Notice descriptive

Auteur :	TOUZGHAR Soufian
Année :	2012
Etablissement :	Facultés des Sciences et Techniques-Fès
Spécialité :	Systèmes Microélectroniques de Télécommunications et de l'Informatique Industrielle (Cycle Master en Sciences et Techniques)
Etablissement d'accueil :	Zodiac Aerospace Maroc
Cadre du rapport :	Projet de fin d'études

Titre du projet :

Test unitaire des logiciels aéronautiques embarqués : Automatisation selon la norme DO-178B

Encadrement entreprise :

M.DOUKKALI Driss : Chef de groupe Logiciel.

M.AZZAZ Abdellatif : Ingénieur Logiciel.

M.SALAY Ibrahim : Ingénieur Logiciel.

Encadrants pédagogiques :

M. MECHAQRANE Abdellah : Prof. Ens. Sup FST-Fès

M. AHAITOUF Abdelaziz: Prof. Ens. Sup A FP-Taza

Date de début et de fin du stage:

Début : 27/02/2012
Fin : 15/06/2012

Dédicaces

A celle qui a attendu avec patience les fruits de sa bonne éducation…
à ma Mère

A celui qui m'a indiqué la bonne voie en me rappelant que la volonté fait toujours les grands hommes…
à mon Père

Remerciement

Je tiens à remercier dans un premier temps, toute l'équipe pédagogique de la FST de Fès et les intervenants professionnels responsables de la formation Systèmes Micro-électroniques de Télécommunications et Informatique Industrielle, pour avoir assuré le bon déroulement de celle-ci.

Je remercie également mes encadrants pédagogiques : Monsieur Abdellah MECHAQRANE et Monsieur Abdelaziz AHAITOUF, Professeurs de l'enseignement supérieur, pour l'aide et les conseils concernant les missions évoquées dans ce rapport, qu'ils m'ont apportés lors des différents suivis.

Je tiens à remercier tout particulièrement et à témoigner toute ma reconnaissance aux personnes suivantes, pour l'expérience enrichissante et pleine d'intérêt qu'elles m'ont fait vivre durant ces mois de stage :

- Monsieur Driss DOUKKALI**,** manager de groupe ECE-Logiciel à ZAM, pour son accueil et la confiance qu'il m'a accordé dès mon arrivée dans l'entreprise.
- Monsieur M. Abdellatif AZZAZ et M. Ibrahim SALAY: Ingénieurs des systèmes embarqués au sein du groupe ECE-Logiciel, pour m'avoir facilité mon intégration, soutenu et assuré un suivi durant la réalisation du projet.
- Monsieur Yassine BELLEFKIH, développeur Logiciel Embarquée D7/TU, pour m'avoir aidé dans mes premiers pas dans le du développement de l'application.

Résumé

Ce travail rentre dans le cadre du projet de fin d'études que nous devons passer pour évaluer et valoriser notre formation. Il s'est déroulé à la société Zodiac Aerospace Maroc (ZAM), au sein de l'équipe ECE-Logiciel.

L'intitulé du sujet est : "**Tests Unitaires des logiciels aéronautiques embarqués : Automatisation selon la norme DO-178B** ", l'objectif de ce travail est de créer une application qui permet de générer des scripts de tests avec la syntaxe RTR suivant un modèle approprié à ECE-Logiciel. La génération se faisant à partir des plans de tests sous format Excel.

Sommaire

Listes des figures et des tableaux

Abréviations

C

CSC: Computer Software Component

CSU: Computer Software Unit

E

EASA: European Aviation Safety Agency

ERD : Equipment Requirement Document

F

FAA: Federal Aviation Administration

FPGA: Field Programmable Gate Array

L

LLR: Low Level Requirement

P

PTU: Plan de Test Unitaire

Z

ZA: Zodiac Aerospace

ZAM: Zodiac Aerospace Maroc

R

RTCA: Radio Technical Commission for Aeronautics

RTRT: Rational Test Real Time

S

SWRD: SoftWare Requirements Document

T

TU : Test Unitaire

U

UAV: Unmanned Aerial Vehicle

Introduction générale

Les exigences de sécurité imposées par les autorités de certification et les opérations de validation constituent aujourd'hui une des phases les plus délicates du développement de logiciels embarqués sur avion. La sécurité des aéronefs est intimement liée à la qualité des logiciels embarqués.

Pour proposer des produits novateurs et à la pointe de la technologie, le groupe Zodiac Aerospace se recentre sur son cœur de métier, entraînant ainsi une redéfinition globale des relations avec les équipementiers. Ces derniers font appel à l'intégration de l'électronique et de l'informatique dans les différents équipements aéronautiques et spatiaux.

Le Maroc est en cours d'une progression industrielle, le domaine aéronautique ne fait pas de l'exception, la preuve est l'installation de Zodiac Aerospace Maroc (ZAM) depuis janvier 2009. Son secteur d'activité crée une chaine diversifiée de métiers partant de la conception à la fabrication des équipements aéronautiques (assemblage et test des cartes électroniques, développement logiciel et FPGA,…).

Le stage réalisé à ZAM, entre dans le cadre d'améliorer les performances de leurs services avec plus d'options, par la création d'une application permettant d'automatiser une tâche dans les activités de Tests Unitaires (TU).

Ce mémoire est organisé en trois chapitres. Le premier, présente l'organisme d'accueil et ses activités, le deuxième porte sur la présentation de la problématique et de l'environnement du développement de l'application, quant au troisième chapitre, il présente le travail effectué en détail avec les différentes solutions.

Chapitre I : Organisme d'accueil

Résumé :

Ce chapitre sera consacré à la présentation de Zodiac Aerospace, son historique, ses activités et ses filières, en particulier notre lieu de stage ZAM: son organisation, implantation et certaines de ses activités.

I. Présentation de Zodiac Aerospace

Le groupe Zodiac Aerospace est un équipementier aéronautique, sa vocation pour l'aéronautique s'est faite en 1896 avec le développement des ballons dirigeables et des aéroplanes. Dans les années 1930, c'est l'invention du concept du bateau pneumatique qui permet à la société de prendre son essor et qui impose l'image de marque du groupe après la seconde guerre mondiale.

Le groupe est aujourd'hui recentré sur l'aéronautique, après la cession de ses activités marine en septembre 2007.

Zodiac Aerospace poursuit son développement pour assurer une croissance constante de ses activités. Avec une rigueur industrielle et de gestion, les savoir-faire technologiques de Zodiac sont organisés en six branches d'activité dont trois métiers principaux : équipements de sécurité, systèmes aéronautiques et intérieurs de cabines [1].

1. Organisation du groupe

En bonne logique avec son approche de développement sur des créneaux spécifiques et porteurs, les implantations industrielles du groupe couvrent les zones géographiques où se concentrent les principaux constructeurs aéronautiques, notamment l'Amérique du Nord, l'Amérique du Sud et l'Europe [2].

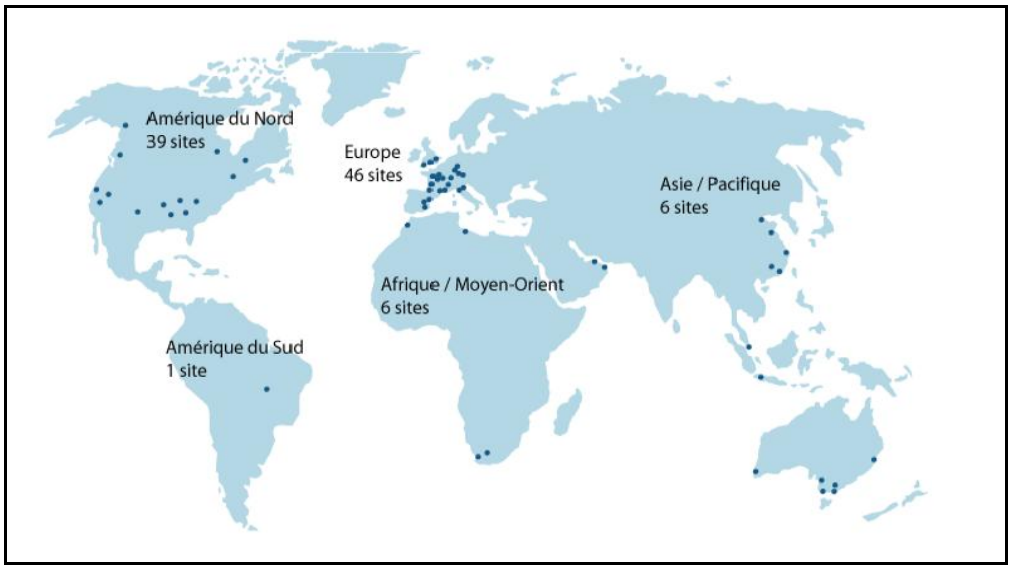


Figure 1: Implantation mondiale du groupe Zodiac Aerospace

Ainsi le groupe peut offrir le meilleur service à la majorité des avionneurs, des compagnies aériennes, des armées, etc...

Le groupe compte 77 sites de production répartis dans le monde et une branche d'activité entièrement dédiée au service et au support clients (Zodiac services), avec 16 centres d'opération répartis entre l'Europe, les Etats-Unis et l'Asie [ci-dessus].

2. Métiers du groupe Zodiac Aerospace

Le groupe Zodiac Aerospace exerce son métier en six branches d'activités:

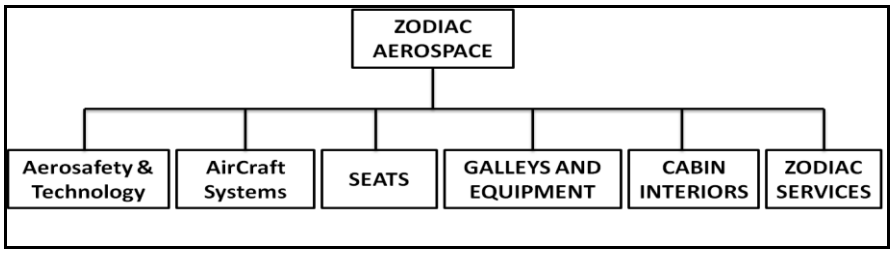


Figure 2: Les branches des activités du groupe Zodiac Aerospace

Aerosafety & Technology :

La branche "Aerosafety & Technology" conçoit et fournit des systèmes complets dédiés au sauvetage et à la protection, ainsi que des solutions technologiques pour des domaines d'applications très variés.

Sa mission a aussi pour vocation d'innover dans le domaine de la télémesure et de la télécommunication avec des produits et systèmes à forte valeur ajoutée, pour les secteurs aéronautique et spatial [3].

	
Siège éjectable	Système de récupération UAV
	
Gilet de sauvetage	Paquetage de survie

Tableau 1 : Quelques produits de Aerosafety & Technology

Aircraft Systems :

La branche "Aircraft Systems" opère dans les parties de circulation du carburant, oxygène et protection physiologique, gestion de la puissance électrique, etc…

Les sociétés de la branche fournissent des équipements fiables et performants aux avionneurs et sont reconnues au plan mondial comme les spécialistes des équipements et des systèmes de haute technologie au service de fonctions essentielles tant en vol qu'au sol.

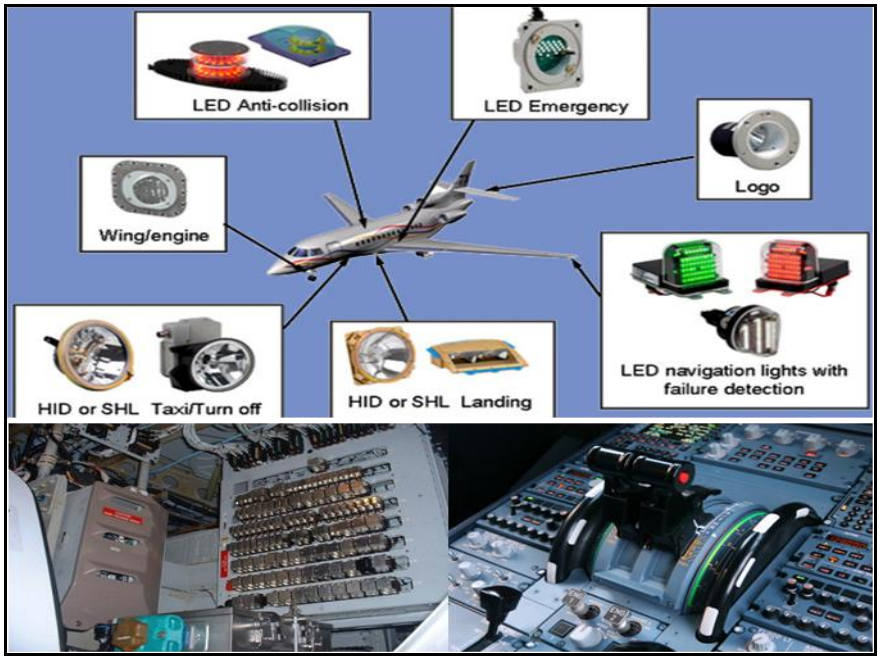


Figure 3 : Quelques produits d'Aircraft Systems

La branche est développée en plusieurs filiales, nous mettons le point sur celles qui opèrent avec Zodiac Aerospace Maroc :

- **Intertechnique *:*** Filiale basée à Plaisir (France), spécialiste en système carburant (distribution et gestion de fuel), surveillance et communication, mesure de la température et de l'humidité...
- **Intertechnique & Avox Systems** : systèmes oxygène…
- **IN-LHC & IN-FLEX** : hydraulique et régulation…
- **PRECILEC** : actionneurs, capteurs & moteurs…
- **ECE:** Distribution et gestion de la puissance électrique, commandes et signalisation dans le cockpit, éclairage, actionneurs,… [3]

Seats :

Cette branche est spécialisée dans la fabrication et la commercialisation de sièges pour les avions civils et les hélicoptères.

Cette branche permet à Zodiac Aerospace d'occuper la première place mondiale dans ce secteur, à travers deux divisions, Siège Europe et Siège US, qui se caractérisent chacune par une offre personnalisée et des gammes très spécifiques suivant les marchés [3].

Figure 4 : Exemple de Seat de la branche Seats

Galley and Equipment :

L'activité galley[1] assure la conception et la production des équipements de galley, en particulier des trolleys[2], et des équipements Cargo[3] destinés aux avions commerciaux [3].

Figure 5 : Quelques produits de Galley and Equipment

Cabin Interiors :

La branche Cabin Interiors intervient principalement dans l'aménagement d'intérieurs de cabine d'avions [3].

Figure 6 : Queleques solutions de la branche Cabin Interiors

[1] Équipements de la cuisine d'avion, housse en cuir, capot vidéo,…
[2] Armoire à roulettes de cuisine.
[3] Containers des avions commerciales

Zodiac Services :

Afin d'améliorer et d'optimiser son savoir-faire tout en restant à l'écoute de ses clients, le groupe a développé une activité service offrant un service après vente sur l'ensemble des produits du groupe [3].

3. Quelques clients de Zodiac

Avions commerciaux		Hélicoptères	
Airbus: A300, A310, A318, A319, A320, A321, A330,		**Agusta Westland:** EH 101, A129, Lynx	
Boeing: MD 11, 737, 757, 777, 787		**AVIC II**: Z-15	
Jets d'affaires		**Avions militaires**	
Bombardier: Global Express, CL300, Learjet 45		**Airbus:** A400M	
Embraer: Legacy 450/500		**BAE:** Nimrod	

Tableau 2: Les principaux clients

II. Présentation de Zodiac Aerospace Maroc (ZAM)

Les leaders mondiaux du secteur aéronaval envisagent une extension massive vers l'étranger. Ils ciblent les pays en cours de développement pour l'implantation de nouvelles unités industrielles. Dans un marché qui est totalement mondial, le Maroc profite de son contexte socio-économique favorable, pour devancer des concurrents régionaux très tenaces.

Plus de 30 ans d'expérience industrielle en aéronautique, proximité géographique et culturelle, disponibilité de compétences et mains d'œuvre hautement qualifiée sont tous des atouts qui font que le Maroc tient bon sur le terrain de compétition.

De nos jours l'enjeu est de taille ; le tissu industriel local continue à s'élargir. Plusieurs projets de développement soutiennent l'attractivité du pays. Nous citons site entre autres :

- Le développement de la sous-traitance (offshoring),

- L'ouverture sur les marchés internationaux,
- L'engagement des établissements de formation dans la qualification des cadres et techniciens du secteur.

Tous ces facteurs font de la création d'une filiale de Zodiac Aerospace au Maroc une vision à des perspectives stratégiques.

1. Installation au Maroc

La société ZAM a réalisé une entité d'études et de production d'équipements aéronautiques au parc industriel Aïn Johra, pour un investissement d'environ 135 millions de dirhams, aux termes d'un contrat dont la signature a eu lieu à Rabat. Le contrat, signé entre le gouvernement marocain et le groupe Zodiac Aerospace, porte sur la création au parc Aïn Johra, avec une unité d'usinage des pièces mécaniques de précision sur des machines à commande numérique à Tiflet, et d'un bureau d'études de définition d'équipements aéronautiques qui se situe à Technopolis de Rabat [4].

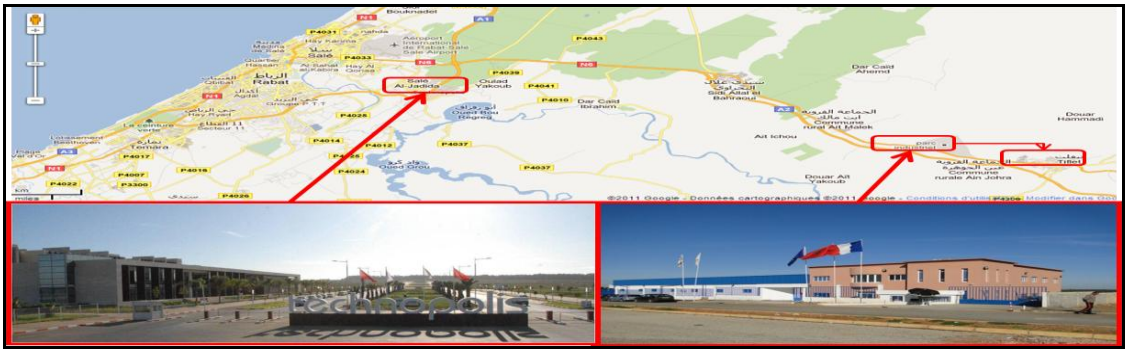


Figure 7 : Situation géographique de ZAM au Maroc

1.1. Site de Technopolis

Situé au sein de Technopolis-Rabat qui comporte des équipes d'ingénierie réparties sur deux plateaux : un pour les logiciels et FPGA, et l'autre pour la conception des parties mécaniques ou électroniques, le calcul et la simulation [5].

Figure 8 : les deux plateaux de ZAM au sein de technopolis-Rabat

1.2. Site de Tiflet

Situé au parc Aïn Johra, il se charge de l'usinage des parties mécaniques, simulation et le montage des pièces fabriquées. Ce site comporte un atelier électronique nommé **Firnas** comme l'un des précurseurs de l'aéronautique « **Abbas Ibn Firnas** » qui se charge du test, de l'assemblage des cartes électroniques et du câblage de quelques parties électriques des pièces usinées etc… [5]

Figure 9 : Atelier électronique et usinage de ZAM

2. Organisation de ZAM

La société ZAM, fait partie de la branche AIRCRAFT SYSTEMS. Elle s'organise en 3 métiers comme le montre la Figure 10, à savoir :

- Production Mécanique,
- Production Electronique
- Pôle ingénierie& bureau d'étude.

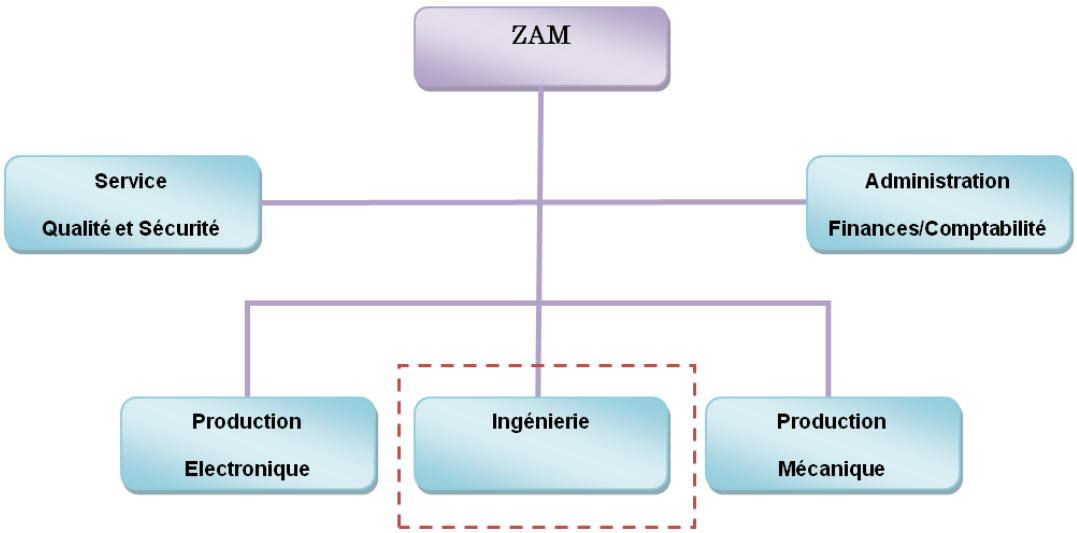


Figure 10 : Organigramme de ZAM

Présentation du pôle ingénierie & bureau d'études

Le département ingénierie est organisé en 2 secteurs d'activité.

- L'ingénierie: Regroupe les équipes de développement et test des logiciels et des FPGA.
- Bureau d'études: Inclut toutes les équipes qui font le design des parties mécaniques ou électroniques, le calcul et la simulation.

L'ingénierie est constituée principalement de trois groupes de travail :

- D7 SW Team.
- ECE SW Team.
- ECE FPGA Team [5].

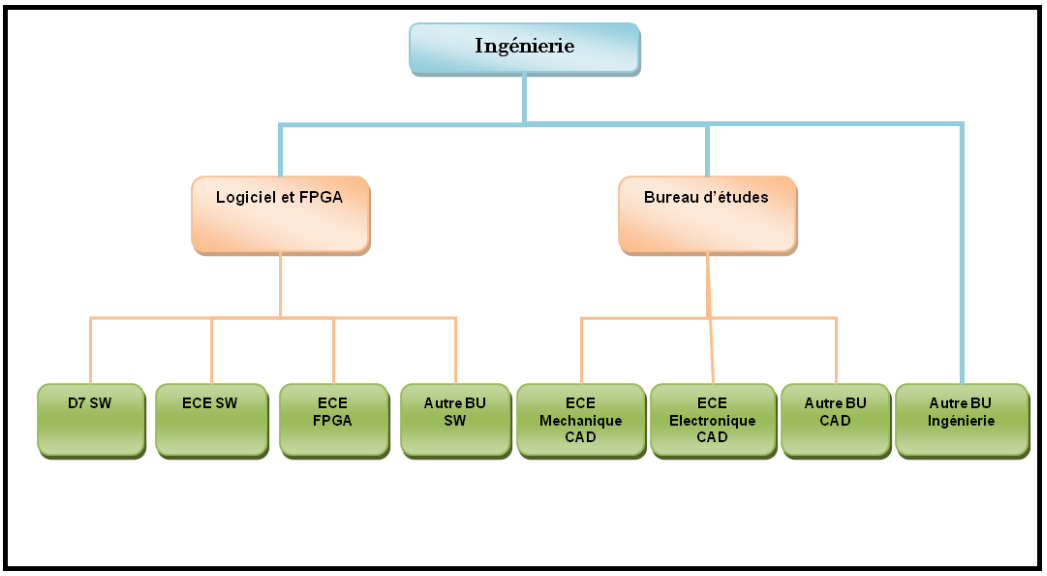
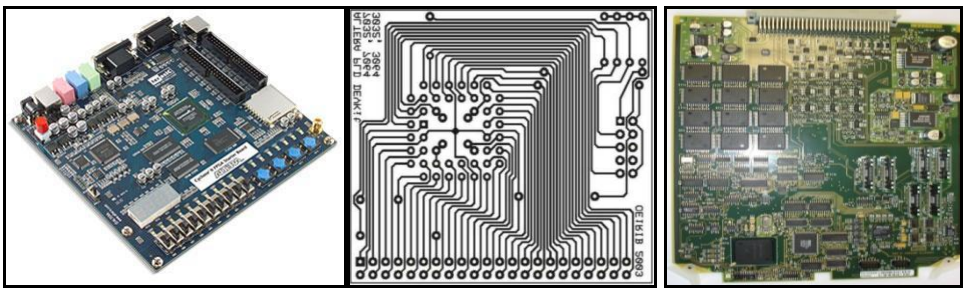


Figure 11 : Organigramme de l'ingénierie & bureau d'études

3. Activités

ZAM est une entreprise en pleine progression au Maroc, elle opère dans le développement et la vérification de logiciel embarqué, le développement en C sur microcontrôleurs, Le développement de FPGA prend aussi une grande part d'importance. Sans oublier la conception électronique qui se manifeste par la conception, schéma câblage … [5]



(1) **(2)** **(3)**

Figure 12 : Quelques activités de ZAM, (1) : FPGA, (2) : schéma, (3) : vérification de logiciel embarqué

Conclusion

Zodiac Aerospace poursuit sa stratégie fondée sur la croissance externe. Quatre grandes constantes guident Zodiac Aerospace dans son évolution:

- La croissance régulière de son bénéfice par action
- La diversification dans des métiers à fort contenu technologique
- La sélection de marchés porteurs susceptibles d'assurer des relais de croissance
- Le leadership mondial dans les principaux créneaux

ZAM suit aussi la stratégie de son entreprise mère, et se lance dans un nouveau territoire en pleine progression pour acquérir de nouveaux marchés par la diversité de ces activités. Le chapitre suivant va présenter la problématique du sujet de stage et l'environnement de développement de l'application.

Chapitre II : Problématique et environnement de développement

Résumé :

Ce chapitre a pour but de présenter la problématique du sujet, et aussi de présenter quelques étapes de développement logiciel en aéronautique : La norme de programmation en métier aéronautique, le cycle de développement en V et le test unitaire.

A. Problématique du sujet de stage

L'intitulé du sujet est : "**Tests unitaires des logiciels aéronautiques embarqués : Automatisation selon la norme DO-178B** ". En d'autres termes, cela signifie que nous devons mettre en œuvre une application pour l'équipe chargée de tests unitaires au sein de ZAM, qui lui permettra d'automatiser la tâche de génération des scripts de tests.

Cette application a été développée dans un environnement PERL (Practical Extraction and Report Language) afin de générer un script de test avec une syntaxe RTR[4]. Ce script sera ensuite utilisé comme point d'entrée pour l'outil Rational Test Real Time (RTRT) dans le but de tester une fonctionnalité du logiciel embarqué.

Notre mission consiste alors à développer une application permettant l'automatisation du développement des scripts de test. L'ancienne méthode était manuelle. La migration vers notre application va entraîner le basculement vers une nouvelle version de processus. Un panel de technologies et d'outils ont alors été étudiés et utilisés pour réaliser ce projet.

Notre point de départ est un plan de test sous format Excel, duquel nous générons, d'une façon automatique, un script de test (en respectant la syntaxe du langage RTR) qui va être exécuté dans l'outil de test RTRT.

Le code doit être modulaire, écrit sous forme de fonctions qui vont être appelées dans un programme principal, pour que le code soit simple à entretenir et modifiable sans devoir le refaire en entier.

B. Environnement de développement de l'application

I. Logiciel en aéronautique

1. Définitions

Un logiciel embarqué monté sur un aéronef (avion, hélicoptère, drone) permet de réaliser une tâche spécifique. Il est constitué d'un ensemble de capteurs, afficheurs, moteurs, etc.…, reliés à un boîtier appelé calculateur.

[4] La syntaxe reconnue par le logiciel RTRT (Rational Test Real Time).

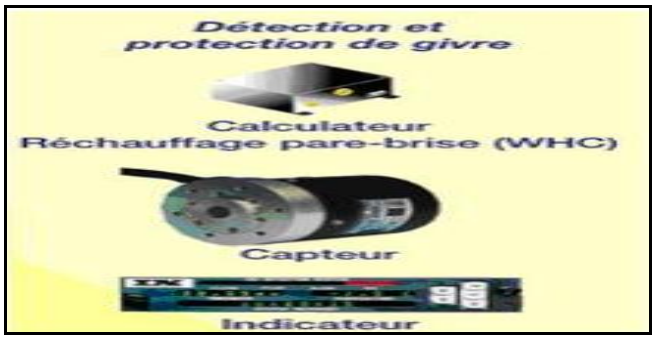


Figure 13 : Exemple de systèmes embarqués développés par le groupe Zodiac

2. Caractéristiques

L'informatique embarquée a des contraintes qui diffèrent de l'informatique personnelle (les micro-ordinateurs). Ce sont principalement :

2.1. La criticité

Les systèmes embarqués sont souvent critiques. En effet, comme un tel système agit sur un environnement physique, les actions qu'il effectue sont irrémédiables. Le degré de criticité est fonction des conséquences des déviations par rapport à un comportement nominal ; conséquences qui peuvent concerner la sûreté des personnes et des biens, la sécurité, l'accomplissement des missions et la rentabilité économique.

2.2. La réactivité

Ces systèmes doivent interagir avec leur environnement à une vitesse qui est imposée par ce dernier. Ceci induit donc des impératifs de temps de réponses. C'est pour cette raison que l'informatique embarquée est souvent basée sur un système temps réel.

2.3. L'autonomie

Les systèmes embarqués doivent en général être autonomes, c'est-à-dire remplir leur mission pendant de longues périodes sans intervention humaine. Cette autonomie est nécessaire lorsque l'intervention humaine est impossible, mais aussi lorsque la réaction humaine est trop lente ou insuffisamment fiable.

2.4. La robustesse

L'environnement est souvent hostile, pour des raisons physiques (chocs, variations de température, impact d'ions lourds dans les systèmes spatiaux, ...) ou humaines (malveillance). C'est pour cela que la sécurité au sens de la résistance aux malveillances et la fiabilité au sens continuité de service, sont souvent rattachées à la problématique des systèmes embarqués.

II. Le cycle de développement en V

1. Introduction

Le domaine « Systèmes Embarqués » fait apparaître la nécessité d'une analyse spécifique des systèmes complexes et critiques, répondant à des exigences de qualité et de sûreté de fonctionnement très strictes, et propose des méthodes et outils de conception pour y parvenir.

Le cycle en V [Figure 14] est devenu un standard de l'industrie logicielle depuis les années 1980 et depuis l'apparition de l'ingénierie des systèmes. Il est devenu un standard conceptuel dans tous les domaines de l'industrie. Le monde du logiciel ayant, de fait, pris un peu d'avance en termes de maturité, nous trouvons dans la bibliographie courante souvent des références au monde du logiciel qui pourront s'appliquer au système [9].

Chacune des étapes est une phase transitoire dans la vie du logiciel, elle permet d'effectuer une tâche bien déterminée.

2. Les phases du cycle en V

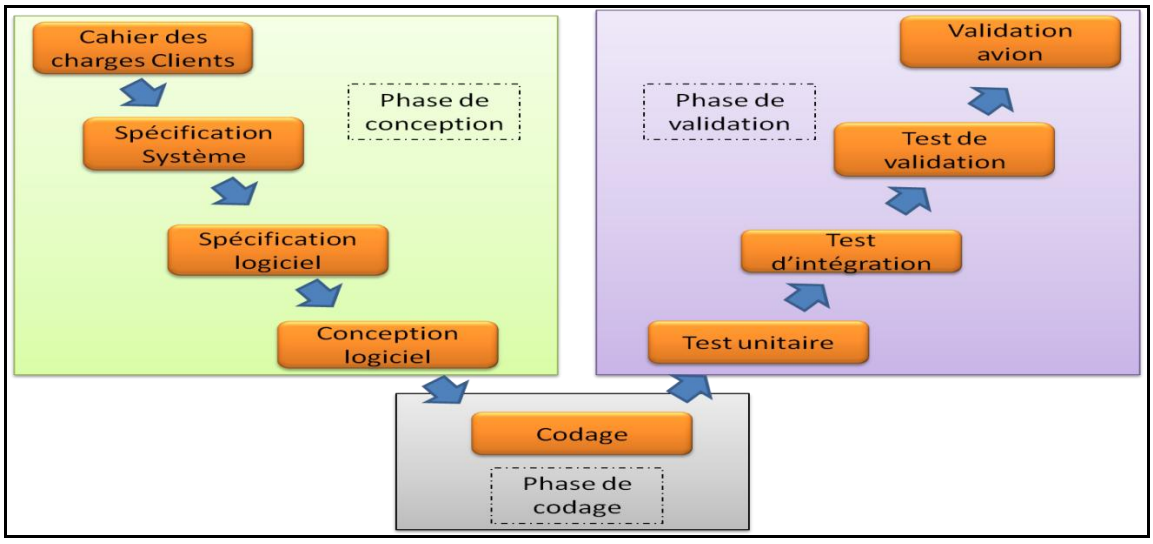


Figure 14 : Modèle de cycle en V

Comme nous pouvons le voir sur la Figure 14, le modèle est constitué de trois grandes phases :

- Phase descendante ou phase de conception
- Phase de réalisation ou de codage
- Phase de validation ou ascendante.

2.1. Phase de conception

Le besoin d'étude et de faisabilité (Cahier des charges) : C'est le point de départ du cycle. Cette étape reflète les besoins du client. Le cahier des charges, rédigé par le client, décrit l'ensemble des besoins fonctionnels attendus du système. Il permet une meilleure compréhension du système et la structuration des besoins du client.

Spécification (système + logiciel) : Les spécifications reprennent en détail les éléments du cahier des charges. Cette étape décrit de façon exhaustive ses exigences.

Conception détaillée : Chaque algorithme spécifié dans l'étape précédente est détaillé pour permettre à un programmeur de coder des algorithmes justes en lisant la conception détaillée. A cette étape, est initiée la phase des tests unitaires qui permettront plus tard de prouver l'absence de bugs[5].

2.2. Phase de codage

Codage : Le codage ou développement informatique, est la transcription en langage interprétable par un compilateur de la conception détaillée avec des langages comme JAVA, C++, PHP…etc.

La fin du codage ne signifie pas la fin du projet. Il reste souvent un ensemble de dysfonctionnement ou bugs qu'il est nécessaire de détecter et corriger. La phase de test est là pour supprimer autant que possible les dysfonctionnements du codage.

2.3. Phase de validation

Tests unitaires : Les tests unitaires permettent de vérifier qu'il n'y a aucune erreur entre la transcription de la conception générale et le code.

[5] Un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement

Tests d'intégration : Cette étape permet de vérifier qu'il n'existe pas d'erreurs entre la conception générale et la conception détaillée.

Validation et Maintenance : Monter au client que le logiciel décrit dans le cahier des charges est bien en accord avec le produit final.

3. Avantage du cycle en V

L'avantage du modèle est qu'il est un excellent support à la formalisation des relations entre le client et l'équipe de développement. La phase de spécification permet à l'équipe de vérifier que la demande du client a été bien comprise. Le client valide généralement la spécification. La vérification et la validation évite les retours arrière. En effet, elle oblige le client à réfléchir aux différents aspects de sa demande.

4. Bases documentaires par phase

Pour une bonne communication entre les différents partenaires du projet, il est nécessaire d'établir un ensemble de documents qui décrivent les différentes phases du cycle en V [9] :

Besoins et Faisabilité	Spécification	Conception Architecturale	Conception Détaillée	Codage	Test unitaire	Test d'intégration	Test de Validation	Recette
Spécification des Besoins Utilisateur Cahier des charges								**Procès Verbal de Validation**
	Spécifications Générales Spécification Technique des Besoins						**Procès Verbal de Validation**	
		Dossier de Définition du Logiciel Dossier d'Architecture Technique				**Rapport de Tests d'Intégration**		
			Rapport de Conception Détaillée LLR		**Rapport de Tests Unitaires**			
				Code source				

Tableau 3 : Les documents générés après chaque étape du cycle en V

5. Classification des tests

Il existe différentes façons de classer les tests. Cependant, il est possible de les regrouper selon :

- **Leur mode d'exécution :** manuel et automatique.
- **Leurs modalités :** statique et dynamique.
- **Leurs méthodes :** Structurelles (Boîte blanche) et Fonctionnelles (Boîte noire).
- **Leurs niveaux :** Tests unitaires, Tests Système, Tests d'intégration, Tests de non régression.
- **Leurs caractéristiques :** performances et de robustesses.

Dans la partie qui suit, nous allons nous focaliser sur le niveau du test et plus précisément le test unitaire, puisque c'est à ce niveau où se situe notre projet [10].

6. Tests unitaires

6.1. Introduction

Les tests unitaires sont effectués avant la phase de test du logiciel en son ensemble (test système). Ils sont une activité vitale à la production d'une application robuste et sans erreur puisqu'ils permettent au testeur de facilement stimuler toutes les fonctions de bas niveau de l'application et de démontrer que les exigences de base ont été implémentées complètement et correctement [11].

6.2. Importance du test unitaire

Le coût d'une erreur augmente de façon exponentielle. Il faut donc mettre en place les méthodes et outils de tests le plus tôt possible dans le cycle de développement.

6.3. La conception du test

S'affranchir de l'environnement projet pose deux problèmes : comment piloter le module à tester ? Et comment faire lorsque le module appelle d'autres fonctions ?.

Pour pouvoir simuler le fonctionnement du module comme s'il était intégré au projet, nous créons un "Driver", le pilote du module, et un "Stub" simulant les appels de fonctions [8].

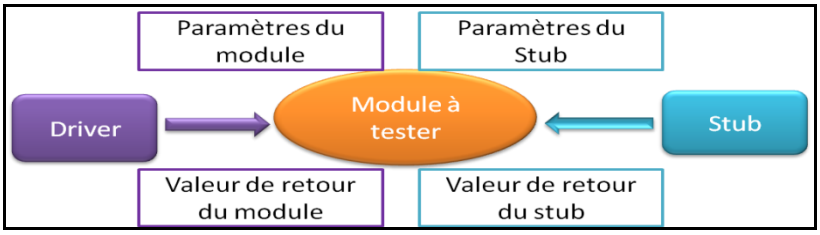


Figure 15 : Conception générale de test unitaire

Le driver :

Il permet de :

- ✈ lancer une série de tests avec des paramètres de valeurs différentes.
- ✈ comparer les résultats obtenus avec ceux attendus [8].

Les Stubs :

Les "Stub" permettent de simuler les appels de fonctions réalisés par le module. Pour chaque appel, le Stub vérifie que les paramètres lui ont été correctement transmis et renvoie un résultat défini à l'avance [8].

6.4. Exécution sur cible

Les organisations qui créent des applications dans le domaine de l'embarqué font face à un défi additionnel : s'assurer que des erreurs ne sont pas introduites dans le logiciel par le compilateur croisé ou en raison de différences d'architecture entre l'hôte et la cible.

III. La norme de programmation aéronautique : DO-178B

1. Introduction

Le document DO-178B de l'organisme privé « *Requirements and Technical Concepts for Aviation* » (RTCA), indique que la qualification d'un outil logiciel est nécessaire quand des activités prescrites par la DO-178B sont éliminées, réduites ou automatisées par cet outil sans que le résultat qu'il génère ne soit contre-vérifié.

Bien que la plupart des méthodes et moyens de tests soient laissés à la discrétion de chaque société, les domaines critiques (aéronautique, automobile, nucléaire,...) sont encadrés par des normes de programmation. Ces normes sont édictées par des consortiums regroupant des industriels et des Etats. Pour veiller à leur application, des systèmes de contrôles peuvent être créés, comme pour les équipements aéronautiques [6].

2. Application de la norme : DO-178B

La DO-178B est utilisée en aéronautique civile pour les logiciels intégrés dans les systèmes et équipements de bord. Nous pouvons cependant l'utiliser comme référence de certification pour des avions militaires [7].

3. Degré de défaillance d'un logiciel aéronautique

Sur l'échelle de gravité, la norme DO-178B définit les degrés de défaillances suivants:

- Catastrophique
- Dangereux
- Majeur
- Mineur

3.1. Catastrophique

Dans le cas où les conditions de panne sont susceptibles d'empêcher la poursuite en toute sécurité d'un vol et d'un atterrissage [7].

3.2. Dangereux

Les conditions de panne sont susceptibles de réduire les possibilités de l'aéronef ou la capacité de l'équipage à faire face à des conditions hostiles:

- Réduction importante des marges de sécurité ou des capacités fonctionnelles.
- Problèmes physiques ou accroissement de charge de travail ou des effets négatifs sur les occupants [7].

3.3. Majeur

Conditions de panne d'une gravité se traduisant par une réduction significative des marges de sécurité, un accroissement significatif de la charge de travail de l'équipage ou des conditions affectant son efficacité, ou un inconfort pour les occupants, comportant éventuellement des blessures [7].

3.4. Mineur

Conditions de panne n'engendrant pas de réduction significative de la sécurité de l'aéronef et susceptibles d'entraîner pour l'équipage des actions se situant tout à fait dans le domaine de leurs capacités.

Gravité	Probabilité			
	Probable 10^{-5} / h	Rare 10^{-6}/ h	Extrêmement rare 10^{-7}/ h	Extrêmement improbable 10^{-9}/ h
Mineure				
Majeure	X			
Dangereuse	X	X		
Catastrophique	X	X	X	

Tableau 4 : Relation Gravité/ Probabilité

4. Les organismes de certification

Les organismes de certification sont financés et constitués par les Etats et les avionneurs. Ils sont chargés d'édicter les règles de conception et de développement pour que les systèmes soient conformes aux exigences.

Ils auditent les équipementiers et les avionneurs tout au long de leur cycle de développement pour vérifier s'ils sont conformes aux normes. Les systèmes sont validés lors d'un audit final appelé « audit de certification ». Ce n'est qu'avec l'accord de ces organismes qu'un système aura le droit d'être embarqué et qu'un avion pourra être commercialisé (certificat de navigabilité).

Lors du développement d'un logiciel embarqué aéronautique, trois entités sont présentes :

- Le systémier ou équipementier (ZODIAC AEROSPACE)
- L'avionneur (Le client : AIRBUS, EMBRAER…)
- Les autorités de certification (EASA, FAA…) [8].

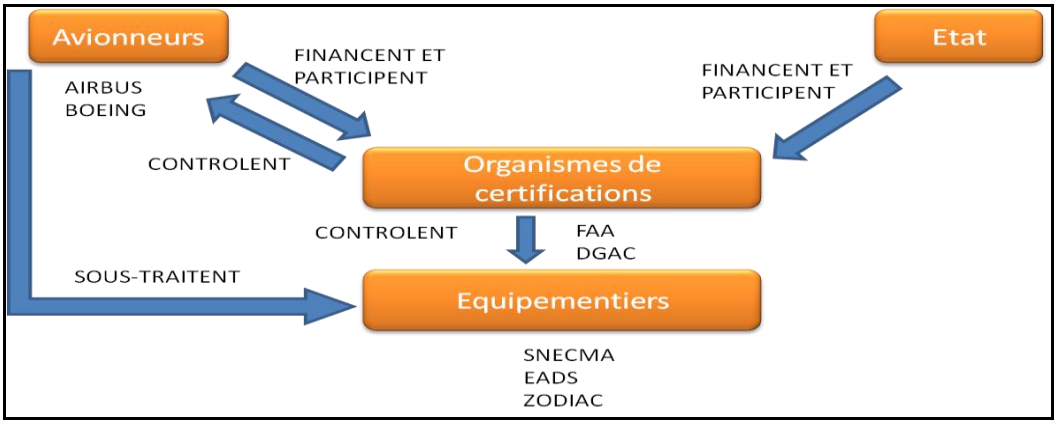


Figure 16 : Les organismes de certification

5.1. Les avionneurs

Les avionneurs conçoivent et produisent des avions. Ils spécifient les besoins pour les différents systèmes qui seront intégrés dans l'avion (commandes de vol, moteurs…). Le développement est sous-traité à un équipementier sélectionné sur appel d'offre [8].

5.2. Les équipementiers

Les équipementiers développent les systèmes embarqués. Ils ont l'obligation d'appliquer tout au long de la conception la norme DO-178B [8].

5.3. Les Etats

Les Etats promulguent les lois nécessaires au contrôle du trafic aérien et à ses effets :

- régulation du trafique aérien,
- normes de sécurités pour les passagers et équipages,
- réduction des nuisances directement liées au trafic aérien (ex : réduction des nuisances sonores) [8].

IV. Outils de développement

1. Perl (Practical Extraction and Report Language)

1.1. Définition

Perl est un langage de programmation créé par Larry Wall en 1987, reprenant des fonctionnalités du langage C et des langages de scripts « sed, awk et shell (sh)… ». C'est un langage interprété, polyvalent et particulièrement adapté au traitement et à la manipulation de fichiers texte.

1.2. L'intégration

Les programmes Perl sont intégralement portables entre GNU/Linux, Mac OS X (ou autre UNIX) et Windows malgré les désignations de fichiers différentes de ces systèmes. Perl permet l'usage du moteur d'interfaces graphiques Tk pour effectuer des entrées-sorties conformes à l'état de l'art [12].

1.3. Syntaxe de Perl : Voir [Annexe 1]

2. RTRT (Rational Test Real Time)

2.1. Introduction

Rational Test Real Time (RTRT) a été initialement développé par Testware ATOLL. Le produit est arrivé d'abord à Rational Software et enfin à IBM. RTRT est une solution complète pour tester et observer des systèmes embarqués en temps réel. Elle prend en charge la couverture du code, la détection des fuites de mémoire et de profilage des performances. L'outil RTRT est souvent utilisé dans des projets qui sont classés comme «mission critique». L'outil effectue une analyse module par module. Il crée alors un squelette à compléter avec les différents cas de tests. Il génère à partir de ce squelette le driver et les stubs, compile le tout avec le module et lance le simulateur avec lequel il est interfacé. Enfin, il récupère les résultats d'exécution et génère un rapport. La Figure 17 montre le fonctionnement général de RTRT [8].

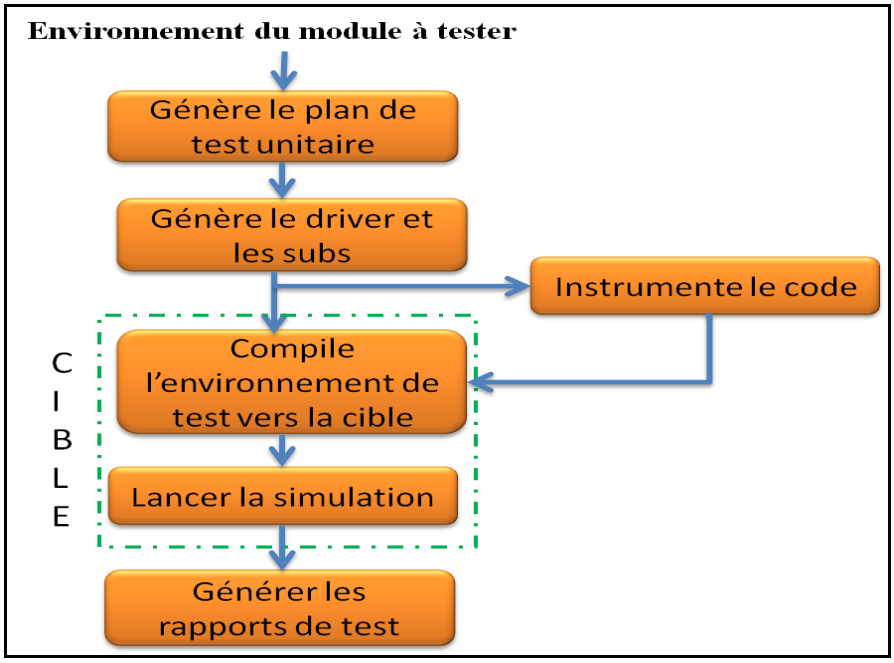


Figure 17 : Schéma de fonctionnement général du RTRT

2.2.1. Génération du plan de test unitaire

A partir du fichier source, un squelette est généré. Ce squelette permet de créer les cas de test beaucoup plus aisément [8].

2.2.2. Génération du driver et des stubs

Le driver et les stubs sont générés, en langage C, à partir des informations saisies dans le plan de test unitaire [8].

2.2.3. Instrumentation du code

Pour pouvoir vérifier que toutes les branches des modules sont couvertes, le logiciel crée un nouveau fichier source à partir du module à tester. Ce nouveau fichier contient des « flags » sur chacune des branches. Lors de la simulation, les drapeaux sont levés lorsque la branche est parcourue. Nous pouvons donc ainsi aisément détecter si une branche est non couverte.

L'exécution de test sous l'outil RTRT fonctionne comme suit :

Exécution non instrumentée

- Utilise le fichier objet (.o).
- Traduit les tests du langage RTR vers le langage C puis compile.
- Compile les librairies de RTRT et fait la liaison des éléments de test pour construire le projet.
- Lance la simulation dans le simulateur (niveau B ou C[6]) ou dans la cible pour niveau A via l'outil BDM (Background Debug Mode).
- Crée un rapport temporaire après l'exécution des tests [16].

Exécution instrumentée

- Instrumente le code source en fonction du niveau du logiciel.
- Instrumente le code source selon le niveau du logiciel (A, B ou C).
- Compile le fichier source instrumenté.
- Traduit les tests du langage RTR vers le langage C puis compile.

[6] Niveaux de la criticité des logiciels aéronautiques embarqués

- Compile les librairies de RTRT et fait la liaison des éléments de test pour construire le projet.
- Lance la simulation dans le simulateur (niveau B ou C) ou dans la cible pour le niveau A via l'outil BDM.
- Crée un rapport temporaire après l'exécution des tests.

RTRT compare les deux résultats d'exécution (instrumenté et non instrumenté) afin de vérifier que l'instrumentation du code source n'affecte pas le comportement du module [16].

2.2.4. Compilation de l'environnement de test vers la cible

Le driver et les stubs sont compilés vers la cible, puis sont liés au module (.o) d'origine et le module instrumenté. A partir de ce moment, nous quittons l'environnement traditionnel PC, pour l'environnement cible [8].

2.2.5. Lancement de la simulation

Un simulateur ou émulateur est piloté par le logiciel. Ceci permet de tester le module dans son environnement cible et de pouvoir détecter d'éventuelles erreurs comme les accès à des mémoires, les exceptions du processeur....

2.2.6. Génération du rapport de test

A partir des résultats obtenus lors des deux simulations, un rapport est créé. Il est constitué d'un rapport d'analyse et d'un rapport de couverture.

2.2. Points forts / Points faibles

- Nous pouvons réellement tester la fonctionnalité des modules.
- Nous testons le module directement dans son environnement cible.
- La préparation des tests est longue et fastidieuse.

Conclusion

Le développement du logiciel dans un domaine critique tel que l'aéronautique, nécessite des méthodes de vérifications afin de s'assurer qu'un produit répond aux exigences de ce domaine.

Notre application est développée par l'outil PERL qui facilite la manipulation et le traitement des fichiers Excel.

Dans le chapitre suivant, nous allons voir le fonctionnement de l'application en détail.

Chapitre III : Conception et réalisation de l’application d'automatisation des scripts de test

Résumé :

Ce chapitre sera consacré à la présentation du sujet de stage, puis aux différentes démarches suivies pour réaliser l’application en définissant son entrée et le fichier généré par cette dernière.

1. Présentation du sujet du stage

1.1. Vue globale du projet

Après la réception du projet à tester (Module Under-test), la mission principale du testeur est de vérifier que le code source développé répond au cahier des charges (spécifications et besoins). Pour s'éloigner du code source et éviter de tendre vers la vision du développeur (afin de pouvoir détecter le maximum possible des erreurs et défauts), le testeur reçoit un document LLR[7] qui décrit le fonctionnement du module et présente les fonctions appelées et un « data dictionnary » qui contient les noms des variables utilisées dans le Logiciel.

Le projet est reçu sous la forme de dossier « CSC » qui se compose de plusieurs modules à tester « CSU ». Le test commence par la préparation de l'environnement du test en créant 3 dossiers [Figure 18]:

- **Procedure** : où s'exécute le test.
- **Script :** où se développent les scripts de test.
- **Report** : où se génère le rapport du test.

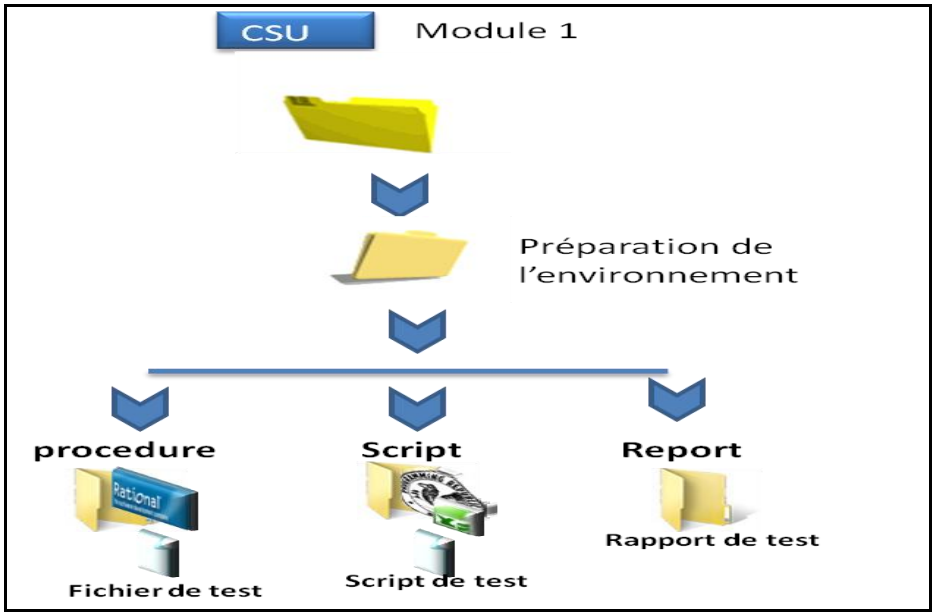


Figure 18: Préparation de l'environnement de test

[7] Low-Level Requirements

Après la préparation de l'environnement de test, il vient le développement des scripts de test au niveau du dossier « *Scripts* ». C'est à ce niveau que se situe l'application développée.

Comme mentionné auparavant, notre application se charge de développer les scripts de test d'une façon automatique à partir des plans de test. Après l'établissement des scripts de test, nous utilisons l'outil de test RTRT pour voir si le code correspond aux LLR et si le test passe sans anomalies (tout le code a été couvert,….). Cette opération se situe au niveau du dossier « *Procedure* ».

La génération du résultat du test se fait au niveau du dossier « *Report* ». Dans le cas où le test échoue, c'est-à-dire qu'une erreur se produit au moment d'exécution du test, le problème peut résider soit au niveau du script de test, et c'est au testeur de le corriger, soit le problème réside au niveau du code ou LLR, le testeur se limite alors à une déclaration de l'erreur afin qu'elle soit corrigée. La Figure 19 présente le processus de TU avec plus de détails.

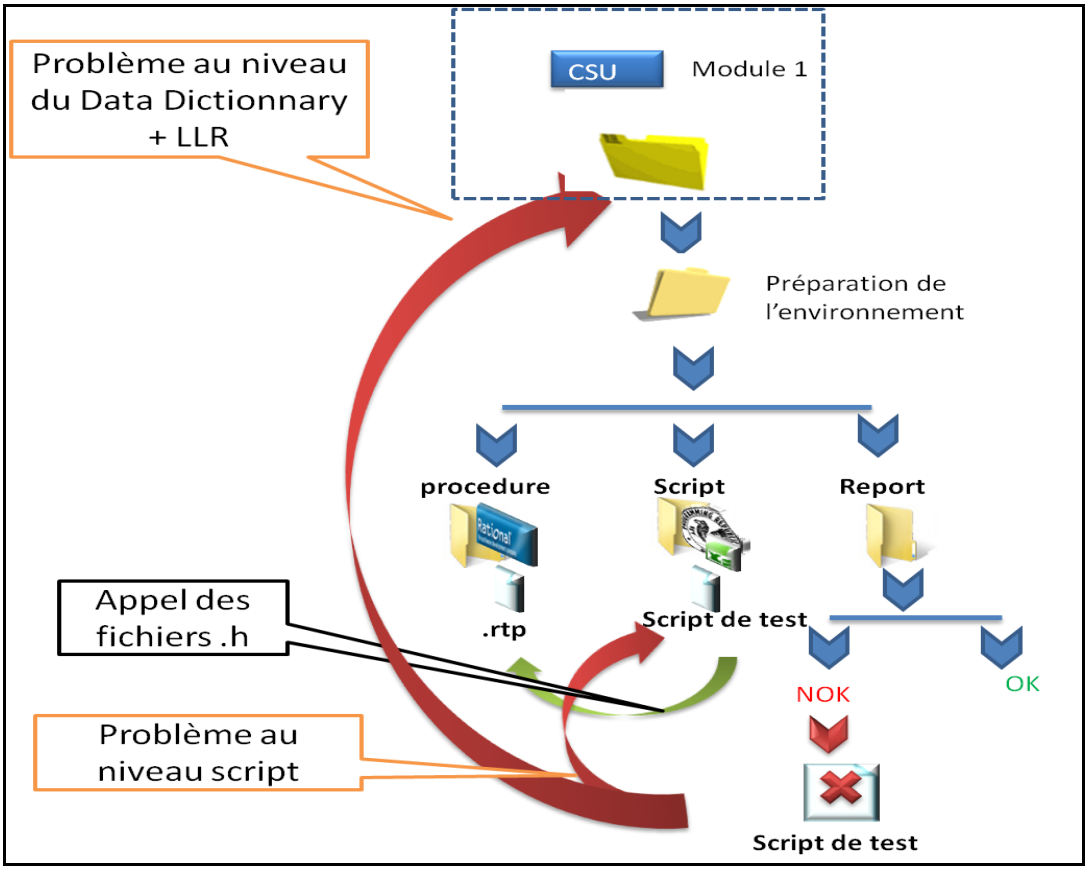


Figure 19 : Chaine de TU au sein de ZAM pour chaque module à tester

1.2. Modélisation du besoin

Le besoin de clarté au moment de développement des scripts de test et la cohérence avec l'art du métier des TU qui se manifeste par l'éloignement du testeur du code source, ainsi que la contrainte du temps qui est un facteur primordial dans tous les processus de l'entreprise sont tous derrière la création de cette application.

L'application développée est située au cœur du processus des TU. Elle est alimentée par des plans de test sous format Excel en entrée. Le plan de test est standard, il a la même forme pour faciliter l'automatisation. L'objectif est de pouvoir accéder au plan de test, d'extraire toutes les informations utiles pour le script de test et de traiter ces informations de manière à obtenir un script qui respecte la syntaxe RTR et qui sera exécutable sur le l'outil RTRT. La Figure 20 montre la situation de l'application et la tendance vers le nouveau processus.

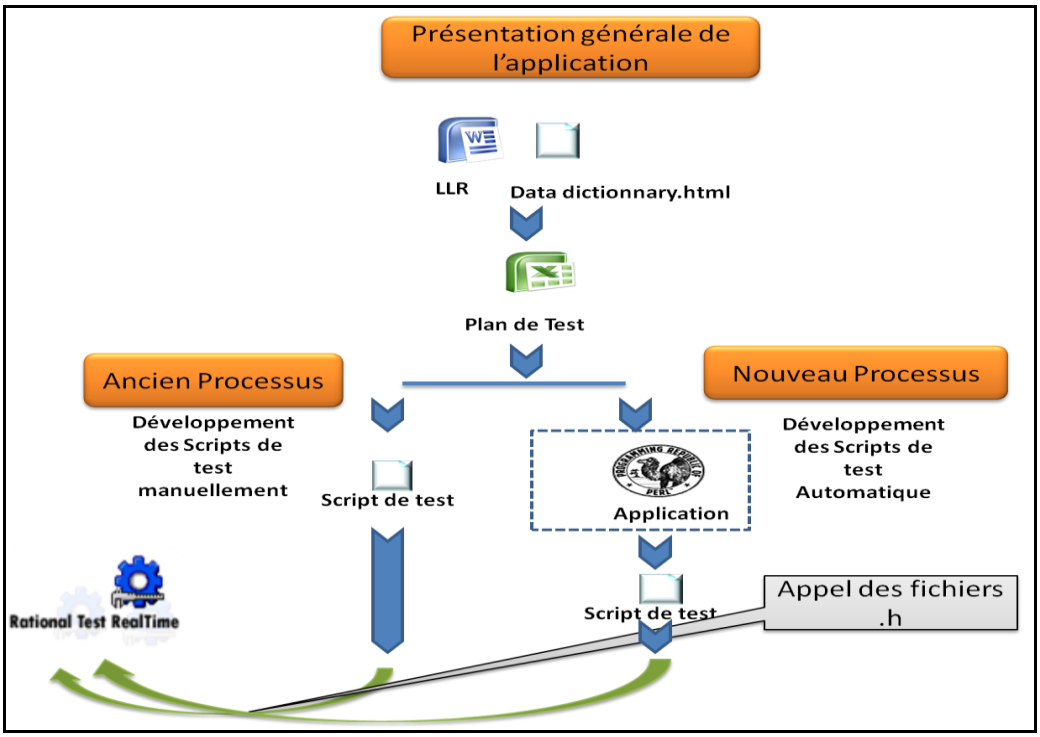


Figure 20 : Situations de l'application dans le processus de test

2. Description de l'entrée de l'application

Pour tester un module (CSU), un plan de test est élaboré. Ce plan doit contenir toutes les informations nécessaires pour réaliser le script de test. C'est un fichier sous format Excel, composé essentiellement de 4 pages : Equivalence Class Analysis, Test Cases, Functional Coverage et Script Appendix.

Chaque feuille Excel a sa propre utilisation. Il n'y a pas de redondance d'information et chaque information écrite sur le fichier Excel est essentielle pour générer le script de test.

Le nom du ficher Excel est standard, il commence par le nom du CSC (nom du projet), après on met un tiret, puis le nom du module, suivi d'un « _STD –UT ». Exemple : « CSU_NAME_STD –UT .xls ».

La structure du fichier Excel aussi est standard. Ainsi le nom, le nombre de pages, les titres de chaque rubrique et l'écriture de chaque rubrique sont standards. Nous avons à la fin un prototype de fichier Excel, que chaque testeur doit respecter et il va remplir juste les valeurs qui ont un rapport avec son test.

La standardisation du format du fichier Excel vient comme nécessité pour l'automatisation du processus de la génération du script de test. Un Template a été réalisé, et chaque testeur en dispose pour écrire son plan de test.

La version du fichier Excel n'est pas importante car l'application doit prendre en considération tout format Excel soit « .xls » ou « .xlsx ».

1.1. Composition du ficher Excel

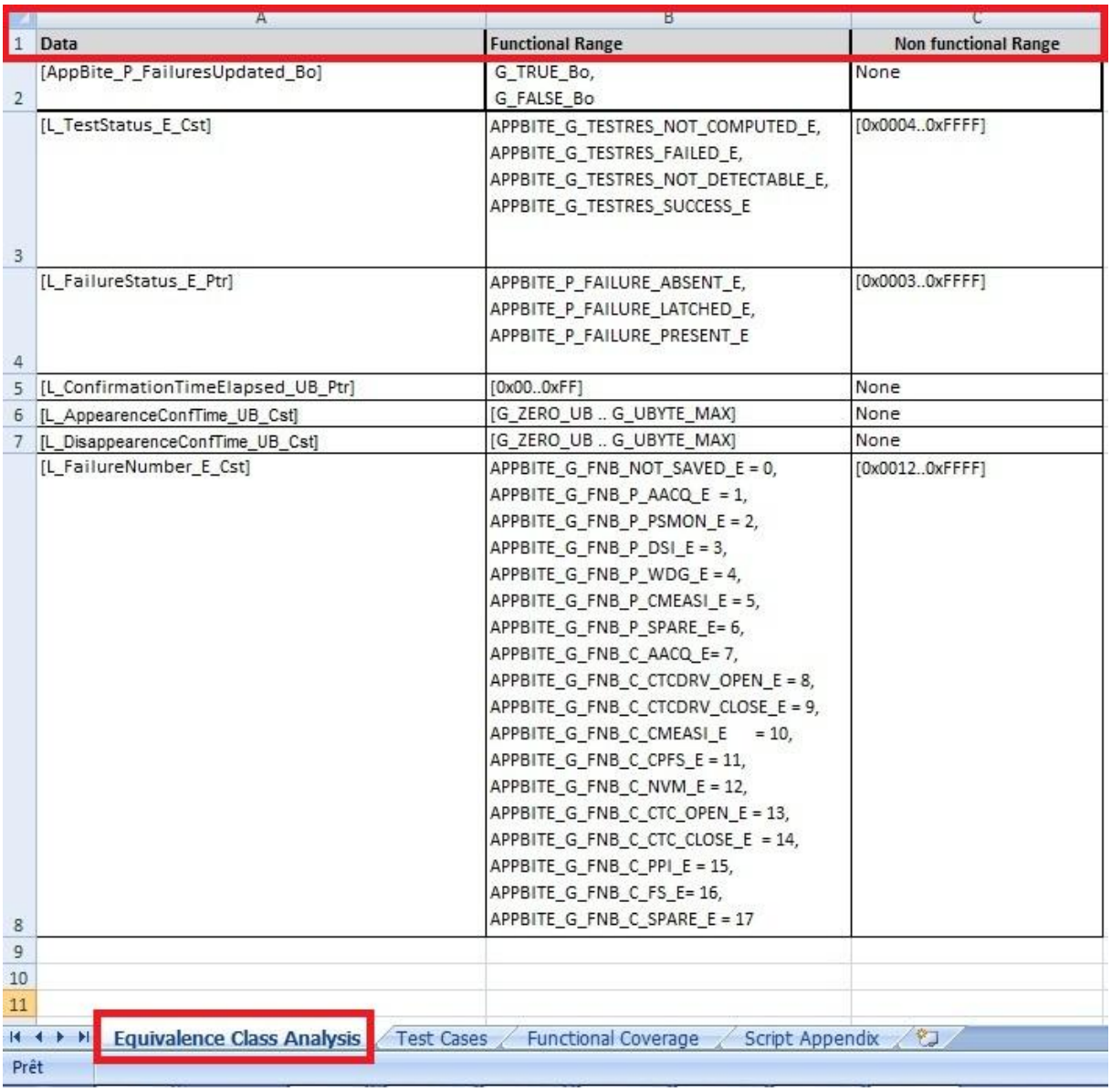
Comme cité ci-dessus, le fichier Excel se compose de 4 pages essentielles, mais il pourrait contenir une page en plus. Celle-ci n'est pas nécessaire pour générer le plan de test. Elle est propre au testeur et pourrait l'aider pour faciliter la revue[8]. Dans ce qui suit, nous allons décrire chaqu'une des pages de notre fichier Excel.

2.2.1. Equivalence Class Analysis :

Elle se compose de trois colonnes [Figure 21]:

- **Data** : c'est là où les inputs qui ont une influence sur le comportement du code sont mis. Ils peuvent être soient des variables simples, structures, tableaux ou retour de stub, etc. …
- **Functional range** : cette colonne est composée des valeurs valides et des variables utilisées principalement dans les tests nominaux.
- **Non functional Range** : elle est composée des valeurs invalides et des variables utilisées dans les tests de robustesse.

[8] La vérification de test par un autre testeur



	A	B	C
1	Data	Functional Range	Non functional Range
2	[AppBite_P_FailuresUpdated_Bo]	G_TRUE_Bo, G_FALSE_Bo	None
3	[L_TestStatus_E_Cst]	APPBITE_G_TESTRES_NOT_COMPUTED_E, APPBITE_G_TESTRES_FAILED_E, APPBITE_G_TESTRES_NOT_DETECTABLE_E, APPBITE_G_TESTRES_SUCCESS_E	[0x0004..0xFFFF]
4	[L_FailureStatus_E_Ptr]	APPBITE_P_FAILURE_ABSENT_E, APPBITE_P_FAILURE_LATCHED_E, APPBITE_P_FAILURE_PRESENT_E	[0x0003..0xFFFF]
5	[L_ConfirmationTimeElapsed_UB_Ptr]	[0x00..0xFF]	None
6	[L_AppearenceConfTime_UB_Cst]	[G_ZERO_UB .. G_UBYTE_MAX]	None
7	[L_DisappearenceConfTime_UB_Cst]	[G_ZERO_UB .. G_UBYTE_MAX]	None
8	[L_FailureNumber_E_Cst]	APPBITE_G_FNB_NOT_SAVED_E = 0, APPBITE_G_FNB_P_AACQ_E = 1, APPBITE_G_FNB_P_PSMON_E = 2, APPBITE_G_FNB_P_DSI_E = 3, APPBITE_G_FNB_P_WDG_E = 4, APPBITE_G_FNB_P_CMEASI_E = 5, APPBITE_G_FNB_P_SPARE_E= 6, APPBITE_G_FNB_C_AACQ_E= 7, APPBITE_G_FNB_C_CTCDRV_OPEN_E = 8, APPBITE_G_FNB_C_CTCDRV_CLOSE_E = 9, APPBITE_G_FNB_C_CMEASI_E = 10, APPBITE_G_FNB_C_CPFS_E = 11, APPBITE_G_FNB_C_NVM_E = 12, APPBITE_G_FNB_C_CTC_OPEN_E = 13, APPBITE_G_FNB_C_CTC_CLOSE_E = 14, APPBITE_G_FNB_C_PPI_E = 15, APPBITE_G_FNB_C_FS_E= 16, APPBITE_G_FNB_C_SPARE_E = 17	[0x0012..0xFFFF]
9			
10			
11			

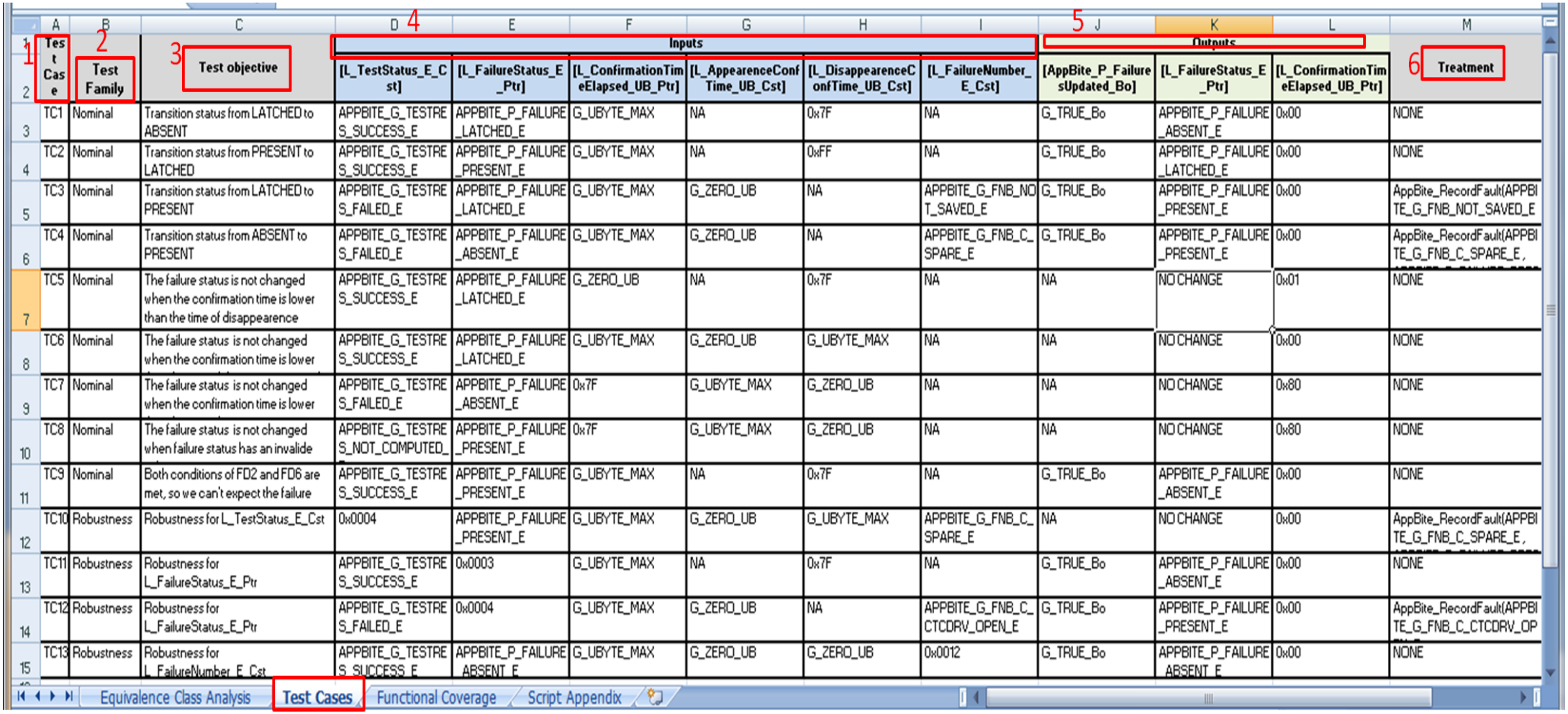
Figure 21 : La première page du fichier Excel « Equivalence Class Analysis »

2.2.2. Test Cases :

Elle se compose de six rubriques : Quatre sont fixes, c'est-à-dire qu'elles ne peuvent prendre qu'une seule colonne, et les deux qui restent peuvent prendre soit une colonne soit plusieurs colonnes:

- **Test Case** : C'est la 1ère colonne de la page Test Cases [Figure 22 : 1]. Elle contient l'identifiant de chaque cas de test.
- **Test Family** : il y a deux familles de test possibles : soit **nominal**, si les valeurs des entrées sont valides, soit **robustesse** si les valeurs d'entrée sont invalides. Ce test peut être considéré comme un test de performance. Cette colonne est aussi fixe [Figure 22 : 2].

- **Test objective** : dans cette colonne fixe on écrit le descriptif du comportement et le résultat attendu suite aux entrées du module (le chemin fonctionnel emprunté) [Figure 22 : 3].
- **Inputs** : elle est sous forme d'une rubrique, qui peut contenir des colonnes variables dépendant du nombre d'entrées du module à tester. Elle se compose de deux lignes comme entête [Figure 22 : 4], une pour écrire input et l'autre pour mettre les noms des variables. Nous la remplissons par les valeurs d'entrées choisies pour chaque cas de test. Il y a des cas où on peut ne pas avoir d'entrées au test donc nous donnons comme valeur pour un input **NONE**.
- **Outputs** : une rubrique qui peut contenir aussi des colonnes variables. Nous y reportons les résultats attendus pour chaque cas de test, autrement dit le comportement que le système doit avoir [Figure 22 : 5].
- **Treatment** : c'est une colonne fixe réservée au comportement des stubs [Figure 22 : 6].



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	1	2	3	4						5			
				Inputs						Outputs			
2	Test Case	Test Family	Test objective	[L_TestStatus_E_Cst]	[L_FailureStatus_E_Ptr]	[L_ConfirmationTimeElapsed_UB_Ptr]	[L_AppearenceConfTime_UB_Cst]	[L_DisappearenceConfTime_UB_Cst]	[L_FailureNumber_E_Cst]	[AppBite_P_FailureUpdated_Bo]	[L_FailureStatus_E_Ptr]	[L_ConfirmationTimeElapsed_UB_Ptr]	6 Treatment
3	TC1	Nominal	Transition status from LATCHED to ABSENT	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00	NONE
4	TC2	Nominal	Transition status from PRESENT to LATCHED	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	NA	0xFF	NA	G_TRUE_Bo	APPBITE_P_FAILURE_LATCHED_E	0x00	NONE
5	TC3	Nominal	Transition status from LATCHED to PRESENT	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_NOT_SAVED_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00	AppBite_RecordFault(APPBITE_G_FNB_NOT_SAVED_E
6	TC4	Nominal	Transition status from ABSENT to PRESENT	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_ABSENT_E	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_C_SPARE_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00	AppBite_RecordFault(APPBITE_G_FNB_C_SPARE_E,
7	TC5	Nominal	The failure status is not changed when the confirmation time is lower than the time of disappearence	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_ZERO_UB	NA	0x7F	NA	NA	NO CHANGE	0x01	NONE
8	TC6	Nominal	The failure status is not changed when the confirmation time is lower	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	G_ZERO_UB	G_UBYTE_MAX	NA	NA	NO CHANGE	0x00	NONE
9	TC7	Nominal	The failure status is not changed when the confirmation time is lower	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_ABSENT_E	0x7F	G_UBYTE_MAX	G_ZERO_UB	NA	NA	NO CHANGE	0x80	NONE
10	TC8	Nominal	The failure status is not changed when failure status has an invalide	APPBITE_G_TESTRES_NOT_COMPUTED_	APPBITE_P_FAILURE_PRESENT_E	0x7F	G_UBYTE_MAX	G_ZERO_UB	NA	NA	NO CHANGE	0x80	NONE
11	TC9	Nominal	Both conditions of FD2 and FD6 are met, so we can't expect the failure	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00	NONE
12	TC10	Robustness	Robustness for L_TestStatus_E_Cst	0x0004	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	G_ZERO_UB	G_UBYTE_MAX	APPBITE_G_FNB_C_SPARE_E	NA	NO CHANGE	0x00	AppBite_RecordFault(APPBITE_G_FNB_C_SPARE_E,
13	TC11	Robustness	Robustness for L_FailureStatus_E_Ptr	APPBITE_G_TESTRES_SUCCESS_E	0x0003	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00	NONE
14	TC12	Robustness	Robustness for L_FailureStatus_E_Ptr	APPBITE_G_TESTRES_FAILED_E	0x0004	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_C_CTCDRV_OPEN_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00	AppBite_RecordFault(APPBITE_G_FNB_C_CTCDRV_OP
15	TC13	Robustness	Robustness for L_FailureNumber_E_Cst	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_ABSENT_E	G_UBYTE_MAX	G_ZERO_UB	G_ZERO_UB	0x0012	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00	NONE

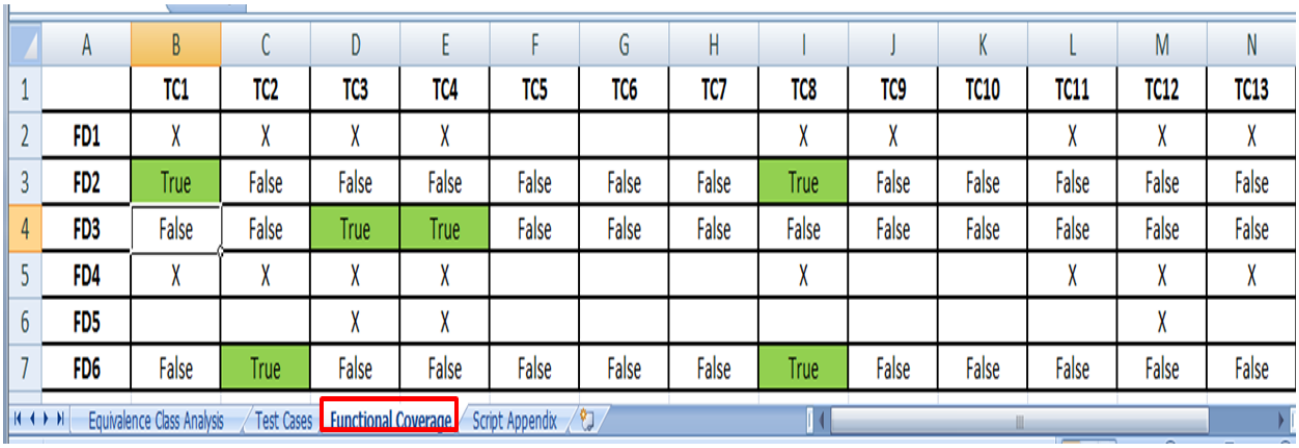
Equivalence Class Analysis | Test Cases | Functional Coverage | Script Appendix

Figure 22 : La deuxième page du fichier Excel « Test Cases »

2.2.3. Functional Coverage

Cette page est sous la forme d'une matrice qui décrit la fonctionnalité de la couverture pour chaque cas de test. Nous avons trois cas de test [Figure 23]:

- **True** : la condition permettant la couverture de la fonctionnalité est évaluée et vérifiée.
- **False** : la condition permettant la couverture de la fonctionnalité est évaluée et non vérifiée.
- **X** : le traitement, non conditionné, quelles que soient les entrées du module la fonctionnalité est couverte.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12	TC13
2	FD1	X	X	X	X				X	X		X	X	X
3	FD2	True	False	False	False	False	False	False	True	False	False	False	False	False
4	FD3	False	False	True	True	False	False	False	False	False	False	False	False	False
5	FD4	X	X	X	X				X			X	X	X
6	FD5			X	X								X	
7	FD6	False	True	False	False	False	False	False	True	False	False	False	False	False

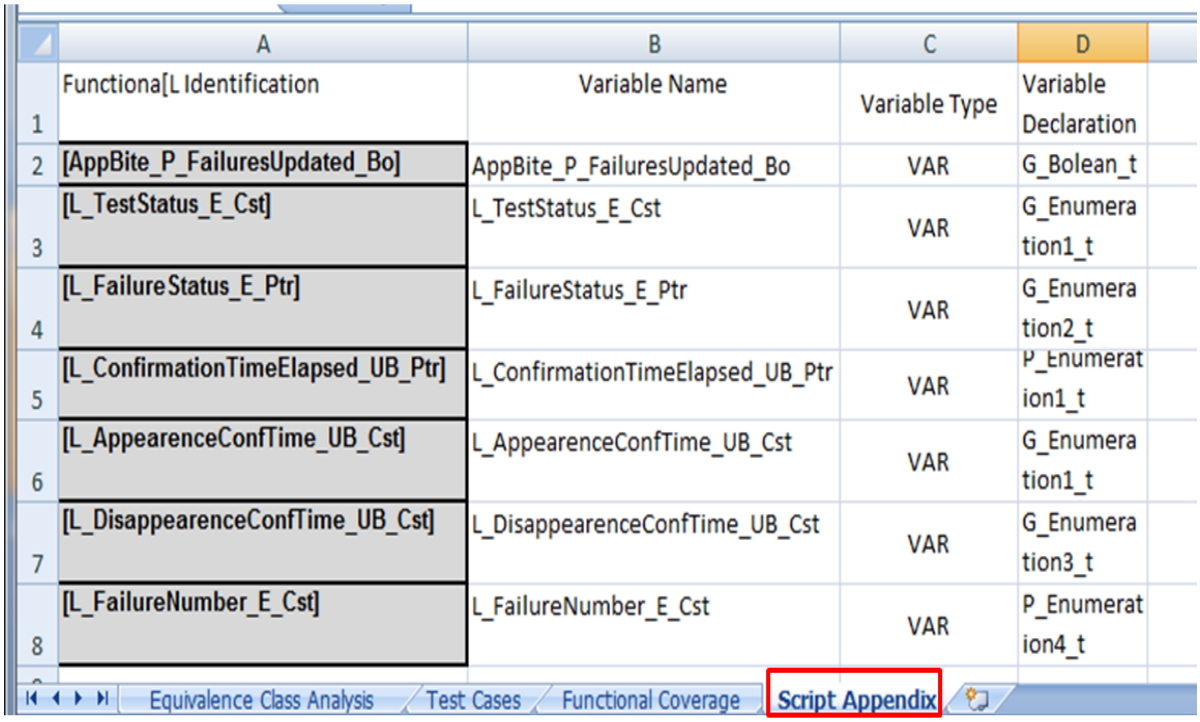
Equivalence Class Analysis | Test Cases | Functional Coverage | Script Appendix

Figure 23 : La troisième page du fichier Excel « Functional Coverage »

2.2.4. Script Appendix

Cette page est spécialement conçue pour ajouter tous nos besoins pour générer le script, mais ne rentre pas dans les scénarios de tests. Elle permet juste d'associer aux variables le type et le nom dans le logiciel. Elle se compose de quatre colonnes [Figure 24]:

- **Functional Identification** : le nom fonctionnel.
- **Variable Name** : le nom dans le Logiciel.
- **Variable Type** : étiquette de la variable utilisée par la syntaxe RTR. Elle peut être VAR, STR ou ARRAY.
- **Variable Declaration** : elle donne le type de la variable.



	A	B	C	D
1	Functiona[L Identification	Variable Name	Variable Type	Variable Declaration
2	[AppBite_P_FailuresUpdated_Bo]	AppBite_P_FailuresUpdated_Bo	VAR	G_Bolean_t
3	[L_TestStatus_E_Cst]	L_TestStatus_E_Cst	VAR	G_Enumeration1_t
4	[L_FailureStatus_E_Ptr]	L_FailureStatus_E_Ptr	VAR	G_Enumeration2_t
5	[L_ConfirmationTimeElapsed_UB_Ptr]	L_ConfirmationTimeElapsed_UB_Ptr	VAR	P_Enumeration1_t
6	[L_AppearenceConfTime_UB_Cst]	L_AppearenceConfTime_UB_Cst	VAR	G_Enumeration1_t
7	[L_DisappearenceConfTime_UB_Cst]	L_DisappearenceConfTime_UB_Cst	VAR	G_Enumeration3_t
8	[L_FailureNumber_E_Cst]	L_FailureNumber_E_Cst	VAR	P_Enumeration4_t

Equivalence Class Analysis | Test Cases | Functional Coverage | Script Appendix

Figure 24 : La quatrième page du fichier Excel « Script Appendix »

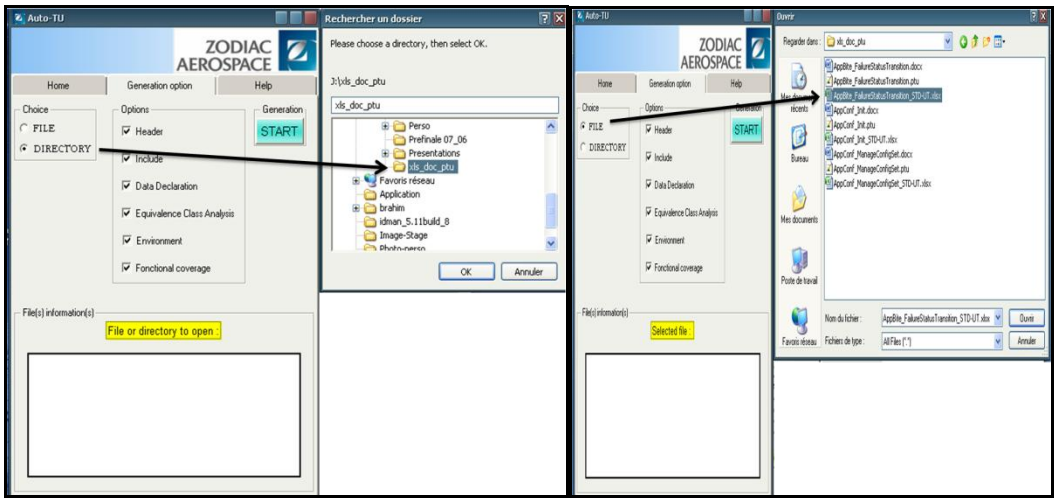
3. Solution

Après la phase de documentation et de la formation à-propos du métier et du besoin, nous avons pu préciser le besoin fonctionnel de l'application en collaboration avec l'équipe. La solution proposée est constituée de plusieurs étapes :

3.1. Recherche récursive

L'application doit être capable de parcourir chaque répertoire donné comme argument et accéder à toute son arborescence. Ainsi, tous les fichiers Excel existant (plan de test) vont être lus et, pour chacun d'eux, un fichier script de test d'extension ***.ptu***[9] va être généré avec le nom du module testé et au même niveau, c'est-à-dire, dans le même dossier où se trouve le plan de test. Dans le cas d'une mise à jour du LLR, un seul fichier Excel pourrait se trouver modifié. L'application doit être capable de ne sélectionner que ce fichier pour le traiter [Figure 25].

[9] Plan de Test Unitaire, c'est l'extension du script de test généré par l'application



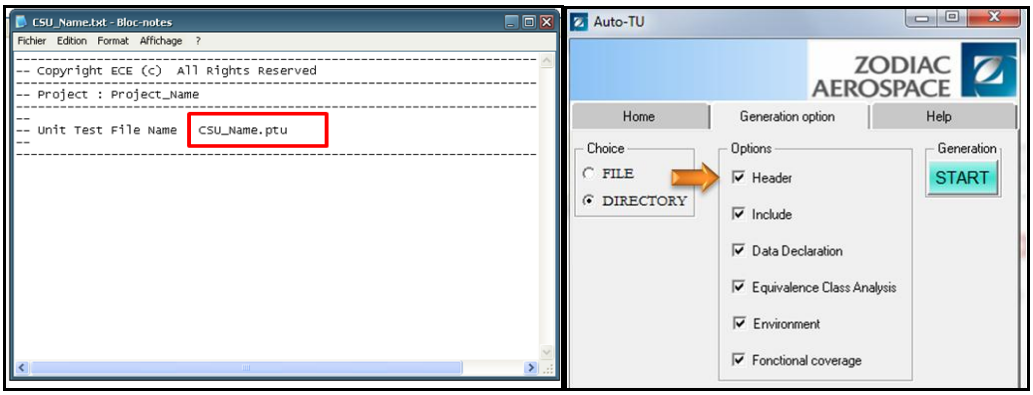
(1) (2)

Figure 25 : Le choix de sélectionner un répertoire(1) ou un plan de test(2)

Lorsque l'application accède à un répertoire et qu'elle trouve la cible, elle est censée le traiter comme entrée afin de générer le script de test équivalent au plan de test trouvé. Pour cela le groupe **ECE_logiciel** a une mise en forme que l'application doit respecter, commençant par :

3.2. L'entête du script de test

Pour générer la partie entête (Header), l'application copie le contenu d'un fichier .txt standard généré par ECE_Logiciel, remplace « CSU_NAME » par le nom du fichier Excel traité et insère le tout dans l'entête du fichier « .ptu ». Cette fonction doit être optionnelle, c'est le l'opérateur qui décide sa génération au niveau du script de test ou pas.

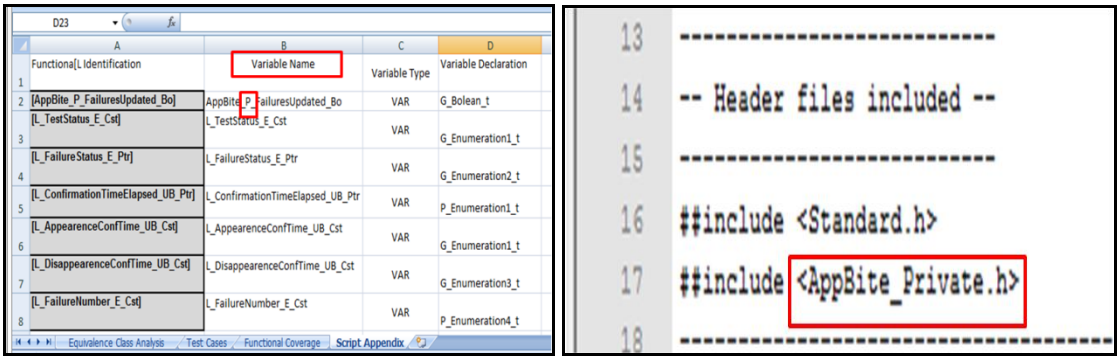


(1) (2)

Figure 26: (1) le Template du Header, (2) le choix optionnel du Header

3.3. Les fichiers à intégrer (Include)

Après la partie Header, vient la partie Include. Pour la remplir, nous devons tout d'abord écrire l'entête de cette partie qui se compose de « HEADER puis nom du module à tester » [Figure 27(2)]. Après, nous devons inclure, par défaut, le « ***standard.h*** » qui reste au choix du testeur de le garder ou de le supprimer, cela dépond du module à tester. Puis, nous ouvrons le fichier Excel et nous accédons à la page : « ***script appendix*** [10]», nous cherchons ensuite dans la colonne : « ***variable Name*** » [Figure 27(1)] les variables qui contiennent le motif « **_P_** » ou **«_G _».** Si nous trouvons le motif « **_P_** » la variable devient «***_Private.h*** », si nous trouvons «**_G _**» la variable devient « ***Global.h*** ». Enfin, nous écrivons sur le fichier .ptu en respectant la syntaxe RTR.



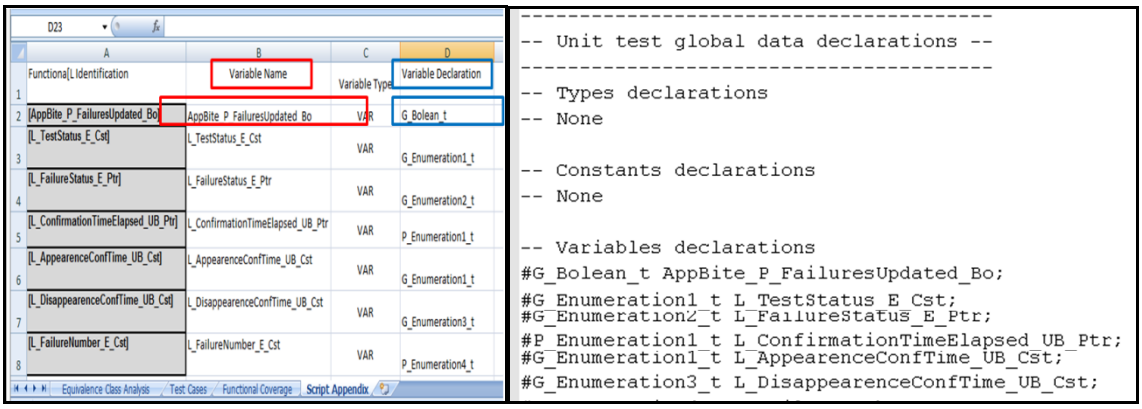
(1) (2)

Figure 27 : Exemple de cas de la présence du motif « _P_ » (1), la partie Include (2)

3.4. Déclaration de data

Pour remplir cette partie, nous avons besoin de remplir deux tableaux au niveau de l'algorithme de notre programme. Le premier tableau à partir de la feuille « ***script appendix*** » en parcourant la colonne « ***variable déclaration*** », et le deuxième tableau à partir de la même feuille en parcourant la colonne « ***variable Name*** ». Pour ce faire, nous ne relevons sur le fichier ***.ptu*** que les variables dont les noms fonctionnels commencent par un « ***[***» Figure 28 (1), pour éviter les stubs commençant par une «} », et qui ne contiennent pas un «.» et pour éviter les éléments d'une structure. Nous déclarons juste la structure mère.

[10] Feuille ajoutée au plan de test, qui contient les informations nécessaire à la l'application

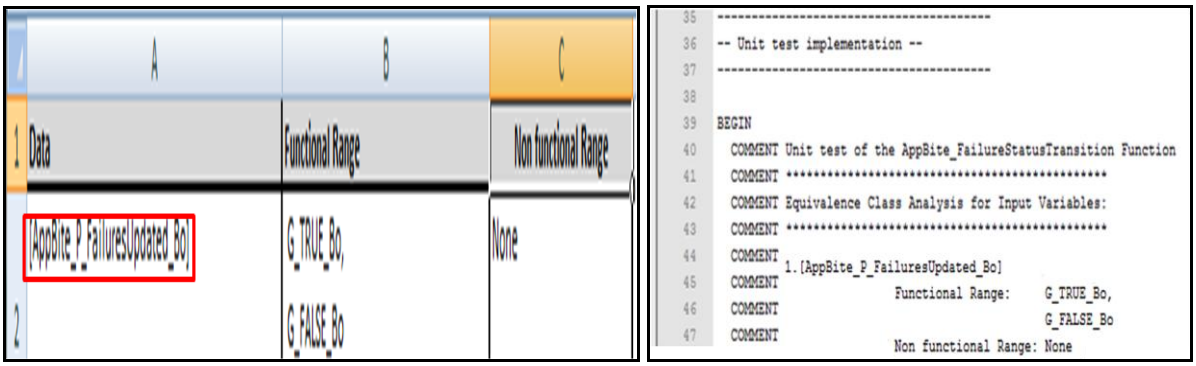


(1) (2)

Figure 28 : L'entrée (1) et la sortie (2) de la partie data déclaration

3.5. Les variables d'entrée de test

Nous commençons par l'écriture de l'entête propre à cette partie [Figure 29 (2)]. Nous utilisons pour le traitement de cette partie la feuille du fichier Excel, nous remplissons trois tableaux avec les trois colonnes de la feuille, puis nous enlevons les retours à la ligne et aussi s'il y a des espaces dans le « ***Functional Range*** [11]» et le « ***Non functional Range*** [12]», puis les écrire en respectant la syntaxe de RTR. Nous traitons aussi le cas où il n'y a pas de data (*variable*) par l'écriture de « ***NONE*** ».



(1) (2)

Figure 29 : L'entrée (1) et la sortie (2) de la partie Equivalence Class Analysis

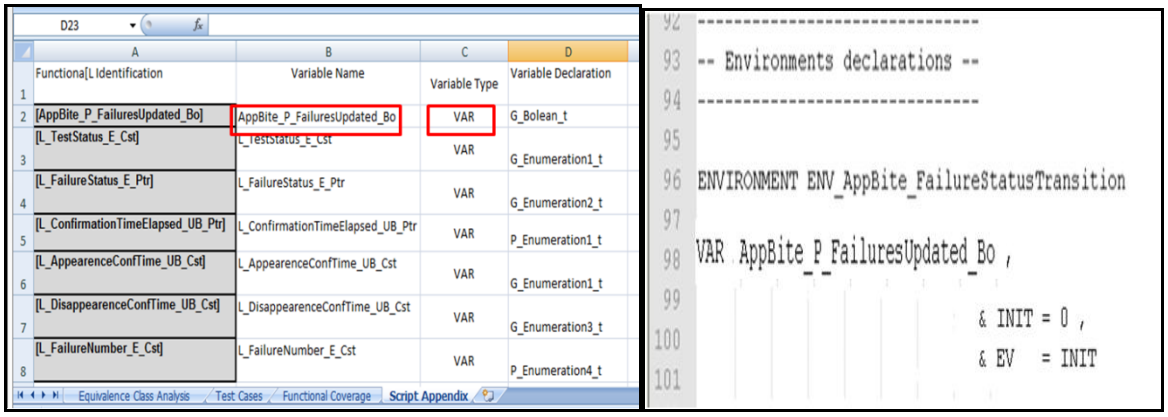
3.6. L'environnement du test

Nous commençons par l'écriture de l'entête propre à cette partie Figure 30(2), puis nous parcourons la case qui contient le type des variables à la recherche du type « ***var*** » au

[11] L'intervalle des valeurs possibles
[12] L'intervalle des valeurs non possibles

niveau de la feuille ***«Script Appendix »***. Pour chaque type ***VAR*** trouvé, le nom de la variable ne doit pas contenir un point pour ne pas déclarer les champs des structures mais déclarer juste la structure mère, et le nom fonctionnel doit commencer par un « ***[*** » pour ne pas considérer les stubs. Enfin, on écrit dans notre fichier **.ptu** avec la syntaxe de RTR en initialisant chaque variable.

Le type ***VAR*** a sa propre initialisation et valeur prévue, pour les autres variables qui ne contiennent pas un point et qui commencent par un « ***[*** » nous leur donnons une initialisation et une valeur prévue différente du type VAR.



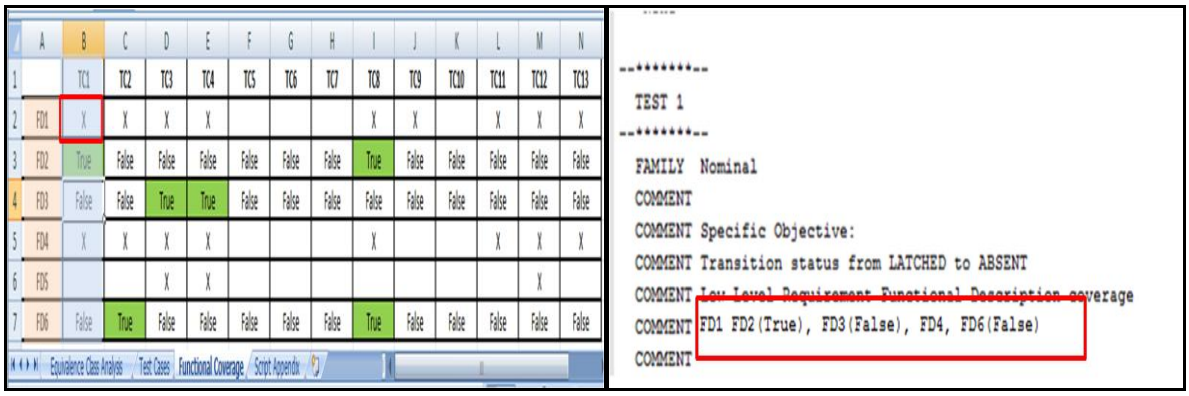
(1) (2)

Figure 30 : L'entrée (1) et la sortie (2) de la partie Environnement

3.7. Couverture fonctionnelle (*Functional Coverage*)

Le traitement de cette partie est inclus dans la partie test case. Pour la remplir, nous devons parcourir la colonne de la feuille du fichier Excel « ***Test Cases*** », prendre le numéro de chaque cas de test, puis nous parcourons la colonne de la feuille «***Functional Coverage*** » pour avoir les « ***FD »*** avec leurs numéros [Figure 31(1)].

Il y a trois cas pour un « ***FD »,*** soit il est à « ***true*** », soit il est à « ***false*** », soit il est à « ***X*** ». Dans chacun des cas, nous devons l'écrire dans le **.ptu** en respectant la syntaxe RTR. [Figure 31(2)].



(1) (2)

Figure 31: L'entrée (1) et la sortie (2) de la partie couverture fonctionnelle

3.8. Les cas de tests (Test case)

C'est la partie la plus importante dans le script du test qui sera généré. Cette partie comporte beaucoup de traitements. Nous commençons d'abord par chercher les conditions d'arrêt pour déterminer la lettre où se terminent les « ***inputs***[13] **»** ***[***Figure 32 ***: 1], et*** les « ***outputs***[14] **»**. Nous parcourons la feuille du fichier Excel « ***Test Cases*** », nous remplissons deux tableaux comme solution algorithmique. Le premier est rempli avec le nom fonctionnel des ***inputs*** qui se trouvent dans la ligne, et le deuxième est rempli avec le nom fonctionnel des ***outputs***. Nous remplaçons ensuite le nom fonctionnel des inputs et outputs par le type et le nom de la variable se trouvant dans la feuille « ***script appendix*** » du fichier Excel.

Après nous remplissons deux autres tableaux qui vont contenir la famille et l'objectif de chaque test. Il y a une contrainte sur le test objectif en l'écrivant dans le fichier **.ptu,** il ne doit pas dépasser quatre vingt caractères.

Puis nous remplissons deux tableaux avec les valeurs des inputs et des outputs, et nous les traitons en enlevant les espaces et les retours à la ligne. Si nous ne trouvons pas d'input, cela est indiqué dans le fichier Excel par un « **None »** à la place du nom de variable, soit pour les inputs soit pour les outputs, donc dans ce cas nous devons écrire sur le script de test un ***« none »*** au lieu des variables.

Nous cherchons, ensuite, les ***inouts***[15], c'est-à-dire les variables qui se trouvent dans les inputs et aussi dans les outputs. Il faut filtrer les valeurs des variables en omettant les

[13] Les variables d'entrée de test
[14] Les variables de sortie de test
[15] Les variables participant en entrée et en sortie dans le test

variables qui ont pour valeur ***« NA »*** et en enlevant les stubs : tous les noms de variables qui commencent par un ***« { ».***

A la fin, nous remplissons des tableaux avec la syntaxe RTR des inputs, outputs et inouts qui seront écrites dans le script de test [Figure 32 :(2)].

La limité des inputs

	Test objective	Inputs						Outputs		
		[L_TestStatus_E_Cst]	[L_FailureStatus_E_Ptr]	[L_ConfirmationTimeElapsed_UB_Ptr]	[L_AppearenceConfTime_UB_Cst]	[L_DisappearenceConfTime_UB_Cst]	[L_FailureNumber_E_Cst]	[AppBite_P_FailuresUpdated_Bo]	[L_FailureStatus_E_Ptr]	[L_ConfirmationTimeElapsed_UB_Ptr]
3	Transition status from LATCHED to ABSENT	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00
4	Transition status from PRESENT to LATCHED	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	NA	0xFF	NA	G_TRUE_Bo	APPBITE_P_FAILURE_LATCHED_E	0x00
5	Transition status from LATCHED to PRESENT	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_NOT_SAYED_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00
6	Transition status from ABSENT to PRESENT	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_ABSENT_E	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_C_SPARE_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00
7	The failure status is not changed when the confirmation time is lower than the time of disappearence	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_ZERO_UB	NA	0x7F	NA	NA	NO CHANGE	0x01
8	The failure status is not changed when the confirmation time is lower than the time of	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_LATCHED_E	G_UBYTE_MAX	G_ZERO_UB	G_UBYTE_MAX	NA	NA	NO CHANGE	0x00
9	The failure status is not changed when the confirmation time is lower than the time of	APPBITE_G_TESTRES_FAILED_E	APPBITE_P_FAILURE_ABSENT_E	0x7F	G_UBYTE_MAX	G_ZERO_UB	NA	NA	NO CHANGE	0x80
10	The failure status is not changed when failure status has an invalide value	APPBITE_G_TESTRES_NOT_COMPUTED_E	APPBITE_P_FAILURE_PRESENT_E	0x7F	G_UBYTE_MAX	G_ZERO_UB	NA	NA	NO CHANGE	0x80
11	Both conditions of FD2 and FD6 are met, so we can't espect the failure status	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00
12	Robustness for L_TestStatus_E_Cst	0x0004	APPBITE_P_FAILURE_PRESENT_E	G_UBYTE_MAX	G_ZERO_UB	G_UBYTE_MAX	APPBITE_G_FNB_C_SPARE_E	NA	NO CHANGE	0x00
13	Robustness for L_FailureStatus_E_Ptr	APPBITE_G_TESTRES_SUCCESS_E	0x0003	G_UBYTE_MAX	NA	0x7F	NA	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00
14	Robustness for L_FailureStatus_E_Ptr	APPBITE_G_TESTRES_FAILED_E	0x0004	G_UBYTE_MAX	G_ZERO_UB	NA	APPBITE_G_FNB_C_CTCDRV_OPEN_E	G_TRUE_Bo	APPBITE_P_FAILURE_PRESENT_E	0x00
15	Robustness for L_FailureNumber_E_Cst	APPBITE_G_TESTRES_SUCCESS_E	APPBITE_P_FAILURE_ABSENT_E	G_UBYTE_MAX	G_ZERO_UB	G_ZERO_UB	0x0012	G_TRUE_Bo	APPBITE_P_FAILURE_ABSENT_E	0x00

(1)

```
133     COMMENT
134     COMMENT Specific Objective:
135     COMMENT Transition status from LATCHED to ABSENT
136     COMMENT Low Level Requirement Functional Description coverage
137     COMMENT FD1, FD2(True), FD3(False), FD4, FD6(False)
138     COMMENT
139
140     ELEMENT
141        -- Test inputs
142        VAR L_TestStatus_E_Cst ,
143                          &INIT = APPBITE_G_TESTRES_SUCCESS_E,
144                          &EV   = INIT
145        VAR L_DisappearenceConfTime_UB_Cst ,
146                          &INIT = 0x7F,
147                          &EV   = INIT
148        -- Expected test results (OUT and IN/OUT)
149        VAR AppBite_P_FailuresUpdated_Bo ,
150                          &INIT = G_TRUE_Bo + 1,
151                          &EV   = G_TRUE_Bo
152        VAR L_FailureStatus_E_Ptr ,
153                          &INIT = APPBITE_P_FAILURE_LATCHED_E,
154                          &EV   = APPBITE_P_FAILURE_ABSENT_E
155        VAR L_ConfirmationTimeElapsed_UB_Ptr ,
156                          &INIT = G_UBYTE_MAX,
157                          &EV   = 0x00
158     --********--
```

(2)

Figure 32 : L'entrée (1) et la sortie (2) de la partie test case

4. Structure générale d'un script de test (.ptu)

Les scripts en langage RTR sont composés d'une série d'instructions dans lesquelles nous pouvons insérer le code C. la syntaxe de script RTR ou code C doit respecter certaines conditions et règles à savoir :

- Chaque instruction RTR commence par un mot clé (Keyword)
- La longueur d'une instruction RTR est limitée à 2000 caractères
- Le caractère '**&**' peut être utilisé dans chaque début de ligne additionnelle pour étendre une instruction RTR (opérateur de concaténation).
- Le code C peut être introduit dans le script RTR en introduisant en début de ligne du code C l'un des deux caractères ''**#**' ou '**@**' [16].

```
HEADER <nom_module>, <version_module>, <version_test_script>
BEGIN
DEFINE STUB nom_module_c
...
END DEFINE

ENVIRONMENT env_nom_environment
...
END ENVIRONMENT

SERVICE nom_module
TEST no.

FAMILY type_test
USE env_nom_environment

ELEMENT
```

```
-- input variable

-- output variable


#function name();

END ELEMENT


END TEST
```

Figure 33 : La structure générale d'un script de test

3.1. Description des sections d'un fichier de test (.ptu)

Le script de tests se compose de plusieurs parties. Le tableau suivant présente les parties essentielles de ce fichier.

Section du fichier .ptu	Description
HEADER	Spécifier le nom et la version du module à tester
BEGIN	Le début du test
SERVICE	Spécifier le nom du service
TEST	Chaque test est identifié par un numéro unique dans un service
FAMILY	indique la famille de cas de test (robustesse, nominale…)
ELEMENT	Décrit le cas de test (Input, Output, Stub…)
END ELEMENT	Fin de la section ELEMENT
END TEST	Fin de la section TEST
END SERVICE	Fin de la section SERVICE

Tableau 5 : Les sections d'un script de test [**16**]

3.2. DEFINE STUB et END DEFINE

Délimite le bloc de simulation. Contient la définition des stubs en précisant le mode de passage des paramètres [16].

Syntaxes:

```
DEFINE STUB <stub_name> [ <stub_dim> ] END DEFINE
```

<stub_name> : C'est le nom du bloc de simulation. (Obligatoire)

<stub_dim> : le nombre maximal des appels stub effectués à chaque test, par défaut = 10.

Exemples:

```
DEFINE STUB my_stub

#int create_file(char _in f[100]);

#int read_file(int _in fd, char _out l[100]);

#int write(int _in fd, char _in l[100]);

#int close_file(int _in fd);

#void fct1(int _inout par1);

#void fct2(int * _no par1);

END DEFINE
```

3.3. ELEMENT et END ELEMENT

Ce bloc contient l'initialisation et le check des variables du test, l'appel aux stubs couvert par le test, et aussi l'appel au module qu'on teste. [16]

```
ELEMENT

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY image, init = 0, ev = init

VAR status , init = 0, ev = 2
```

```
VAR y1, init = 0, ev = 2

VAR histo[0], init = 0, ev = SIZE_IMAGE*2

STUB fct1((1,4));

#status = compute_histo(x1,y1,x2,y2,histo);

END ELEMENT
```

3.4. ENVIRONEMENT et END ENVIRONEMENT

Ce bloc définit le contexte global du jeu de test, et permet de mettre en place une initialisation par défaut aux variables du jeu de test. L'activation d'un environnement se fait via le mot USE, cette activation est reconnue juste au bloc où il est appelé [16].

```
ENVIRONMENT compute_histo

VAR x1, init = 0, ev = init

VAR x2, init = SIZE_IMAGE-1, ev = init

VAR y1, init = 0, ev = init

VAR y2, init = SIZE_IMAGE-1, ev = init

ARRAY histo, init = 0, ev = 0

VAR status, init ==, ev = 0

END ENVIRONMENT
```

Conclusion

La criticité du travail demandé se manifeste par la génération de la syntaxe RTR, qui sera exploitée dans un domaine critique.

La solution n'était pas possible sans la compréhension du besoin et les attentes voulues de cette application, et aussi la compréhension détaillée de chaque partie de plan de test, l'entrée de l'application. Le traitement de la solution a été reparti par modules afin de la tester séparément et de s'assurer que le travail effectué répond parfaitement au besoin.

Conclusion générale

Les tests unitaires sont des activités très importantes pour le cycle de développement des logiciels en général, et notamment pour les logiciels avioniques embarqués. Dans ce travail nous avions à automatiser la génération des scripts de tests qui se situent au cœur des activités de tests unitaires. La génération se fait à partir des plans de tests sous format Excel.

Pour accomplir notre travail, nous avons utilisé le logiciel PERL qui facilite la manipulation et le traitement des fichiers. La contrainte était la génération de la syntaxe RTR qui est compilée par le logiciel de test RTRT, et cette opération devra passer sans erreur.

Les premiers tests de notre application ont révélé certains besoins pour limiter les erreurs susceptibles d'apparaître au moment de test, comme la création d'un modèle de plan de test.

Comme suggestions d'amélioration de l'application, il est souhaitable d'ajouter trois options pour plus de performances:

- Ajouter la possibilité d'accéder au réseau de l'entreprise, afin de pouvoir exporter les projets à tester. Cette fonctionnalité permettra à l'application une autonomie au niveau de traitement des projets sur le réseau, il suffit juste de donner au développeur le droit d'accès au réseau pour répondre à ce besoin.
- Se focaliser sur la mise en accord pour définir la partie des appels aux modules « STUB » au niveau des plans de tests, pour chercher les algorithmes convenables aux traitements de cette partie.
- Gestion des erreurs qui apparaissent au niveau de la génération des scripts de tests. Il faut que l'application soit capable de détecter l'origine de l'erreur, et de proposer des solutions pour orienter le testeur.

Ce stage au sein de ZAM m'a permis de découvrir la réalité d'un projet en milieu professionnel. J'ai dû affronter les difficultés liées à un projet informatique en général. Le projet réalisé était un projet de durée courte mais très consistent.

J'ai pu découvrir en participant au développement du projet la rigueur nécessaire à la satisfaction du client et les bonnes pratiques nécessaires à la réalisation d'un logiciel de

qualité. Je me suis également rendu compte de l'importance de la coopération dans l'équipe et du dialogue avec le client. D'un point de vu technique, le projet a été très enrichissant. J'ai également travaillé sur des aspects métiers à la fin de mon projet.

Ce projet a donc été extrêmement enrichissant. J'ai énormément appris sur le plan technique mais aussi en terme d'organisation de projet. Il était également très intéressant de participer depuis le début des développements jusqu'aux portes de la livraison.

Je n'aurais jamais pu m'intégrer aussi facilement sur ce projet si l'équipe n'avait pas, tout au long du projet, été extrêmement soudée et toujours là pour m'aider. Ce stage a donc été une excellente assise pour ma future vie professionnelle et je pense que les connaissances acquises me serviront aussi bien à court terme dans le cadre de la poursuite du projet qu'à plus long terme sur d'autres expériences de développement.

Références

SITES INTERNET CONSULTES

1 : http://fr.wikipedia.org/wiki/Zodiac_Aerospace

2 : http://www.zodiacaerospace.com/content/group-fr/index.php?ifile=/content/group-fr/Groupe/Organisation_du_groupe/

3 : http://www.zodiacaerospace.com/content/group-fr/index.php?ifile=/content/group-fr/Groupe/Organisation_du_groupe/

4 : http://maghrebinfo.actu-monde.com/archives/article1961.html

5 : Présentation ZAM- Logiciel ECE, D. DOUKKALI : 01/2012, page 9

6: http://www.vectorcast.fr/industries/do178b-tool-qualification.php

7: Document (ZODIAC AEROSPACE/INTERTECHNIQUE : Formation logiciel, module 1 : introduction au développement Section 1 : introduction au développement DO-178B, page : 25-50)

8: http://www-igm.univ-mlv.fr/~dr/XPOSE2002/Site_Vaubourg_Stephane_IR3/

9: http://fr.wikipedia.org/wiki/Cycle_en_V

10 : Document (ZODIAC AEROSPACE/INTERTECHNIQUE : Formation Tests Unitaires, module 1 : définition, Types et Techniques de Tests, page : 28-32)

11 : http://www.vectorcast.fr/testing-solutions/unit-integration-embedded-software-testing.php

12 : http://fr.wikipedia.org/wiki/Perl_(langage)

13 : http://articles.mongueurs.net/magazines/linuxmag39.html

14 : http://ferry.eof.eu.org/lesjournaux/pl/public_html/c6842.html

15: http://perl.enstimac.fr/perl-all-fr-pdf.pdf "documentation perl (en français),version du 6aout 2009,chapitre 3,pages (12->18).

16 : Document (ZODIAC AEROSPACE/INTERTECHNIQUE : Formation RTRT, pages:(17->22, 25, (33->52)).

Annexes

Annexe 1: la syntaxe de PERL

1. Types de variables Perl

Perl propose trois types de variables : les scalaires, les tableaux et les tables de hachage.

1.1. Scalaires

Un scalaire représente une valeur unique :

```
my $animal = "chameau";
my $reponse = 42;
```

Les scalaires peuvent être indifféremment des chaînes de caractères, des entiers, des nombres en virgules flottante, des références, plus quelques valeurs spéciales. Perl procédera automatiquement aux conversions nécessaires lorsque c'est cohérent. Il est inutile de déclarer au préalable le type des variables mais, en revanche, il faut déclarer les variables lors de leur première utilisation en utilisant le mot-clé **my** [15].

1.2. Les tableaux

Un tableau représente une liste de valeurs :

```
my @animaux = ("chameau", "lama", "hibou");
my @nombres = (23, 42, 69);
my @melange = ("chameau", 42, 1.23);
```

Le premier élément d'un tableau se trouve à la position 0 [15].

1.3. Les tables de hachage

Une table de hachage représente un ensemble de paires clé/valeur :

```
my %fruit_couleur = ("pomme", "rouge", "banane", "jaune");
```

```
my %fruit_couleur = (
```

```
pomme => "rouge",
banane => "jaune",);
```

Pour accéder aux éléments d'un hash [15] :

```
$fruit_couleur{"pomme"}; # donne "rouge"
```

2. Structures conditionnelles et boucles

Perl dispose des structures d'exécution conditionnelles et des boucles usuelles des autres langages de programmation, à l'exception de case/switch.

Une condition peut être n'importe quelle expression Perl. Elle est considérée comme vraie ou fausse suivant sa valeur, 0 ou chaîne vide signifiant FAUX, et toute autre valeur signifiant VRAIE [15].

2.1. *if*

```
if ( condition ) {...}
elsif ( autre condition ) {...}
else {...}
```

Il existe également une version négative du if :

```
unless ( condition ) {...}
```

2.2. *While*

```
while ( condition ) {...}
```

Il existe également une version négative de while :

```
until ( condition ) {...}
```

2.3. *for*

La boucle for ressemblant à celle en C est, toutefois, rarement nécessaire en Perl dans la mesure où Perl fournit une alternative plus intuitive : la boucle de parcours de liste foreach.

2.4. *Foreach*

```
foreach (@array) {print "L'élément courant est $_\n";}
```

***Remarque** : En pratique nous pouvons substituer librement for à **foreach** et inversement. Perl se charge de détecter la variante utilisée* [15].

3. Opérateurs et fonctions internes

Perl dispose d'une large gamme de fonctions internes. Voici quelques-unes parmi les plus utilisées :

3.1. Arithmétique

```
+ addition
- soustraction
* multiplication
/ division
```

3.2. Comparaison numérique

```
== égalité
!= inégalité
< inférieur
> supérieur
<= inférieur ou égal
>= supérieur ou égal
```

3.3. Comparaison de chaînes

```
eq égalité
ne inégalité
lt inférieur
gt supérieur
le inférieur ou égal
ge supérieur ou égal
```

3.4. Logique booléenne

```
&&  and et
||     or ou
!     not négation
```

3.5. Divers

```
=   affectation
.   concaténation de chaînes
x   multiplication de chaînes
..  opérateur d'intervalle (crée une liste de nombres)
```

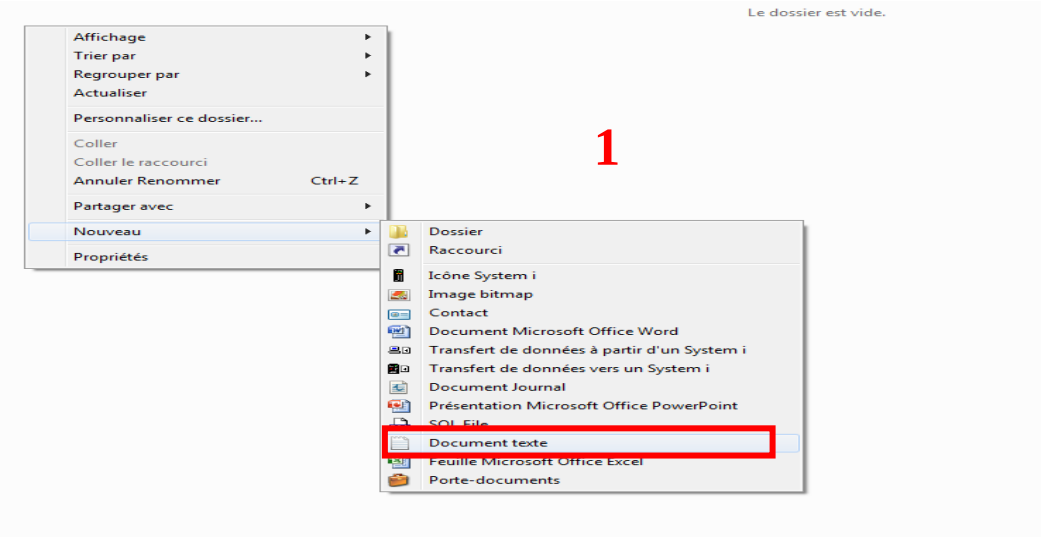
Annexe 2 : Développement des scripts avec PERL

Pour exécuter un programme PERL on suit les étapes suivantes :

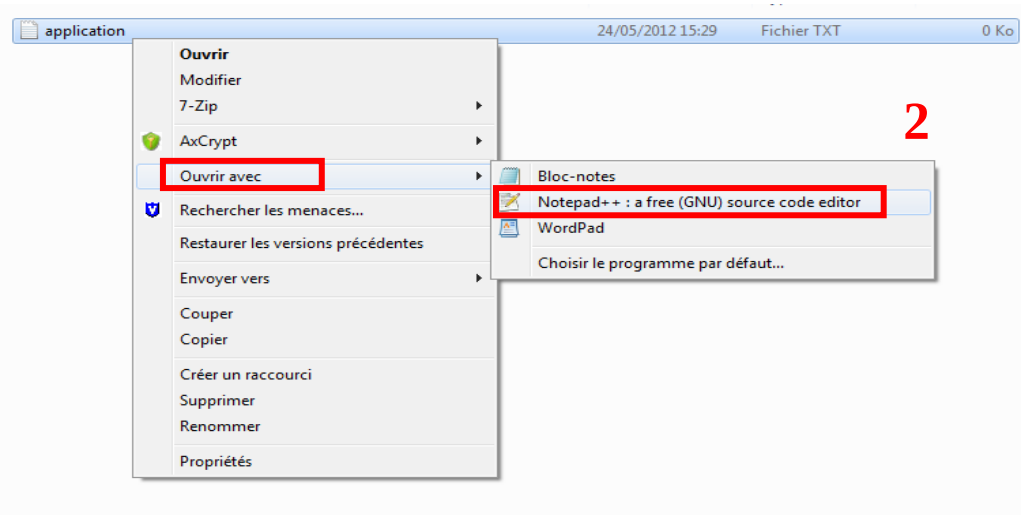
1. Installer ActivePerl.
2. Installer notepad++.

Remarque : Faute à la contrainte de droit d'administration pour les machines d'entreprise nous avons utilisé une autre méthode concernant l'installation d'ActivePerl et notepad++. Nous avons juste déposé le dossier contenant PERL dans la partition du disque dur **C:\. De** même pour notepad++.

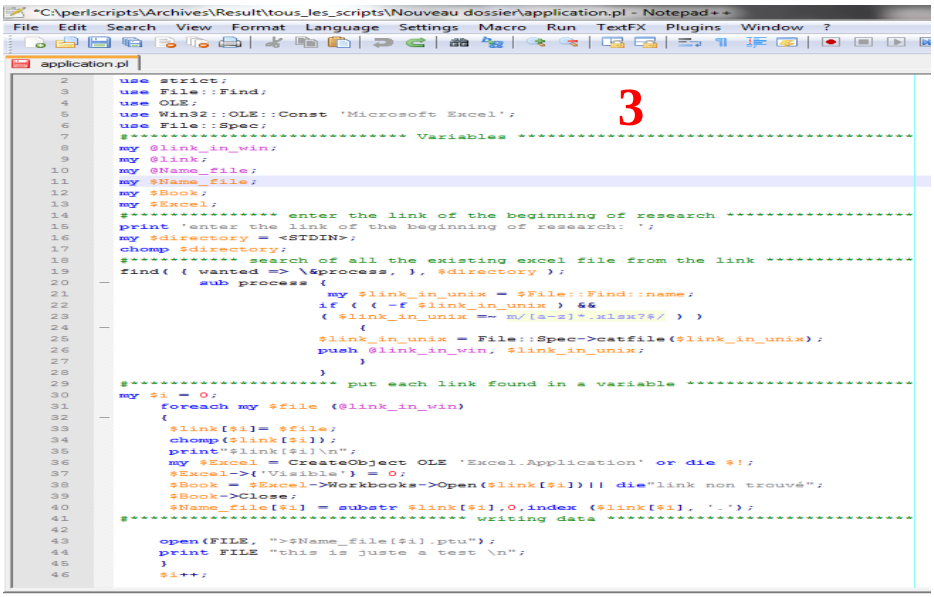
3. Créer un fichier .txt



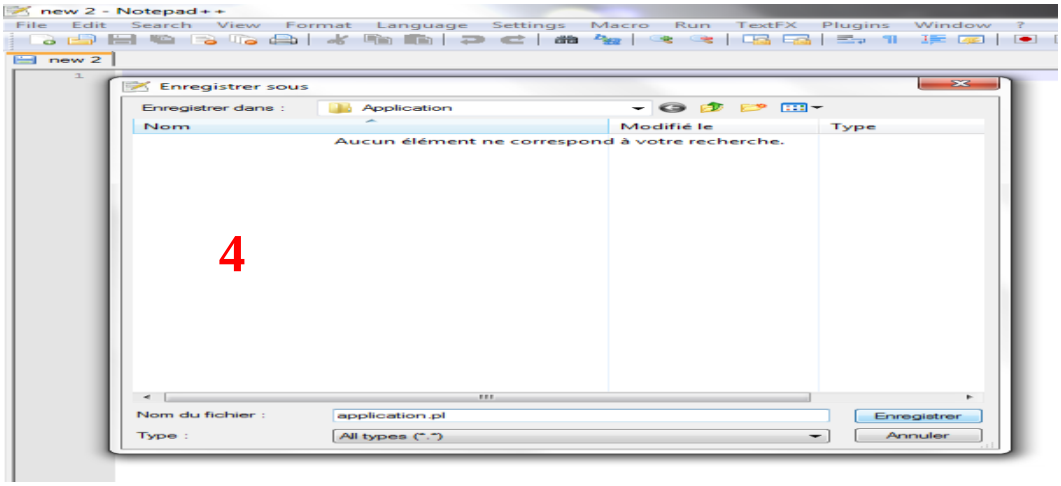
L'ouvrir avec notepad++.



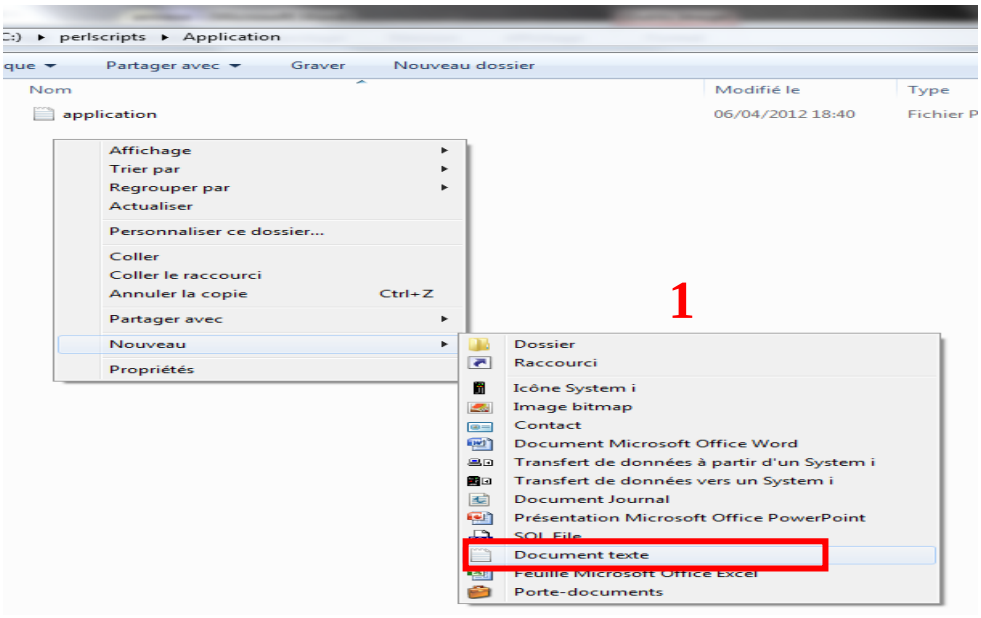
Ecrire le code

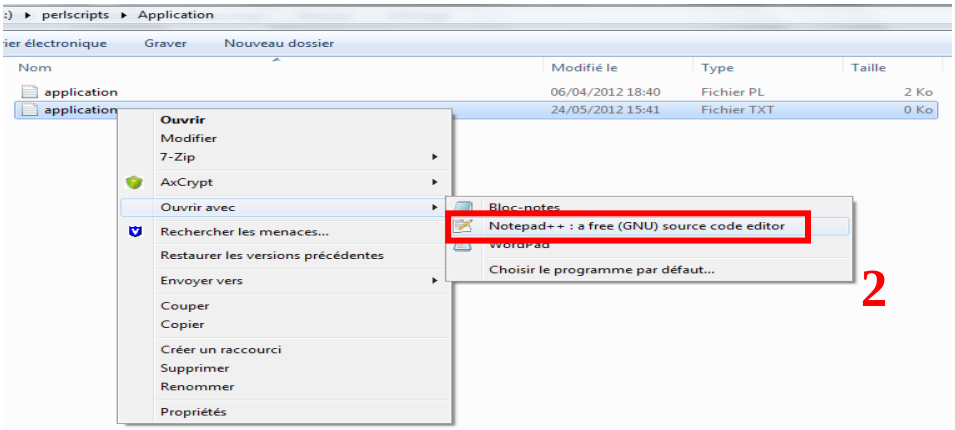


L'enregistrer sous format **.pl**

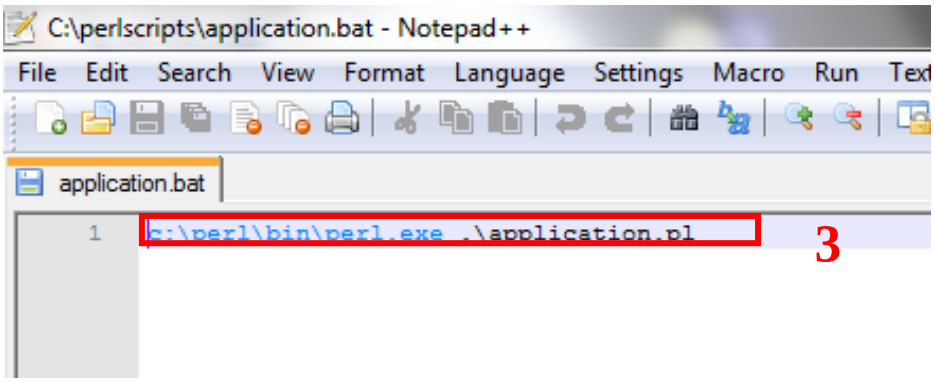


Créer un autre fichier .txt cette fois pour exécuter le premier fichier « .pl »

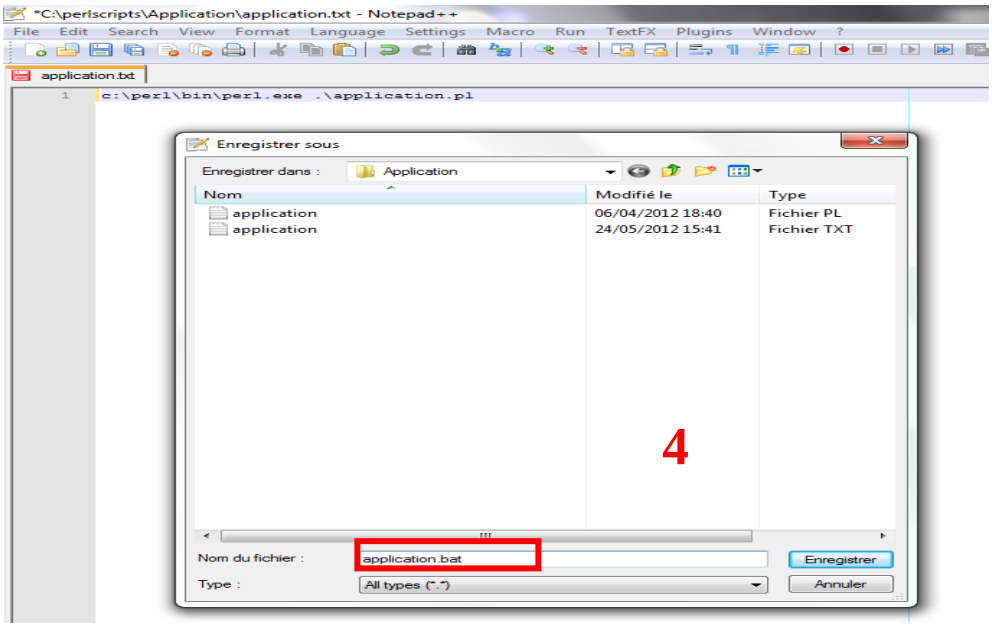




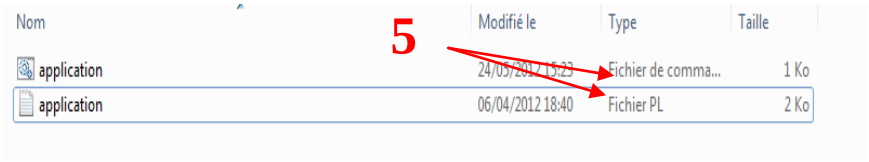
Donner le chemin de perl.exe suivi de **« \nomdeprogramme.pl »**



L'enregistrer sous format **.bat**



A la fin on obtient deux fichiers .pl contenant le programme le .bat contenant l'exécutable



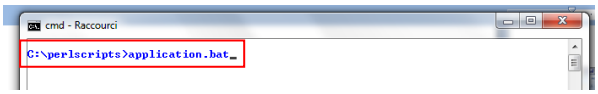
4. Exécution de programme

Il y a deux méthodes :

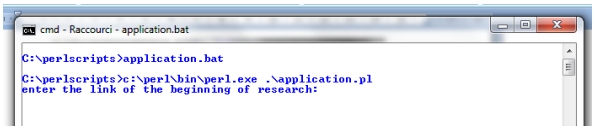
Soit nous cliquons directement sur le fichier (.bat), et l'exécution du programme passera automatiquement, si l'exécution passera normalement, il n'y aura pas de problème, mais dans le cas échouant, cette méthode ne permet par de voir les erreurs du programme.

Pour surmonter ce problème nous utilisons la deuxième méthode :

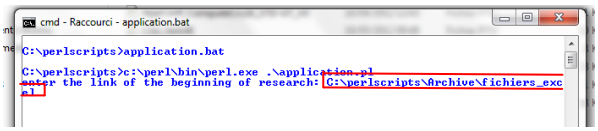
- Nous ouvrerons la console de Windows puis nous accédons au chemin où se trouve le fichier (.bat) de notre programme voici un exemple :



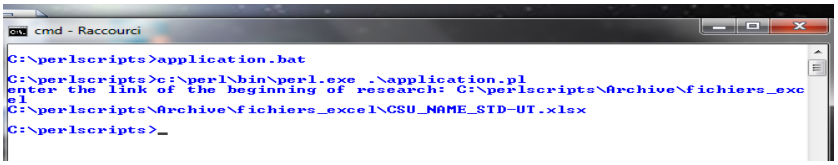
- Nous lançons l'exécution :



- Le programme fonctionne correctement, il demande le chemin du parcours (traduit ce qu'il ya dans le programme (.pl)), on lui donne le chemin du parcours :

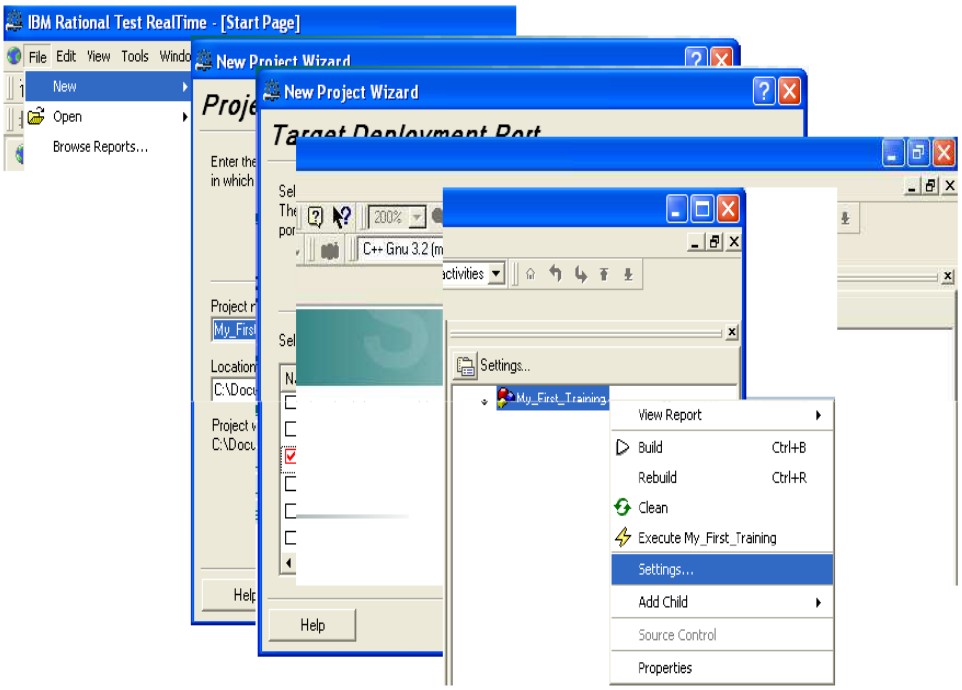


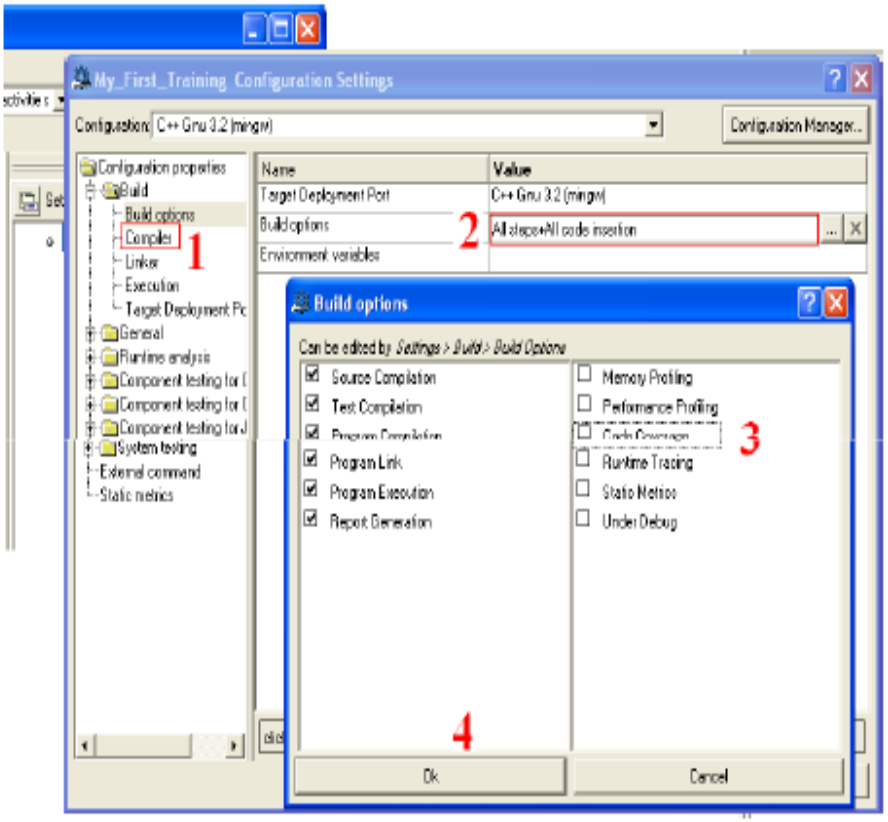
- Voici le résultat de parcours (il a trouvé un fichier Excel dans ce chemin)



Annexe 3 : Exécution des tests sous RTRT [16]

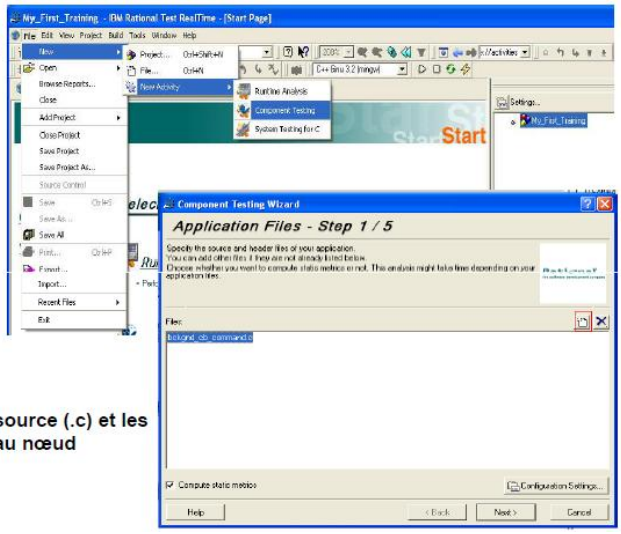
- Création du projet





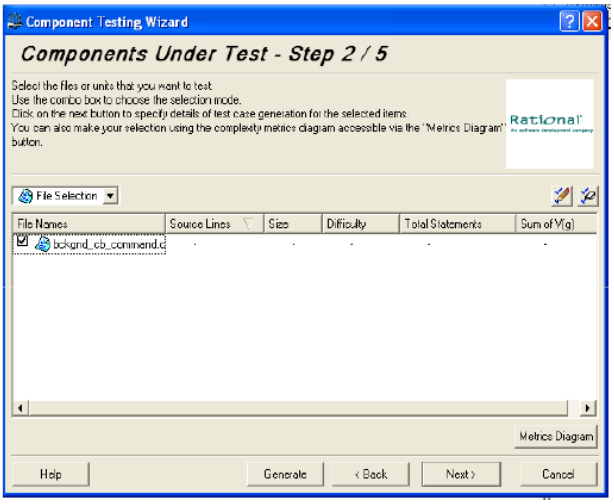
- Génération automatique d'un PTU

➢Lancer l'assistant « component testing »



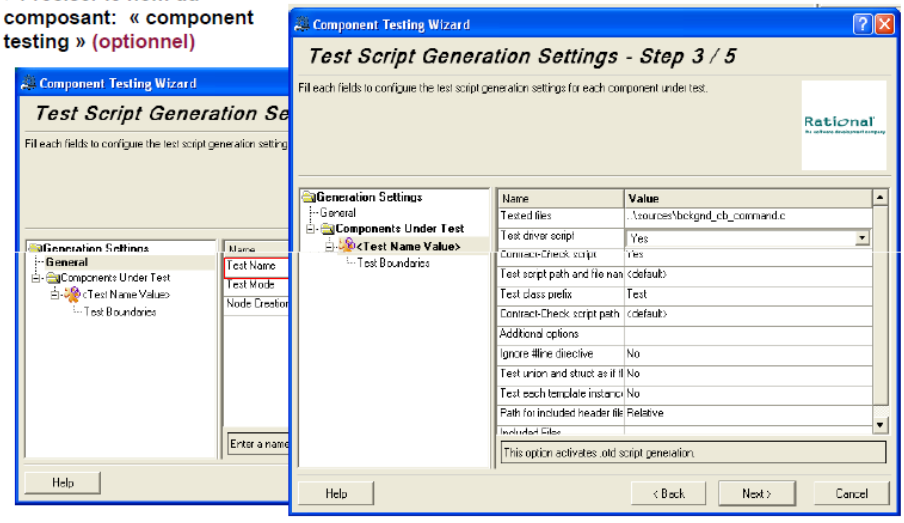
➢Ajouter les fichiers source (.c) et les fichiers (.h) à utiliser au nœud

➢Sélectionner le fichier source (.c) à tester

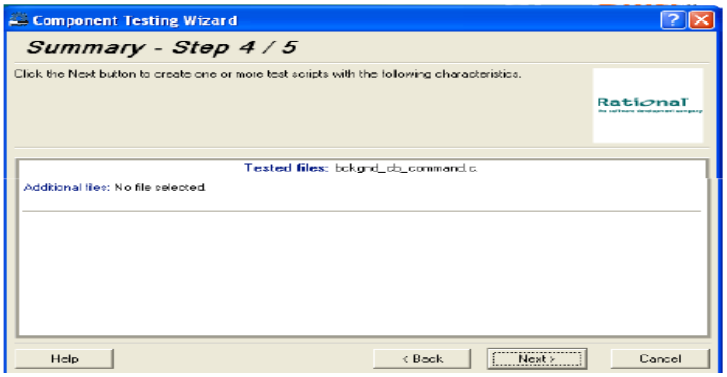


➢Préciser le nom du composant: « component testing » (optionnel)

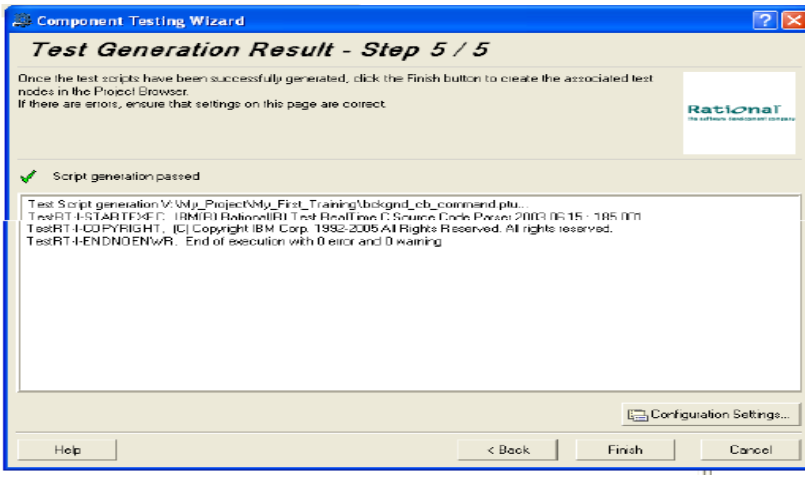
➢Préciser le chemin du fichier script



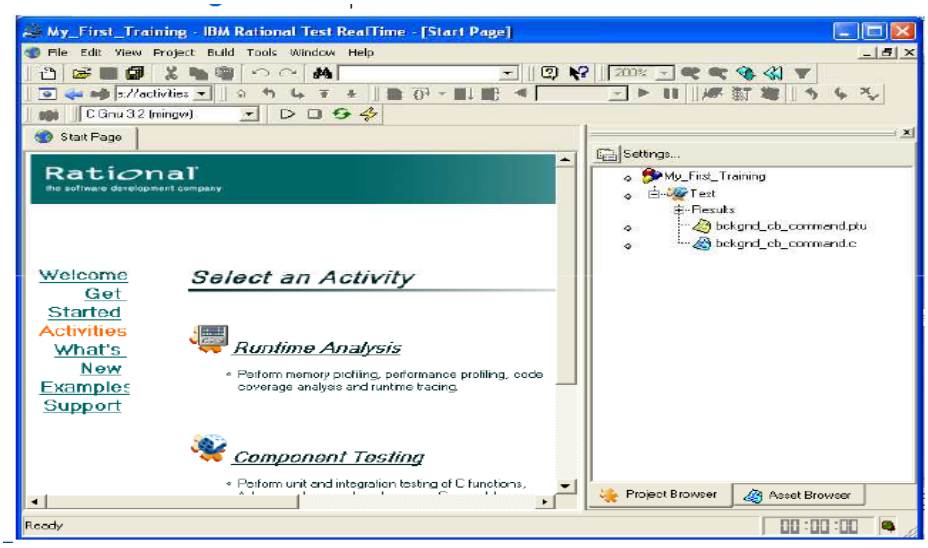
- Synthèse à valider avant génération du fichier de test



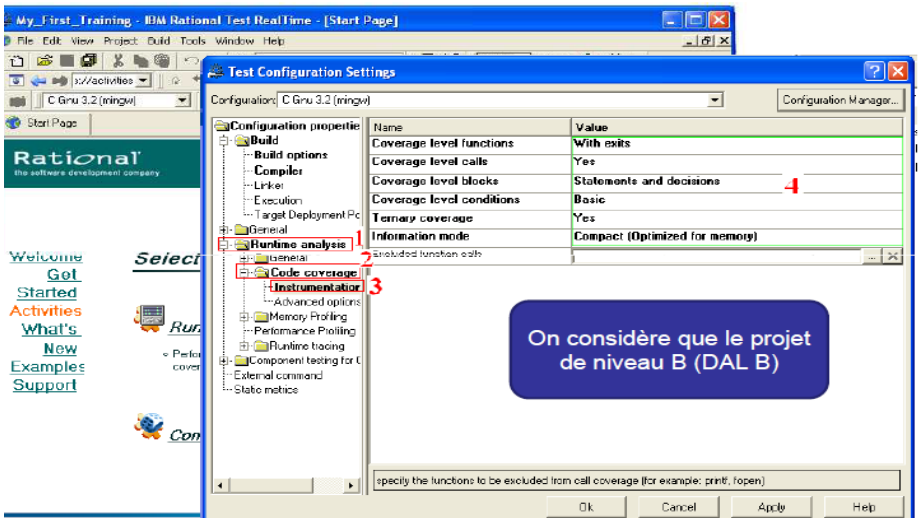
- Génération du fichier PTU



- Résultat de la génération



- Configuration d'un projet



- Signification des options de configuration

Name	Value
Coverage level functions	With exits
Coverage level calls	Yes
Coverage level blocks	Statements and decisions
Coverage level conditions	Basic
Ternary coverage	Yes
Information mode	Compact (Optimized for memory)

- **Coverage level functions:**

- **With exits**: les entrées, sorties de la fonction sont instrumentés.
- **Coverage level calls:**
- **Yes**: Instrumentation des appels de fonction (stub).
- **Coverage Level Blocks:**
- **Statements And decision**: Bloc Simple et implicite (implicite else, implicite default de switch) sont instrumentés.
- **Coverage Level Conditions :**
- **Basic :** Condition basique est instrumentée

Ternary Coverage:

- **Yes**: les expressions ternaires sont instrumentées.

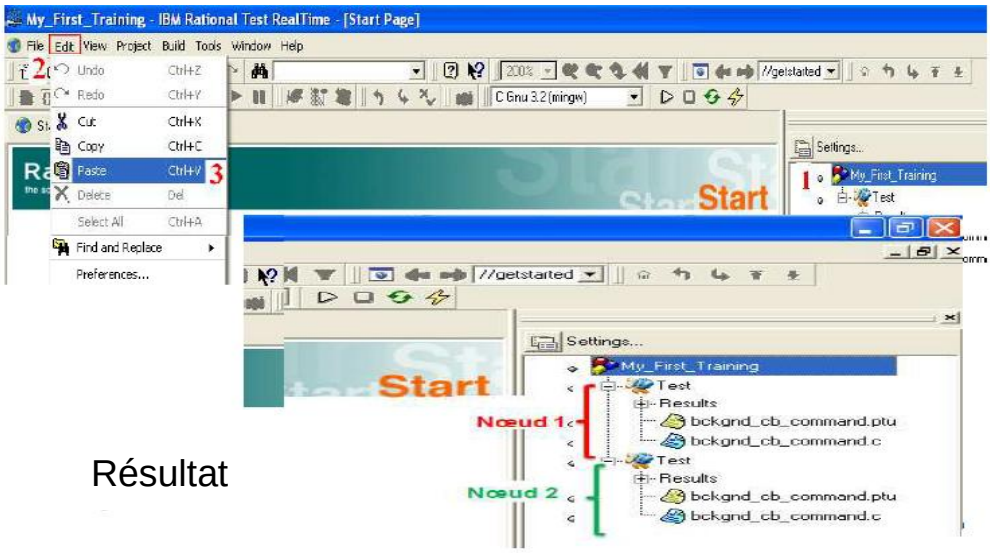
Information Mode :

- **Compact mode** : Chaque branche est stockée dans un bit au lieu d'un octet (le cas de "Pass mode")

- Configuration d'un projet (1/2)

Sélectionner Nœud « TEST» ->Edit ->Copy

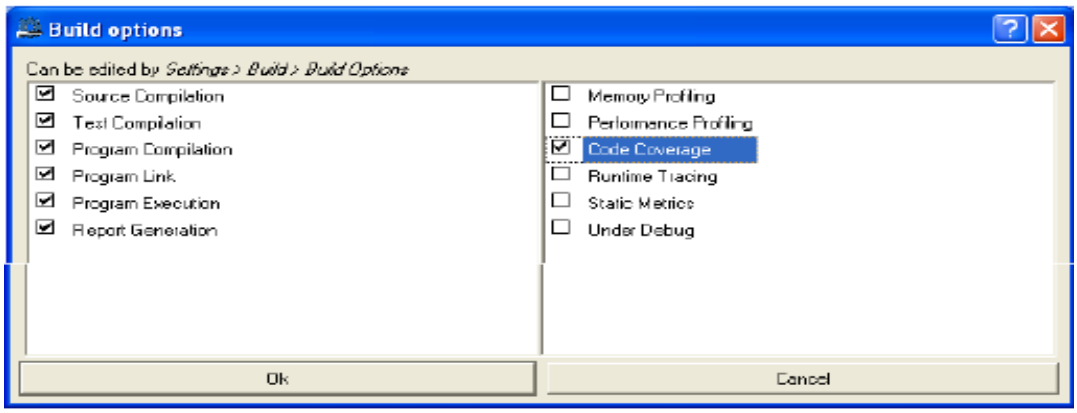
Sélectionner Projet -> Edit -> Paste



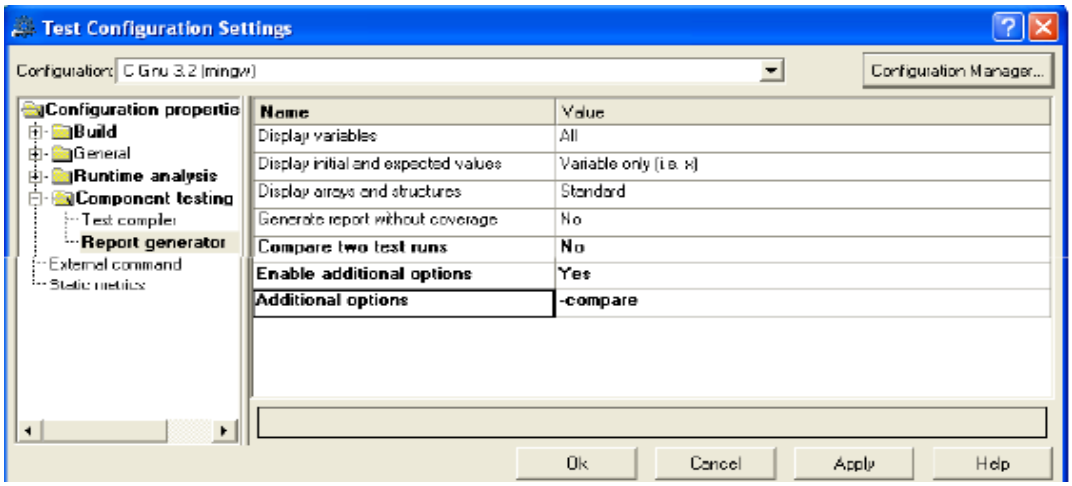
Résultat

Configuration d'un projet (2/2)

(2ème Noeud) Bouton droit -> Setting -> Build option -> Cocher « Code Coverage»



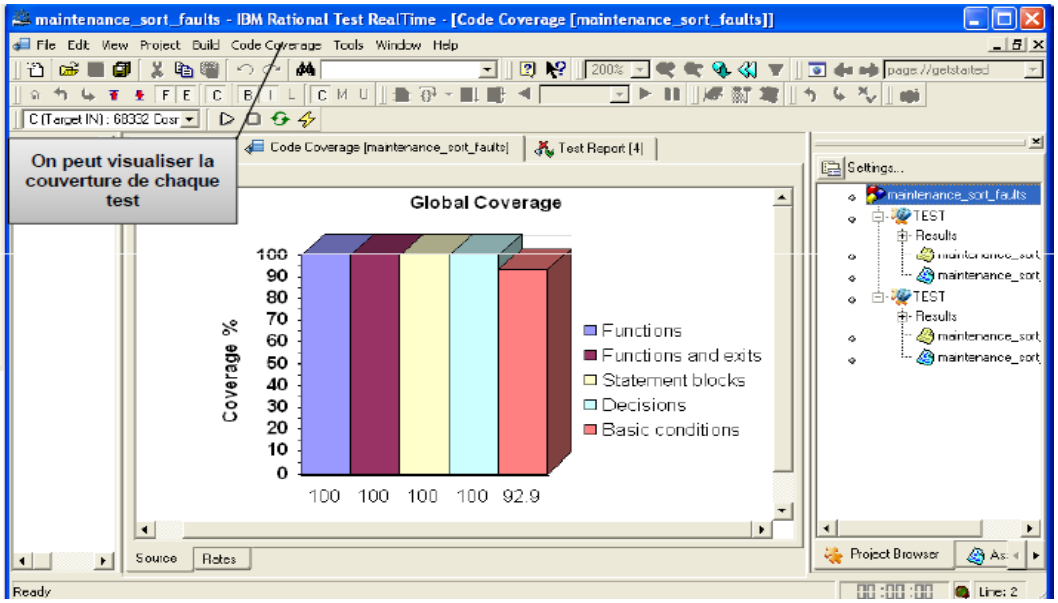
(2eme Noeud) Bouton droit -> Setting -> Component testing -> Report generator



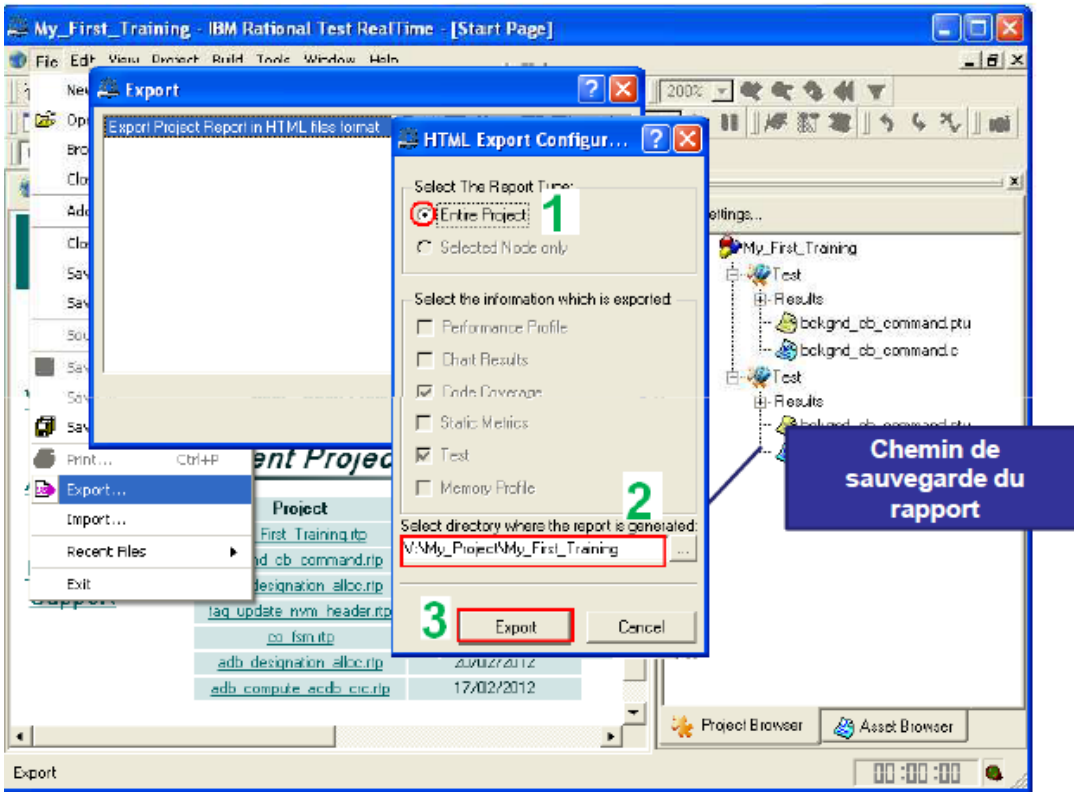
- Génération des résultats

Code Coverage (couverture de code):

Noeud de projet (bouton droit) -> View Report -> All



- Génération du rapport



- Fichiers générés par RTRT

Fichier	Description
< FileName>.*rtp*	Projet RTRT
<FileName>.*ptu*	Plan de test unitaire (script)
<FileName>.*xrd*	Résultats de test en format de Test Real time.
<FileName>.*fdc*	Fichier de couverture
.....	.....

Annexe 4 : Exemple de test avec RTRT [8]

- Le programme à tester

```
extern int f(int);
extern int global_variable;

int function ( int test )
{
int i=0;
if( i )
  {
  return 0;  La branche n'est pas couverte
  }

if( test )
  {
  return ( f( 1 )+ global_variable );Si global_variable=test=1
et                                    f(1)retourne 1
functionretourne 2
  }
else
  {
  return ( f( 0 )+ global_variable );
  }
}
```

- Squelette généré et initialisé à partir du code précédent

```
-- Tested service parameters declarations
#int test;
-- By function returned type declaration
#int ret_function;

TEST 1
FAMILY nominal
   ELEMENT
VAR global_variable,   init = 1, ev = init
    Initialise la variable globale à 1 et spécifie que
    sa valeur à la fin de l'exécution (ev) ne doit pas être
différente.

VAR test,              init = 1, ev = init
    Même chose que pour la variable précédente.
```

```
VAR ret_function,      init = 0, ev = 1
    Précise que la valeur attendue de retour du module pour
    ce cas de test doit être 1.
    !! La valeur réelle de retour est 2.


stub f (1) 1
    Spécifie qu'il doit y avoir un appel au stub simulant
    un appel de f avec 1 comme paramètre.
    La valeur de retour du stub est fixée à 1.

#ret_function = function(test);
  END ELEMENT
END TEST
-- TEST 1
```

- Le résultat de l'analyse structurelle

```
int function ( int test )
  {
  int i=0;
  if implicit else

( i )
    {
    return 0;
    }

  if( test )
    {
    return ( f( 1 )+ global_variable );
    }
  else
    {
    return ( f( 0 )+ global_variable );
    }
  }
```

La branche non couverte est bien détectée. De telles erreurs peuvent se produire si les cas de tests ne sont pas suffisamment nombreux pour couvrir toutes les branches. Il ne s'agit pas forcément d'une erreur dans le module.

Variable	Init.	Expected	Obtained
global_variable	1	1	1
test	1	1	1
ret_function	0	1	2

La valeur réelle de retour est de 2. Celle attendue est 1. Le test n’est pas valide.