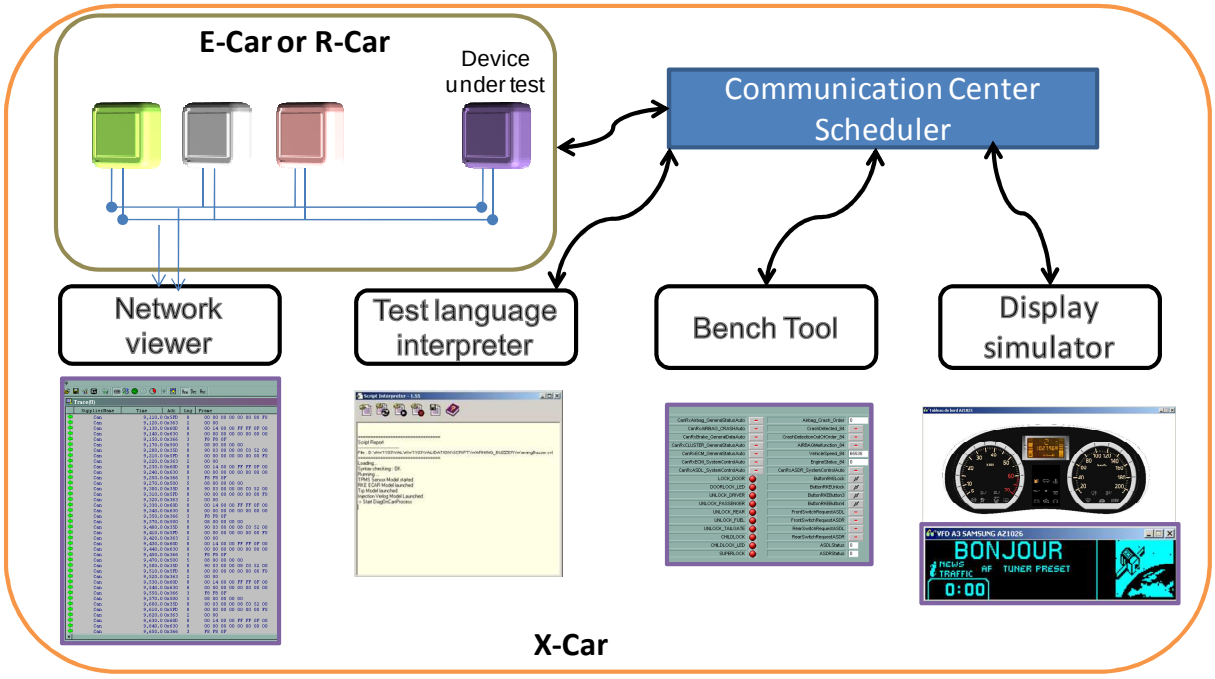


Figure C.5 – X-Car framework (Johnson Controls source)

The core of *X-Car* is the *Communication Center* where all the signals present in the vehicle network are stored. Every program that attempt to modify or to check the value of a signal will go in there. Another important component is the *Scheduler* which manages in time the platform.

Integration test may be executed on *E-Car*; however, *validation test* may be executed either on *E-Car* or on *R-Car*. When a bug is detected on *E-Car*, it must be confirmed on *R-Car*. In fact, *E-Car* is a simulation on computer while the *R-Car* is the real physical environment and therefore the behavior of the real hardware can differ from the behavior of the simulation hardware.

Appendix D: Commercial test case design tools (Survey done in 2006)

Tool name	Company name	Company location	Description	Input	Output	Free trial	Application domain
CONFORMIQ TEST GENERATOR	VERYSOFT	Germany	**Conformiq Test Generator** is a solution for dynamic model-based test generation and automatic test execution	Graphical test model which uses extended UML statecharts	A system adapter translates the test cases provided by Conformiq Test Generator into a format understood by the test execution platform	YES	Automotive, Aircraft, Telecommunication
MATELO	ALL4TEC	France	**MaTeLo** generates, according to several optimization algorithms, test cases from a usage model	Usage model	Test cases are generated in XML/HTML format for manual execution or in TTCN-3 and TestStand 2.0 for automatic execution	NO	Telecommunication, Automotive, Railway, Aerospace
PRO-TEST/PRAXIS	DIGITAL COMPUTATIONS, INC	USA	**Pro-Test** is a Windows based stand-alone tool implementing HTT approach. The goal of HTT is to ensure all pairs test case coverage with a minimal number of test cases **Praxis** is a new service-based HTT solution. Praxis offers a custom application of HTT to problem based upon specific needs	Software input/level listing	Test cases can be exported in a variety of formats including XML, Excel, and HTML	YES	Telecommunication, Railway, Aerospace, Defense, PC Software editor
REACTIS	REACTIVE SYSTEMS, INC	USA	**Reactis** automates the generation of test data from Simulink and Stateflow models	Simulink and Stateflow model	Test Cases are saved in a special format (".rst")	YES	Automotive, Telecommunication, Aerospace, Medical.
RHAPSODY TESTCONDUCTOR/AUTOMATIC TEST GENERATOR	I-LOGIX/TELELOGIC	USA	**Rhapsody TestConductor** is a UML compliant, scenario-based test generation for real-time embedded applications. With Rhapsody TestConductor, developers can test a design against its requirements **Rhapsody Automatic Test Generator** is a UML model-based testing solution. ATG allows engineers to define and test individual components for specific purposes such as state and transition coverage	UML diagram	UML diagram	NO	Automotive, Telecommunication, Aerospace, Medical, Defense

… To be continued

Tool name	Company name	Company location	Description	Input	Output	Free trial	Application domain
T-VEC RAVE/TESTER for Simulink/Stateflow	T-VEC	USA	**T-VEC RAVE** solution is a method and integrated toolset for requirement-based defect prevention and automated testing. System requirements are modeled and analyzed with RAVE before design and coding **T-VEC Tester** analyzes Simulink and Stateflow models for errors, and generates comprehensive test cases for verifying the models and their implementations	**T-VEC RAVE:** T-VEC Tabular model **T-VEC Tester:** Simulink and Stateflow model	**T-VEC RAVE:** Test cases can be transformed in test drivers in any programming language or test scripts for any test execution tool **T-VEC Tester:** Test scripts or drivers are automatically generated for executing tests in Matlab simulator	YES	Automotive, Telecommunication, Aerospace, Medical, Client-Server, Web

Table D.1 – Commercial test case design tools (Survey done in 2006)

Appendix E: A second-level typology of software problems

Problem/Correction type		
First-level	**Second-level**	**Second-level : description**
Specification update	Requirements incorrect	*The requirement or a part of it is incorrect*
	Requirements logic	*The requirement is illogical or unreasonable*
	Requirements completeness	*The requirement as specified is either ambiguous, incomplete, or overly specified*
	Requirements verifiability	*Specification bugs having to do with verifying that the requirement was correctly or incorrectly implemented*
	Requirements presentation	*Bugs in the presentation or documentation of requirements. The requirements are presumed to be correct, but the form in which they are presented is not.*
	Requirements changes	*Requirements, whether or not correct, have been changed between the time programming started and testing ended*
Design update *… To be continued*	Design correctness	*Having to do with the correctness of the design*
	Design completeness, Feature	*Having to do with the completeness with which features are designed*
	Design completeness, Requirement	*Having to do with the completeness of requirements within features*
	Domains	*Processing requirements or feature depends on a combination of input values. A domain bug exists if the wrong processing is executed for the selected input-value combination*
	User messages and diagnostics	*User prompt or printout or other form of communication is incorrect. Processing is assumed to be correct: e.g., a false warning, wrong message*
	Exception conditions mishandled	*Exception conditions such as failure modes, which require special handling, are not correctly handled or the wrong exception-handling mechanisms are used*
	Diagnostic conditions mishandled	*Diagnostic conditions, which require special handling, are not correctly handled*

Design update		*or the wrong diagnostic-handling mechanisms are used*
	Software architecture	*Architectural problems*
	Performance	*Bugs related to the throughput-delay behavior of software under the assumption that all other aspects are correct*
	Partitions and overlays	*Memory or virtual memory is incorrectly partitioned, overlay to wrong area, overlay or partition conflicts*
	Environment	*Wrong operating system version, or other host environment problem*
	External and other third-party software	*Bugs in the interface to third-party software or other software developed externally. Due to a misunderstanding or wrong interpretation of the features and operation of the third-party software; or due to problems in the third-party software which the vendor does not correct*
Implementation update *… To be continued*	**Data definition, structure, declaration**	***Bugs in the definition, structure and initialization of data: e.g., Type, Dimension, Initial values, default values, Duplication, Scope (local, global), Static/dynamic resources***
	Data access and handling	***Having to do with access and manipulation of data objects that are presumed to be correctly defined: e.g. Type, Dimension, Duplication, Resources, Access***
	Control flow and sequencing	***Bugs specifically related to the control flow of the program or the order and extent to which things are done, as distinct from what is done***
	Processing	***Bugs related to processing under the assumption that the control flow is correct***
	Coding and typographical	***Bugs which can be clearly attributed to simple coding and typographical bugs. If a programmer believed that the correct variable was "ABCD" instead of "ABCE" but she/he changed D to E because of a typewriting bug, then it belongs to this correction type***
	Standards violation	***Bugs having to do with violating or misunderstanding the applicable programming standards and conventions (MISRA, Johnson Controls rules …).***

Implementation update		***The software is assumed to work properly***
	Documentation	***Bugs in the documentation associated with the code or the content of comments contained in the code***
Integration update	Internal interfaces	*Bugs related to the interfaces between communicating components with the program under test. The components are assumed to have passed their component level tests. In this context, direct or indirect transfer of data or control information via a memory object such as tables, dynamically allocated resources, or files, constitute an internal interface (e. g. Component invocation, Interface parameter, Component invocation return, Invocation in wrong place, Duplicate invocation, ...)*
	External interfaces and timing	*Having to do with external interfaces, such as I/O devices and/or drivers, or other software not operating under the same control structure (e. g. Interrupts, Devices and drivers, I/O timing)*
Manufacturing update	Manufacturing bugs	*Bugs related to the manufacturing process*
Test Case update	Test design bugs	*Bugs in the design of tests*
	Test execution bugs	*Bugs in the execution of tests*
	Test documentation	*Documentation of test case or verification criteria is incorrect or misleading*
	Test case completeness	*Cases required to achieve specified coverage criteria missing*
Update none	None	*None of the proposed types of problem is applicable*

Table E.1 – A second-level typology of software problems