



Institut Supérieur
d'Informatique, de
Modélisation et de
leurs Applications

Campus des Cézeaux
1 rue de la Chebarde
TSA 60125
CS 60026
63187 Aubière CEDEX



Société pour
l'Informatique
Industrielle

7 rue Paulin Talabot
Immeuble New Horizon
31100 Toulouse

Rapport d'ingénieur
Stage de 3^{ème} année

Filière : Informatique des Systèmes Embarqués

Réalisation du logiciel embarqué d'un calculateur de commandes de vol sur l'hélicoptère Airbus X6

Présenté par : **Nicolas MARSAUX**

Responsable ISIMA : Kun Mean HOU

Soutenance le 01 Septembre 2017

Tuteur entreprise : Julien FAIVRE

Durée du stage : 5 mois

Remerciements

Je voudrais remercier l'ensemble des employés de l'agence SII de Toulouse et notamment Antoine Leclercq, directeur de l'agence SII Sud-Ouest, et Cécile Brebion, responsable Ressources Humaine, pour m'avoir accueilli pendant ces cinq mois de stage.

Je tiens à remercier plus particulièrement mon tuteur de stage et chef de projet Julien Faivre pour son implication tout au long de ce stage et le partage de ses connaissances.

Pour finir, je remercie l'ensemble de l'équipe du projet, Marc Patry, Valentin Perez, Stephan Messegue, Steeve Remion et Chrystophe Ledru, ainsi que l'équipe Airbus, Denis Favre-Felix, Flavien Miral et Abderrahmane Brahmi pour l'aide précieuse qu'ils m'ont apportée durant le stage.

Résumé

Le but de ce stage était de développer des composants unitaires qui seront utilisés dans les commandes de vol du prochain hélicoptère X6 de la société Airbus. Ce stage a été réalisé à l'agence SII de Toulouse. Pour chaque composant, deux livrables devaient être produits, son code et le plan de tests unitaires associé. Le projet est soumis à la norme avionique DO178C au niveau A. Cela implique par exemple une indépendance entre le codeur et le testeur dans la réalisation de ces deux parties. Le stage a été directement réalisé au sein d'une équipe projet de la société SII. Tous les composants créés seront ensuite intégrés sur un processeur P2020 de Freescale.

L'ensemble des composants a été réalisé à l'aide d'extensions du logiciel Eclipse réalisées par Airbus. Tous les composants utilisés par la bibliothèque de symboles ont été développés avec un plugin permettant la gestion de versions nommée IMPACT* et une autre, nommée MARCEL*, permettant de lancer des commandes pour modifier et compiler les fichiers. Quant à la partie système, elle a été développée avec l'extension Flyworks*, pouvant réaliser l'ensemble des actions des deux autres plugins. Deux langages de programmation ont servi à l'écriture du code. La partie système a été réalisée avec le langage C et la bibliothèque de symboles a été réalisée en langage Assembleur.

Mots clés : Commandes de vol, DO178C, P2020, langage C, langage Assembleur

Abstract

The goal of this internship was to develop unitary components which will be used in flight control for the next X6 helicopter from the Airbus Company. This internship was done at SII's Toulouse office. For each component, two deliverables needed to be produced, the program and the associated unitary tests plan. The project is subjected to the DO178C norm at the level A. This for example implies an independency between the developer and the tester during the realization of these two parts. This internship is directly done within a project team in the SII Company. All the created components will then be integrated in a P2020 processor from Freescale.

All the components are done with Eclipse plugins developed by Airbus. All the components used by the symbol library have been developed with the version management plugin named IMPACT and another plugin called MARCEL, enabling the modification and the compilation of the files. All the system components have been developed with a plugin called Flyworks which can do what the other plugins can both do. Two programming languages have been used for the code writing. The system part has been realized using the language C and the symbol library has been done using the Assembly language.

Keywords: Flight Control, DO178C, P2020, language C, Assembly language

Table des Matières

Remerciements	i
Résumé.....	ii
Abstract	ii
Introduction	1
1. Le cadre du projet	2
1.1. Présentation de l'entreprise.....	2
1.1.1. Le groupe SII	2
1.2. Le projet X6.....	4
1.2.1. L'hélicoptère X6.....	4
1.2.2. Le fonctionnement des commandes de vol	6
1.3. Le management du projet	11
1.3.1. Le cycle en V	11
1.3.2. La DO178C	12
1.3.3. L'équipe projet	13
1.3.4. Les outils utilisés.....	14
2. La conception de SLIB.....	17
2.1. Les fichiers d'entrée	17
2.2. Les fichiers de sortie	19
2.3. Le travail effectué	20
2.3.1. La relecture SDD	21
2.3.2. Le codage.....	24
2.3.3. La création des tests.....	29
2.3.4. La relecture du code.....	35
3. La conception de SYST	36
3.1. Les fichiers d'entrée	36
3.2. Les fichiers de sortie	38
3.3. Le travail effectué.....	38
3.3.1. Le codage.....	40
3.3.2. Les fichiers de test	42
Conclusion	43
Lexique	v
Bibliographie	vi
Annexe 1 : Diagramme de Gantt Prévisionnel	vii
Annexe 2 : Diagramme de Gantt Définitif	viii

Table des Illustrations

Figure 1 : Agences du groupe SII	2
Figure 2 : Organisation de l'agence SII Sud-Ouest	3
Figure 3 : Organisation de la BU ASP	4
Figure 4 : Hélicoptère H225 Super Puma	5
Figure 5 : Commandes du pilote d'un hélicoptère	6
Figure 6 : Architecture du système des commandes de vol	7
Figure 7 : Composition des commandes de vol PRIM et SEC avec leur organisation industrielle	7
Figure 8 : Composition des ordinateurs PRIM et SEC	8
Figure 9 : Composants du processeur P2020	9
Figure 10 : Schéma descriptif du pipeline	9
Figure 11 : Schéma d'une planche SCADE	10
Figure 12 : Cycle en V	11
Figure 13 : Niveau de criticité de la norme DO178	12
Figure 14 : Organisation industrielle du projet	13
Figure 15 : Hiérarchie de l'équipe de projet SII	14
Figure 16 : Interface de l'extension IMPACT	15
Figure 17 : Interface de l'extension Flyworks	16
Figure 18 : Hiérarchie des fichiers d'entrées	17
Figure 19 : Interface du symbole ATAN	23
Figure 20 : Courbe d'erreur de la fonction d'approximation de la fonction atan	23
Figure 21 : Inscription des remarques sur le symbole ATAN dans le fichier PRF	24
Figure 22 : Prise en compte des remarques dans le fichier PRF	24
Figure 23 : Fenêtre de création d'objet avec IMPACT	25
Figure 24 : En-tête de la macro du symbole ATAN	26
Figure 25 : Exemple d'instructions assembleur pour le symbole ATAN	29
Figure 26 : Déclaration de variables dans le wrapper du symbole ATAN	30
Figure 27 : Identifiants dans le wrapper du symbole ATAN	30
Figure 28 : Initialisation des entrées dans le wrapper du symbole ATAN	30
Figure 29 : Appel de la macro et test dans le wrapper du symbole ATAN	31
Figure 30 : Déclaration dans le fichier ptu du symbole ATAN	32
Figure 31 : Classes d'équivalence pour le symbole ATAN	33
Figure 32 : Exemple de test pour le symbole ATAN	34
Figure 33 : Remarque fait sur le code du symbole ATAN	35
Figure 34 : Prise en compte des remarques sur le symbole ATAN	35
Figure 35 : Déclaration du service dans le fichier codda	39
Figure 36 : Remarque PRF du fichier codda et dcsl	39
Figure 37 : Fenêtre pour les commandes gradle	40
Figure 38 : Extrait du fichier d'un service dcsl	41
Figure 39 : Extrait du code C d'un service	41

Introduction

Le stage que j'ai effectué cette année a eu lieu dans l'entreprise appelée Société pour l'Informatique Industrielle, ou SII, à Toulouse. Il s'agit d'une Entreprise de Service du Numérique française présente dans 66 villes dans le monde et travaillant dans de nombreux secteurs comme la télécommunication, les énergies, le commerce ou encore l'aéronautique. Elle est amenée à travailler pour de grands groupes comme les opérateurs Orange, SFR, Bouygues, ou des sociétés majeures dans les domaines des transports, du spatial et l'aviation comme Safran, Thalès ou Airbus. Dans le cadre d'un contrat avec ce dernier, SII est amené à travailler sur les commandes de vol de leur prochain hélicoptère. C'est dans l'équipe travaillant sur ce projet que j'ai été intégré.

Conçu par Airbus Helicopters et prévu pour l'année 2020, le X6 est le nouveau projet imaginé afin de remplacer le Super Puma pour des missions dans le secteur du pétrole et du gaz, mais aussi dans la recherche, le sauvetage et le transport. Un des aspects critiques dans la conception d'un hélicoptère est le développement de ses commandes de vol. En effet, il s'agit d'éléments indispensables pour garantir le bon fonctionnement général de l'aéronef, mais aussi la sécurité des biens et surtout des personnes à bord pour garantir le bon fonctionnement général de l'aéronef, mais aussi la sécurité des biens et surtout des personnes à bord. Leur conception est donc soumise à de nombreux contrôles à chacune des étapes de son développement, que ce soit lors de la création des documents de conception ou du codage des commandes.

Le travail effectué par SII sur ce projet n'est pas de concevoir directement les commandes de vol. Cette partie est effectuée par Airbus EYY (département Avionics & Simulation d'Airbus avions), mandaté par Airbus Helicopters. En revanche, le calcul des commandes comprend l'utilisation d'opérations diverses – additions, opérations binaires, affectations, interpolations,... – dont l'implémentation est affectée à SII. Cette implémentation est divisée en quatre parties : la relecture des documents de conception, le codage de l'opération, les tests qui permettent de démontrer que l'opération est bien réalisée et pour finir, la relecture du code et des tests. Ce travail est à réaliser sur deux parties des commandes de vol, la partie système et la partie librairie, la première permettant l'appel des composants présents dans la seconde.

Dans ce rapport, je présenterai dans un premier temps le contexte de ce projet en décrivant les différentes entreprises concernées, le fonctionnement des commandes de vol et le management du projet. Ensuite, j'expliquerai le travail effectué sur la partie librairie en présentant les documents d'entrées fournis par Airbus ainsi que les livrables attendus. La présentation du développement de la partie système reprendra la même structure que celle utilisée pour la conception de la librairie.

1. Le cadre du projet

1.1. Présentation de l'entreprise

1.1.1. Le groupe SII

La société SII est une ESN, ou Entreprise de Services du Numérique, internationale. Elle a été créée en 1979 à Paris par Bernard Huvé et s'est, dans un premier temps, développée en France grâce à la création de neuf agences à travers le pays, puis à l'internationale avec la création d'une filiale en Pologne en 2006. Aujourd'hui, la société compte 22 agences dans le monde représentant un total de 66 implantations géographiques, dont 9 agences en France pour 22 implantations en France.

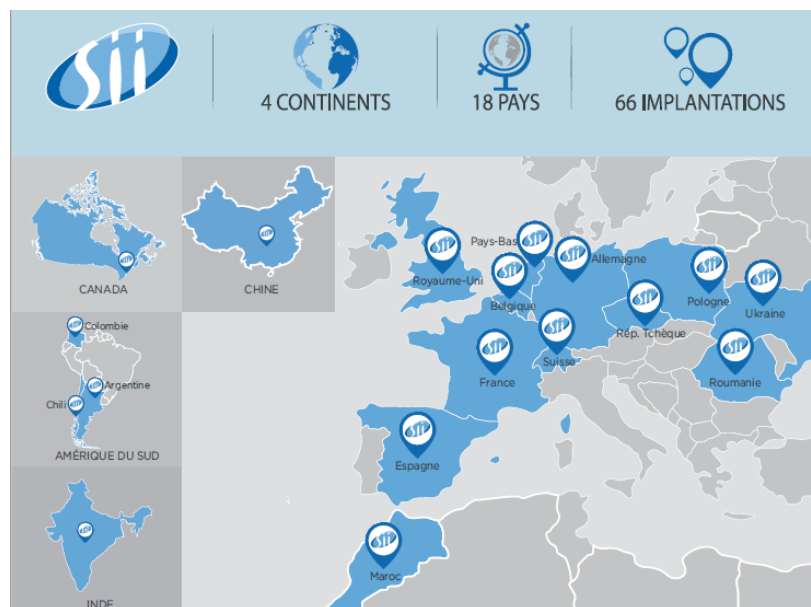


Figure 1 : Agences du groupe SII

Le groupe SII est présent dans de nombreux secteurs et notamment :

- La télécommunication et les médias
- L'aéronautique
- La défense
- L'électronique
- Les banques et les assurances
- L'automobile
- Le tourisme et le transport
- Les services publics
- L'énergie
- Le commerce et la distribution

L'organisation du groupe SII est centrée autour de deux axes, la recherche et le développement, et le conseil et le service informatique.

Au cours de l'exercice 2015-2016, la société comptait en moyenne 5226 employés et a réalisé un chiffre d'affaires de 360.1 millions d'euros.^[1]

1.1.2. L'agence SII Sud-Ouest

L'agence SII Sud-Ouest est composée de deux implantations, celle de Toulouse, créée en 2000 et celle de Bordeaux, créée en 2001. Cette agence est dirigée par Antoine Leclercq depuis 2010 et comptait en plus de 660 collaborateurs lors de l'exercice 2015-2016 et a généré un chiffre d'affaire de 45 millions d'euros.

Dû à son positionnement géographique, l'agence jouit d'une forte activité dans les domaines aéronautique, spatial, de la défense, des télécommunications et de l'automobile. Ainsi, elle peut compter parmi ses clients de grands groupes tels qu'Airbus, Thalès, Orange ou encore Continental et elle peut offrir des emplois dans les domaines du logiciel embarqué, des technologies de l'information et de la conception des systèmes aéronautiques.

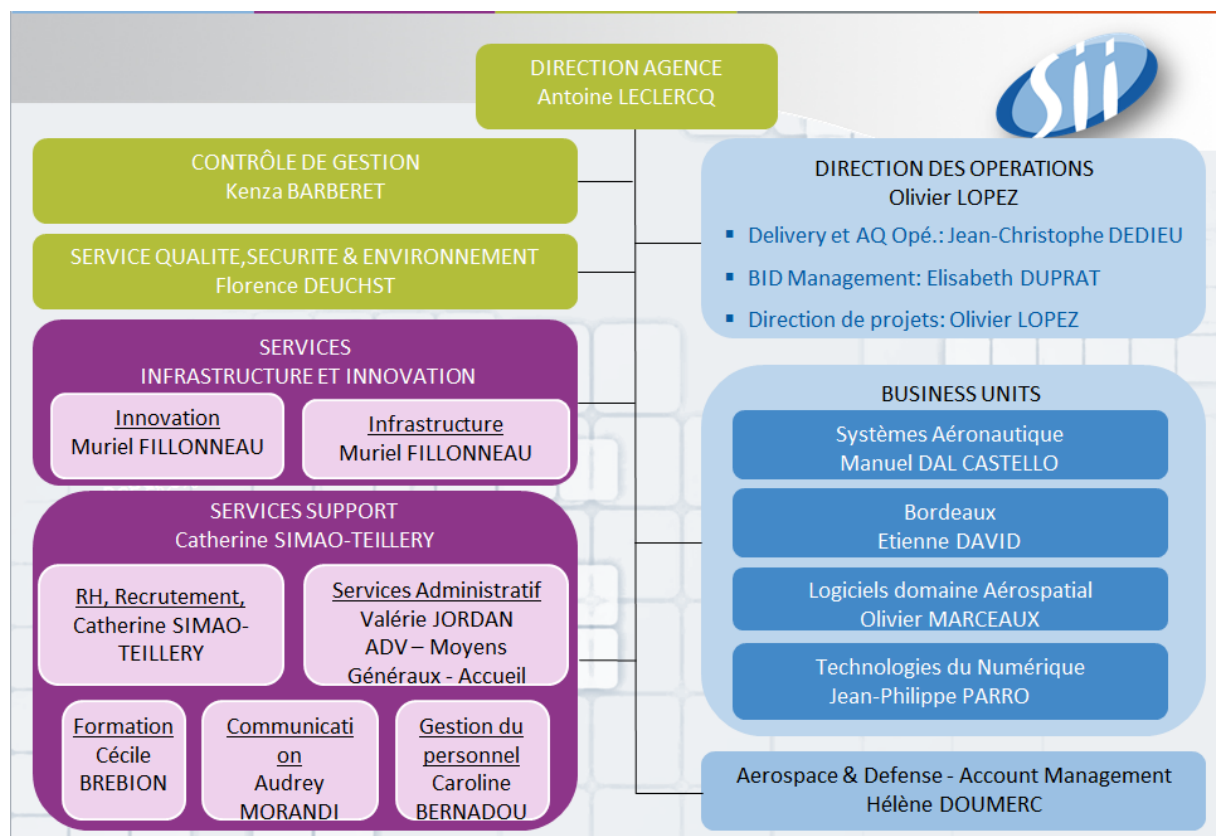


Figure 2 : Organisation de l'agence SII Sud-Ouest

1.1.3. La Business Unit Aérospatiale

La BU concernant les logiciels du domaine aérospatial, ou BU ASP, a été créée en avril 2016 dans l'agence SII Sud-Ouest. Son objectif est de répondre aux appels d'offre du domaine aérospatial de la région. Elle compte aujourd'hui environ 150 collaborateurs et est sous la direction d'Olivier Marceaux.

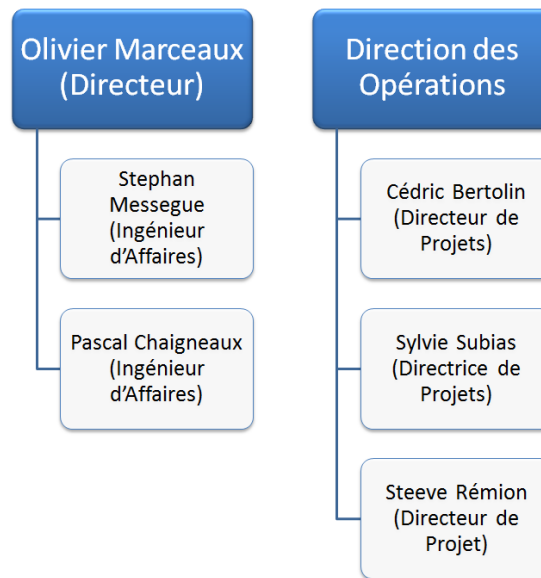


Figure 3 : Organisation de la BU ASP

Chaque ingénieur d'affaires s'occupe d'une liste de clients spécifiques dont les principaux pour cette BU sont Thalès, Airbus, notamment la branche équipementière nommée EYY, Liebherr, un constructeur travaillant entre autres dans l'aérospatiale, et Rockwell Collins, une entreprise américaine spécialisée dans l'avionique.

1.2. Le projet X6

1.2.1. L'hélicoptère X6

En juin 2015, la branche du groupe Airbus, Airbus Helicopters, lance la conception d'un nouvel hélicoptère européen, nommé le X6. Cet hélicoptère fera partie de la gamme des hélicoptères lourds et sera principalement utilisé dans le secteur pétrolier, bien qu'il puisse également être utilisé pour du transport de personnalités ou de la recherche et du sauvetage. Il aura pour objectif de remplacer les hélicoptères de type Super Puma dont la dernière génération désignée H225 Super Puma est sortie en novembre 2000.^[2]



Figure 4 : Hélicoptère H225 Super Puma

Prévu pour l'année 2020, Airbus souhaite créer un appareil bimoteur de nouvelle génération pouvant voler sous des conditions météorologiques drastiques et possédant des commandes de vol électriques, une technologie nouvelle chez les hélicoptères, mais qui est relativement commune dans le domaine avionique.

Les commandes de vol font partie des éléments principaux dans un hélicoptère puisque ce sont elles qui transmettent et interprètent les commandes du pilote en actions mécaniques et en déplacement. D'un point de vue matériel, le pilote possède trois commandes :

- Le pas collectif, qui permet d'augmenter ou de diminuer symétriquement l'angle d'incidence des pales du rotor principal, ce qui permet de modifier l'altitude de l'hélicoptère. Il s'agit d'un levier situé à gauche du siège du pilote.
- Le pas cyclique, qui modifie asymétriquement l'angle d'incidence des pales du rotor principal, faisant s'incliner l'hélicoptère vers le côté ou vers l'avant ou l'arrière. Il s'agit d'un manche situé devant le pilote.
- Le palonnier, qui permet de modifier l'angle d'incidence des pales du rotor arrière, ce qui permet à l'hélicoptère de tourner sur lui-même ou de rester fixe. Il s'agit de deux pédales situées aux pieds du pilote.[3]

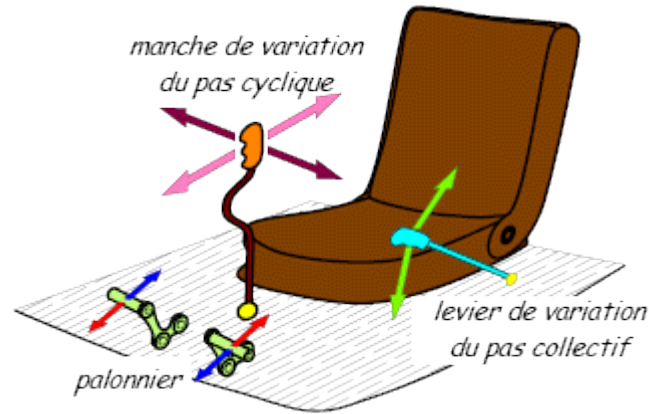


Figure 5 : Commandes du pilote d'un hélicoptère

Des capteurs présents au niveau de ces commandes envoient des données à deux processeurs, un processeur primaire et un processeur secondaire qui calculeront les modifications à appliquer aux pales des deux rotors.

Afin de développer ces commandes, quatre entreprises sont en collaboration :

- Airbus Helicopters, qui s'occupe des commandes de vol automatique
- SAGEM, qui s'occupe de la partie matérielle et des équipements logiciels des commandes de vol
- Airbus EYY, qui s'occupe de la partie logicielle des commandes de vol. Dans ce cadre, la société a fait appel à une société de service, SII.

1.2.2. Le fonctionnement des commandes de vol

Les commandes de vol de l'hélicoptère X6 sont composées de quatre ordinateurs de vol, deux ordinateurs primaires appelés PRIM* et deux secondaires appelées SEC*. Ces ordinateurs fonctionnent en parallèle et sont identiques deux à deux.

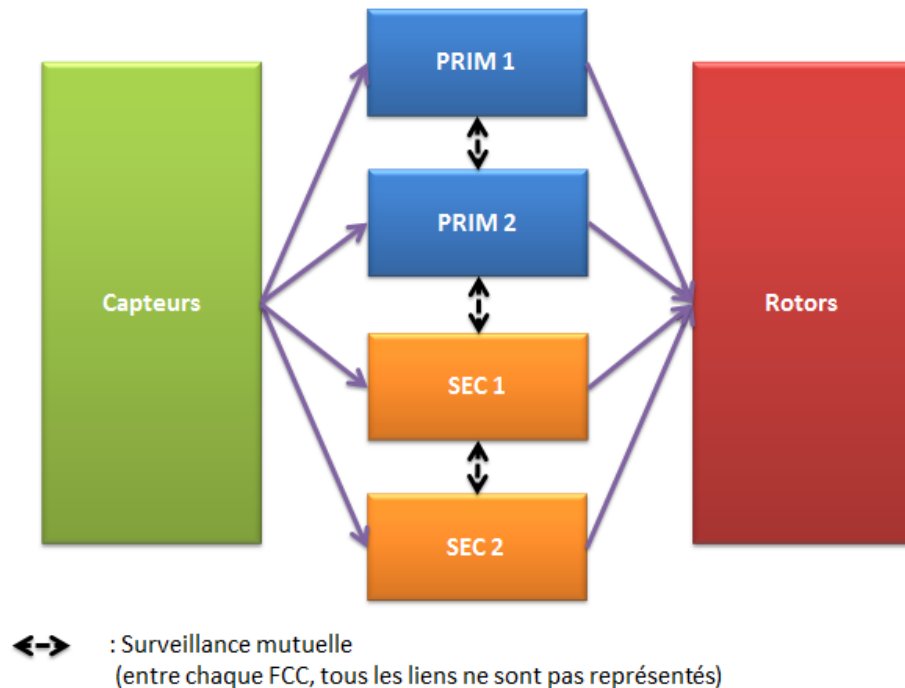


Figure 6 : Architecture du système des commandes de vol



Figure 7 : Composition des commandes de vol PRIM et SEC avec leur organisation industrielle

Les commandes de vol PRIM et SEC possèdent chacune un système de génération de code automatique et une plateforme matérielle et deux voies A et B – rigoureusement identiques au niveau structurel – utilisées pour traiter les commandes. Ces deux voies réalisent les mêmes opérations en même temps pour ensuite comparer les résultats qu'elles obtiennent. Si les résultats obtenus sont différents, cela indique un problème lors de l'exécution d'au

moins une des deux voies. Matériellement chaque voie est réalisée avec une carte mère. Sur PRIM, cette carte est notamment composée d'un processeur P2020.

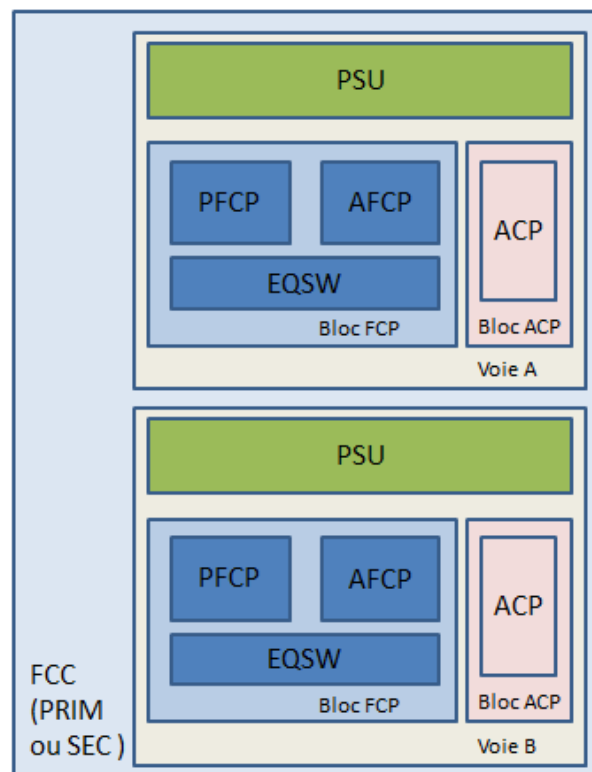


Figure 8 : Composition des ordinateurs PRIM et SEC

Les processeurs P2020

Les processeurs P2020 font partie de la gamme des microprocesseurs PowerPC, développée par les sociétés Apple, IBM et Freescale. Ils possèdent deux cœurs fonctionnant à une fréquence de 1200 MHz. Ils possèdent deux mémoires caches, une de premier niveau subdivisée en deux parties – une pour la gestion des données et une pour la gestion des instructions faisant chacun 32 KB – et une de deuxième niveau, partagée par les cœurs, faisant 512 KB.

Ils sont capables de gérer des mots codés sur 64 et sur 32 bits. Pour cela, ils possèdent plusieurs unités de traitement, deux unités simples – l'une réservée pour les mots de 32 bits et une capable de gérer les deux – et une unité multiple capable de réaliser des calculs sur deux mots de 32 bits simultanément ou un mot de 64 bits en le divisant en deux mots de 32 bits.

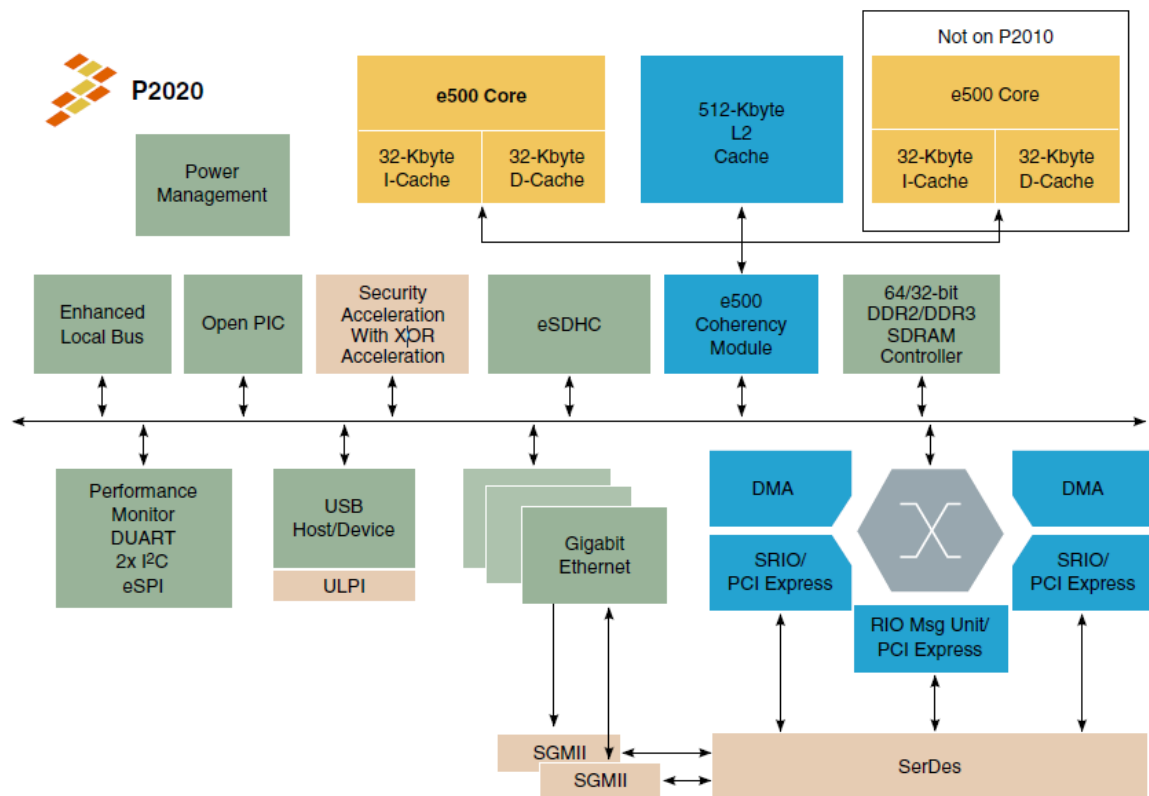


Figure 9 : Composants du processeur P2020

Les processeurs P2020 sont des processeurs RISC, pour Reduced Instruction Set Computing, caractérisés par un jeu d'instructions limité avec un format équivalent pour chacune d'entre elles. Cela permet une bonne fluidité, car le processeur peut lancer une instruction par cycle d'horloge et ainsi permettre l'utilisation d'un pipeline. Le pipeline est une technique divisant l'exécution des instructions en plusieurs étapes successives :

- Le chargement de l'instruction
- Le décodage de l'instruction
- L'exécution
- Le transfert du résultat vers la mémoire
- Le stockage du résultat

Avec cette technique, le processeur peut contenir une instruction par étape et ainsi, exécuter une instruction par cycle.[4]

Chargement	Décodage	Exécution	Transfert	Stockage					
	Chargement	Décodage	Exécution	Transfert	Stockage				
		Chargement	Décodage	Exécution	Transfert	Stockage			
			Chargement	Décodage	Exécution	Transfert	Stockage		
				Chargement	Décodage	Exécution	Transfert	Stockage	

Figure 10 : Schéma descriptif du pipeline

Le logiciel de commandes de vol exécute des cycles opératoires de 160 millisecondes divisés en 16 créneaux de 10 millisecondes chacun. Durant chacun de ces créneaux, le processeur va appeler des planches SCADE* réalisées par Airbus et stocker leurs résultats dans la mémoire.

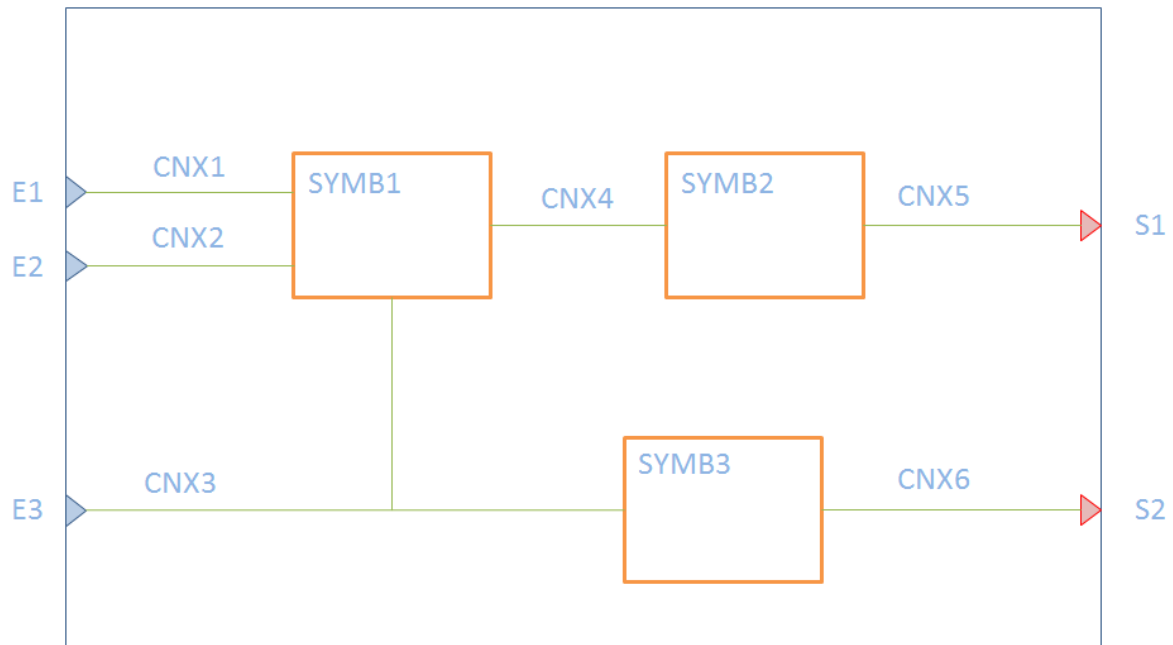


Figure 11 : Schéma d'une planche SCADE

SCADE, pour Safety Critical Application Development Environment – est un environnement de développement créé par la société Esterel Technologies, permettant de modéliser et de simuler des systèmes temps réel critique et de générer un code en langage C ou Ada correspondant aux modèles imaginés. Cet environnement est certifié par la norme DO178B au niveau A.

En utilisant cet environnement, Airbus génère de planches correspondant à une succession d'opérations, appelées symboles. Ces opérations peuvent être simples – affectation de constantes à des variables, addition, logique combinatoire – ou plus complexes – interpolation, calcul matriciel. Ces symboles sont regroupés dans une bibliothèque nommée SLIB* pour SCADE Library.

Afin de déterminer quelle planche utiliser, le processeur utilise des fonctions système regroupées dans un module nommé SYST*. Ces fonctions peuvent par exemple calculer les temps d'exécution des planches ou encore déterminer en fonction du cycle, du créneau et du mode de calcul la planche à appeler. Il existe six modes :

- OPERINI, qui est le mode d'initialisation et de synchronisation des voies
- INIT, qui est le mode d'initialisation lorsque les voies sont synchronisées
- IBIT, qui est le mode de maintenance du système

- PFBIT, qui est le mode de test avant le vol
- NOSYNC, qui est le mode d'opération sur une voie quand les voies ne sont pas synchronisées
- OPER, qui est le mode opérationnel par défaut

Le mode courant est mis à jour à chaque début de cycle.

1.3. Le management du projet

1.3.1. Le cycle en V

L'ensemble de la conception des commandes de vol est réalisée grâce à la méthode du cycle en V.

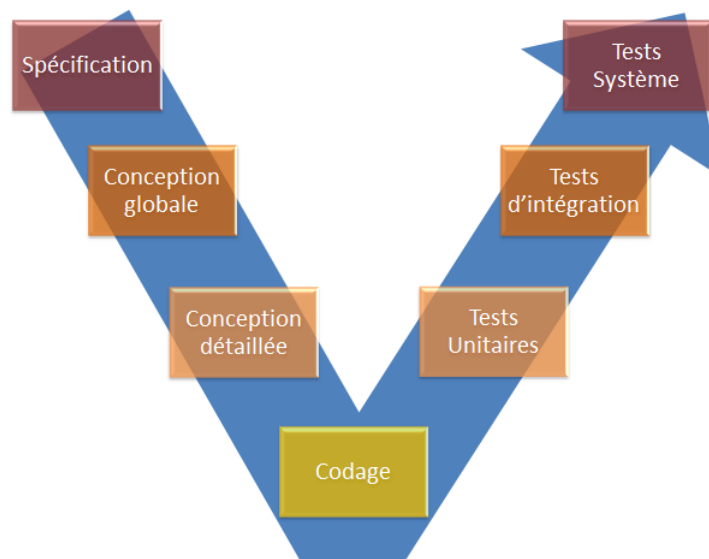


Figure 12 : Cycle en V

Il s'agit d'un modèle de gestion de projet. Elle est découpée en trois phases :

- La phase de conception, correspondant dans le schéma à la partie descendante du V
- La phase de réalisation, correspondant à la pointe du V
- La phase de test, correspondant à la partie montante du V

Son objectif est de faire correspondre une phase de conception à une phase de test, permettant ainsi une démarche plus proactive pendant cette première.^[5]

1.3.2. La DO178C

Avec la recrudescence du nombre de logiciels dans les appareils de transports et l'accroissement de leur complexité, une norme a été mise au point afin d'assurer une fiabilité maximum pour garantir la sécurité de leurs utilisateurs. Il n'est cependant pas possible d'assurer qu'un logiciel n'ait aucun bug, par conséquent, le meilleur moyen d'optimiser la fiabilité est d'assurer chaque cycle de développement. La norme donnant la liste des règles et des recommandations pour les équipements avioniques est la DO178. En Europe, l'application de cette norme est assurée par l'Agence Européenne de la Sécurité Aérienne, ou EASA. Cette agence est la seule habilitée au niveau européen à délivrer les certifications autorisant les appareils à voler sur le territoire européen. Pour déterminer s'ils peuvent être certifiés, elle vérifie que chacune des étapes décrites a été entièrement et correctement réalisée.

Cependant, tous les logiciels ne peuvent pas être traités de la même manière car un dysfonctionnement dans un n'a pas les mêmes conséquences que dans un autre. Ainsi, une classification du niveau de criticité a été établie. Ce niveau est appelé DAL pour Design Assurance Label. Ici, le niveau de criticité est DAL A.

Ce niveau de criticité détermine un ensemble d'objectifs à atteindre pour chaque étape de la conception logicielle. Ainsi, plus le niveau est élevé, plus le nombre d'objectifs sera important.

Niveau	Appellation	Explication
A	Catastrophique	Erreurs causant la perte de l'appareil
B	Sévère	Forte réduction de la capacité à faire face aux dysfonctionnements
C	Majeure	Réduction significative de la capacité à faire face aux dysfonctionnements
D	Mineure	Faible réduction de la sécurité du vol
E	Sans impact	Aucun impact sur la sécurité du vol

Figure 13 : Niveau de criticité de la norme DO178

La norme impose une indépendance à plusieurs niveaux :

- Dans le processus de vérification, si la vérification n'est pas effectuée par le développeur
- Dans le processus d'assurance qualité. Celui-ci est réalisé par une personne qualifiée et indépendante du projet.

Cette indépendance permet d'éviter la transmission des erreurs du développeur vers le testeur en passant par le relecteur. Dans le projet X6, elle est réalisée dans deux cas :

- Lors de l'élaboration de fichiers de code et de test, le codeur et le testeur doivent impérativement être différents
- À la fin du processus de création, un ingénieur qualité, extérieur au projet, effectue une relecture des fichiers produits.

Cependant, il est possible de réduire ce niveau en utilisant la dissimilarité. Plusieurs versions d'un logiciel, réalisées par des équipes différentes et indépendantes, coexistent et peuvent vérifier les résultats les uns des autres et prendre le relais en cas de défaillance. Cette dissimilarité est réalisée à deux niveaux, avec les deux ordinateurs PRIM et SEC et avec les deux voies A et B dans chacun des ordinateurs.[6]

1.3.3. L'équipe projet

Pour concevoir les commandes de vol de l'hélicoptère X6, quatre entités sont concernées, Airbus Helicopters, Airbus EYY, SAGEM et SII. Chacune travaille sur des aspects différents à l'exception d'Airbus EYY et SII, la première ayant contractualisé la seconde pour l'aider sur le développement des fonctions système et de la bibliothèque de symboles.

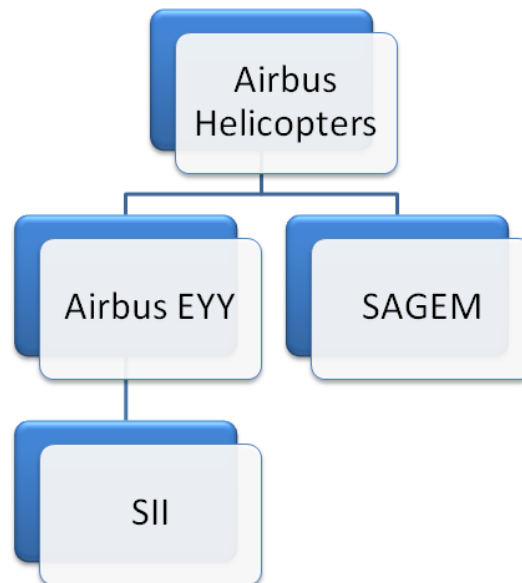


Figure 14 : Organisation industrielle du projet

Cependant, dans cette organisation, les acteurs sont régulièrement en attente d'entrées provenant d'autres maillons de la chaîne. Ceci provoque des problèmes de synchronisation engendrant des trous de charge chez les acteurs en attente.

Concernant l'équipe SII, celle-ci est composée comme suit : un chef de projet, un leader technique et une équipe de développeurs. Le premier dirige les autres et est lui-même dirigé par un directeur de projet. D'autre part, un ingénieur d'affaires gère la partie commerciale du projet et un ingénieur qualité s'assure que le projet respecte la norme DO178C et que le travail réalisé et le niveau de qualité attendu par le client.

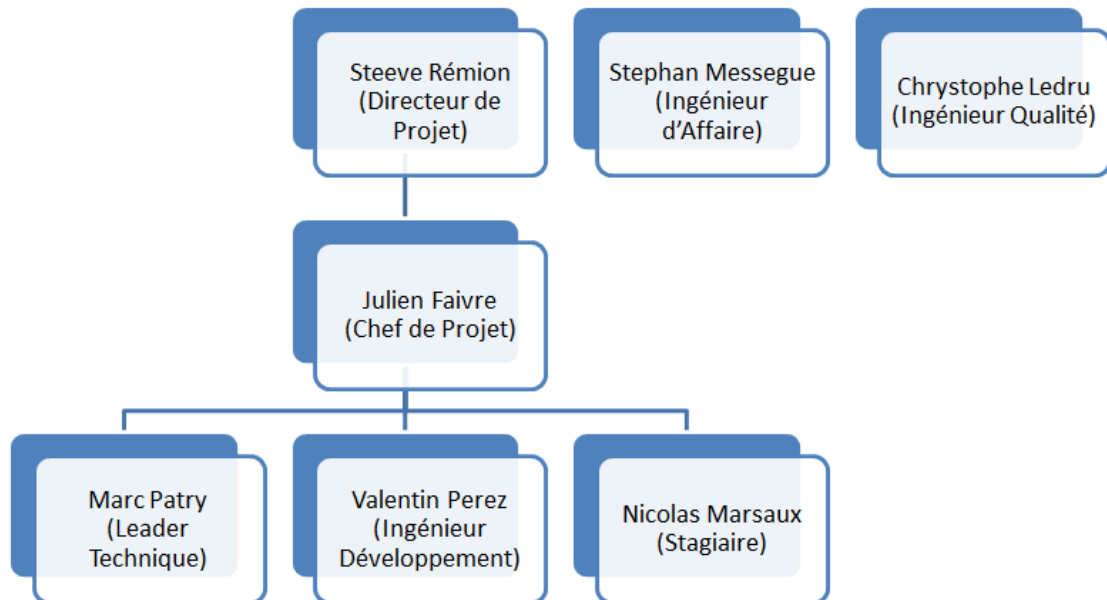


Figure 15 : Hiérarchie de l'équipe de projet SII

1.3.4. Les outils utilisés

L'ensemble du projet est réalisé sur des outils mis à notre disposition par Airbus. Cependant SLIB et SYST sont conçus sur des outils différents. En effet, Airbus est actuellement en train de développer de nouveaux outils permettant une nouvelle méthode de travail par l'automatisation des développements logiciels. Mais ces outils n'étant pas opérationnels au début du projet, la conception de SLIB est réalisée sur des anciens outils et les symboles créés seront ensuite portés sur les nouveaux. Malgré tout, ces outils ont un aspect en commun car ils sont tous des extensions du logiciel Eclipse.

Pour SLIB, deux extensions sont utilisées, IMPACT et MARCEL. IMPACT, pour Integrated Management Process And Configuration Technic ou Technique de Configuration et de gestion de processus intégrés, est une interface de gestion de version pour les projets.

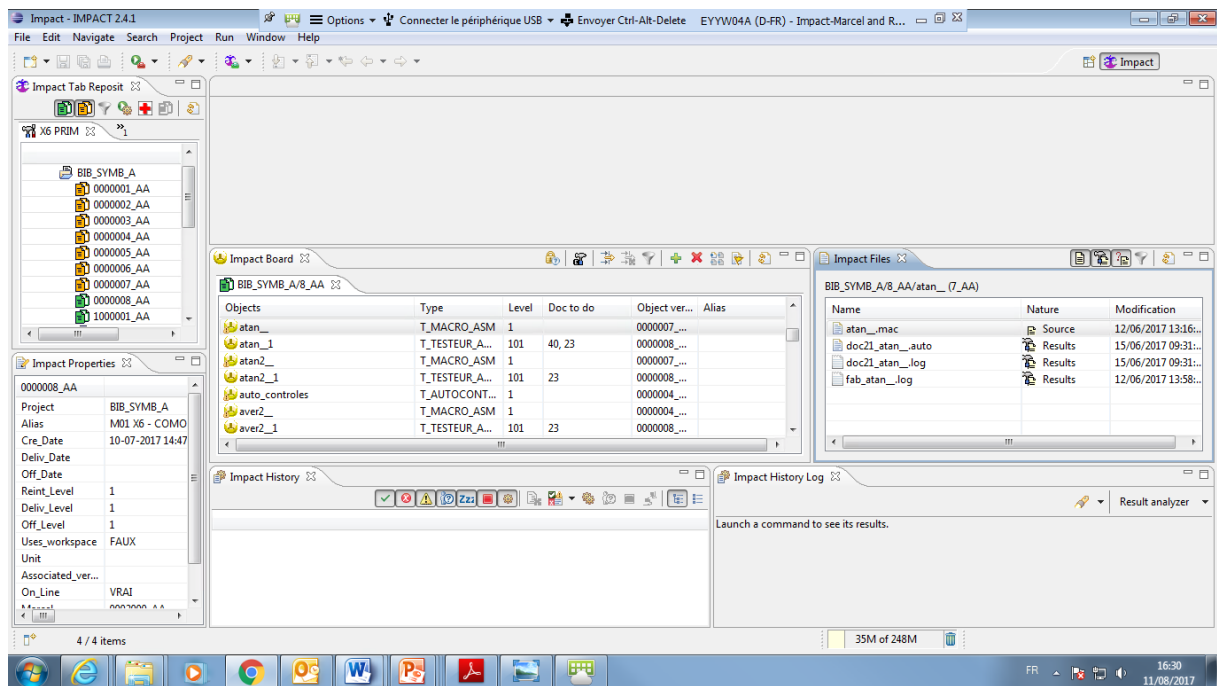


Figure 16 : Interface de l'extension IMPACT

Cette interface permet de gérer l'ensemble des versions d'un projet et permet ainsi de récupérer facilement d'anciennes versions. Elle permet également de vérifier l'indépendance codeur-testeur en affichant un message d'avertissement si un des deux souhaite accéder à un fichier qu'il ne doit pas modifier. Toutes les versions des fichiers sont stockées sur un serveur appartenant à Airbus afin de pouvoir les partager avec l'ensemble de l'équipe projet. Pour permettre les modifications, la compilation et l'exécution des fichiers, Eclipse utilise un autre plugin nommé MARCEL, pour Mécanisme d'aide à la réalisation et au contrôle d'état logiciel. Cette extension permet de lancer des commandes permettant d'effectuer ces différentes actions en leur donnant les numéros de version de fichier, le nom de ce dernier et le nom de la commande.

Pour la conception de SYST, un autre plugin est utilisé : Flyworks. Ce plugin permet, comme MARCEL et IMPACT, de gérer des versions de fichiers, de les modifier et de les compiler et les exécuter.

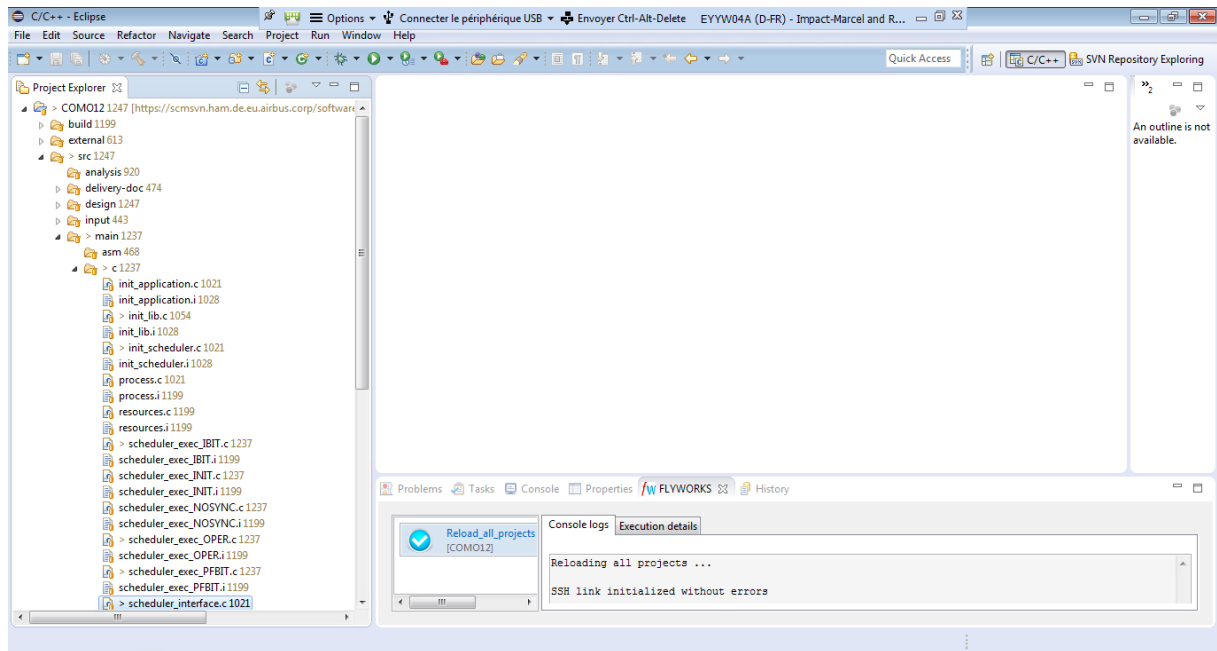


Figure 17 : Interface de l'extension Flyworks

Pour cela, Flyworks fonctionne avec deux outils :

- SVN, qui est un outil de gestion des fichiers fonctionnant avec un serveur afin de stocker les fichiers. Chaque nouveau dépôt occasionne un nouveau numéro de version. Ainsi, avec ce numéro et l'adresse du serveur, l'utilisateur peut récupérer n'importe quel fichier.
- Gradle, qui est un moteur permettant d'effectuer des tâches telles qu'exécuter des tests ou assurer la qualité d'un code. Il fonctionne à l'aide d'un fichier xml, donnant la liste des actions possibles dans le cadre du projet.

2. La conception de SLIB

2.1. Les fichiers d'entrée

Les fichiers d'entrée de la partie SLIB sont tous les fichiers que la compagnie Airbus fournit à SII afin de permettre aux développeurs de concevoir les différents symboles.

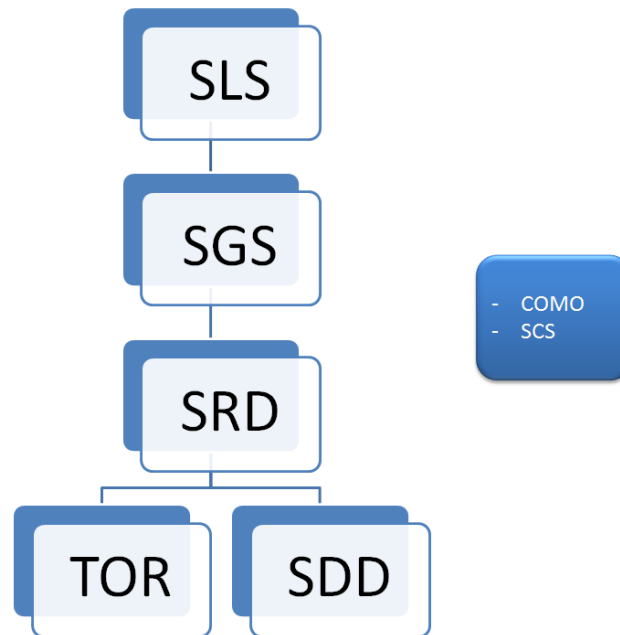


Figure 18 : Hiérarchie des fichiers d'entrées

Le Mémo de Coordination

Le COMO* – pour Coordination Memo – contient une liste de tâches à effectuer établies par Airbus. Ce dernier détermine le contexte de chacune des tâches en donnant leurs contenus, la liste des fichiers livrables, les dates prévisionnelles de livraison et le lieu où ses tâches doivent être effectuées. Ce contexte est ensuite discuté lors d'une réunion entre Airbus et le chef du projet SII afin de déterminer le coût de chacune des tâches ainsi que des dates finales de livraison.

Le Document d'Exigences Opérationnelles

Le TOR*, Tool Operational Requirements, est un document donnant la liste des exigences à respecter pour permettre la génération automatique du code des composants des commandes de vol de l'hélicoptère X6. Ces exigences prennent généralement la forme d'une interface donnant la liste des paramètres d'entrée et de sortie du composant.

Le Document de Spécification SCADE

Ce document, nommé SLS* pour SCADE Library Specification, permet de donner la liste des composants qui seront intégrés à l'environnement SCADE. Pour chacun de ces éléments, le document donne la représentation graphique du composant, une description de sa fonction, la liste de ses entrées et de ses sorties, son algorithme et une liste d'exigence à respecter. Ce document est utilisé pour constituer le fichier SDD*.

Ce document contient également pour certains symboles, un plan de tests obligatoires appelés Tests nominaux. Ces tests sont à ajouter dans le fichier au format .ptu, en plus des plans de tests choisis.

Le Document d'Exigences Logicielles

Le SRD*, Software Requirements Document, est un document contenant une liste d'exigences que les composants logiciels doivent respecter à la fois dans leur design et dans leur architecture. Chacune de ces exigences est tracée afin d'assurer qu'aucune d'entre elles ne soit en contradiction avec une autre et qu'elles soient toutes justifiées. Cette traçabilité se fait à partir d'un document de plus haut niveau appelé SGS pour Software General Specification mais ces exigences font également appel à d'autres documents, notamment le SLS et le TOR.

Le Document de Conception Logicielle

Ce document est appelé SDD, pour Software Design Data. Ce document est écrit par le concepteur de la librairie SCADE et permet de décrire tous les symboles et de toutes les macros à développer. Il contient ainsi pour chacun d'eux une description de leur but, une interface et un algorithme à implémenter. Chaque détail de cette description doit être justifié. Cette justification doit être faite dans le document ou en faisant référence au SRD. Toute la description doit être précise afin que le développeur n'ait pas à faire de l'interprétation. En effet, c'est uniquement sur ce document qu'il doit baser son programme assembleur ou son plan de test unitaire. Afin de s'assurer que ce soit le cas, l'équipe de développeur relit le document et le compare avec le SRD afin de vérifier chacune des justifications et leur véracité. Si une erreur est trouvée, le concepteur doit la corriger. C'est uniquement quand toutes les erreurs sont corrigées que le développement peut commencer.

Le document de règles de codage

Le SCS*_ASM, pour Software Coding Standard for Assembly, est un document regroupant toutes les règles de codage devant être implémentées afin que les différents programmes écrits en langage Assembleur respectent la norme DO178C. Le document couvre chacun des aspects du code (nommage des variables et des types, registres autorisés et interdits, commentaires...). Il existe trois catégories de règles : obligatoires, recommandés et non applicables. Seules les règles obligatoires sont vérifiées par les experts qualité.

2.2. Les fichiers de sortie

Les fichiers de sortie correspondent à tous les fichiers créés par les employés de la société SII lors du processus de création des composants de la bibliothèque SLIB.

Le Formulaire de Relecture

Le PRF*, Proof-Reading Form, est un document tableur permettant de regrouper l'ensemble des remarques des codeurs lors d'une relecture du fichier SDD ou du code. Dans celui-ci, le relecteur fait une liste des différentes remarques qu'il a pu trouver dans le fichier SDD ou dans le code. Une fois le document terminé, le créateur vérifie les remarques et réalise les modifications qu'il juge nécessaires en laissant une trace de ses choix dans le document à côté des remarques correspondantes. Une fois terminé, le relecteur vérifie chacune des modifications et les valide s'il estime qu'elles sont pertinentes avec ses remarques. Sinon, il explique dans un commentaire pour quelle raison il demande un autre changement. Ce processus continue jusqu'à ce que toutes les remarques soient validées. Il est composé de 5 parties :

- Un en-tête donnant la version du document relu, le nom des personnes concernées ainsi que l'avancement du document
- Un en-tête complémentaire donnant la liste des documents annexes (SRD, TOR, SLS dans le cas d'une relecture SDD) utilisés ainsi que leurs versions et les exigences à relire
- La liste des remarques et des modifications effectuées
- La liste des exigences à respecter dans la création du code ou du fichier SDD
- La liste des exigences à respecter pour que le fichier PRF soit le plus lisible et compréhensible possible

Le fichier de code

Il s'agit d'un fichier au format .mac contenant l'implémentation des différents composants des commandes de vol. Il comprend deux parties :

- Un en-tête comprenant une description de la macro assembleur implémentée, une liste de ses paramètres d'entrée et de sortie ainsi les constantes et les rémanents associés
- Le corps de la macro contenant son implémentation

Le fichier doit respecter de nombreuses contraintes dans sa mise en forme et dans la gestion des instructions utilisées.

Les fichiers de tests

Pour générer les tests, on crée une planche contenant uniquement le symbole à tester. En donnant différentes valeurs pour chacun des paramètres de la macro, on peut ainsi tester son fonctionnement.

Il comprend deux fichiers :

- Le wrapper, qui permet la création de la planche et le placement des paramètres dans la mémoire. Il est au format env.s.
- La liste des tests, permettant de créer des cas de tests en choisissant les valeurs des paramètres et en testant les valeurs des résultats. Il est au format .ptu.

2.3. Le travail effectué

Le travail est effectué sur la bibliothèque SLIB est divisé en un ensemble composé de quatre activités répétées pour chaque symbole:

- la relecture du fichier SDD
- le développement du code
- le développement des tests
- la relecture du code

Plus tard dans le projet, une autre activité sera ajoutée, la relecture des tests. Cette activité nécessite un autre fichier, le SUTS, Software Unit Test Standards, contenant la liste des règles à observer lors de la création des tests. Ce fichier n'est cependant pas disponible à l'heure actuelle.

Généralement, pour un symbole, deux développeurs sont nécessaires, un réalisant la relecture SDD et le code tandis que l'autre développe les plans de tests et relis le code.

Afin d'expliquer plus en détail les différentes étapes de conception des symboles, un exemple sera utilisé, celui du symbole atan, permettant de calculer la valeur de l'arc tangente d'un nombre réel.

2.3.1. La relecture SDD

La première étape dans la création d'un symbole consiste à prendre connaissance de la nature de ce dernier, de son interface et de son algorithme. Il n'existe pas d'instruction assembleur permettant de faire directement le calcul de l'arc tangente. Par conséquent, l'équipe de conception du design des symboles a choisi de calculer une approximation polynomiale de la fonction. Cette approximation est donnée sous la forme :

$$G(x) = x * (1.0 + (\frac{x^2(P0 + P1x^2)}{Q0 + x^2}))$$

Cette fonction donne une approximation en degré de atan sur l'intervalle $[0, 2 - \sqrt{3}]$. Sur cet intervalle, une précision relative inférieure à 10^{-3} est requise et une inférieure à 2.10^{-9} nous est promise. Afin de calculer l'approximation sur l'ensemble de l'intervalle $]-\infty ; +\infty[$, on utilise plusieurs formules propres à atan :

$$\text{atan}(x) = \frac{\pi}{6} + \text{atan}(\frac{\sqrt{3}x - 1.0}{\sqrt{3} + x})$$

$$\text{Pour } x > 1, \text{atan}(x) = \frac{\pi}{2} - \text{atan}(x)$$

Ainsi on obtient :

$$\text{Pour } x \in [0, 2 - \sqrt{3}], \text{AppArctan}(x) = G(x)$$

$$\text{Pour } x \in [2 - \sqrt{3}, 1], \text{AppArctan}(x) = \frac{\pi}{6} + G(\frac{\sqrt{3}x - 1.0}{\sqrt{3} + x})$$

$$\text{Pour } \frac{1}{x} \in [0, 2 - \sqrt{3}], \text{AppArctan}(x) = \frac{\pi}{2} - G(\frac{1}{x})$$

$$\text{Pour } \frac{1}{x} \in [2 - \sqrt{3}, 1], \text{AppArctan}(x) = \frac{\pi}{3} - G(\frac{\frac{\sqrt{3}}{x} - 1.0}{\sqrt{3} + \frac{1}{x}})$$

On peut connaître également le résultat pour $x < 0$ en utilisant la formule $\text{atan}(-x) = -\text{atan}(x)$.

Ces formules sont alors retranscrites en l'algorithme suivant :

```

NI1 = 0
VI1 = ABS(E1)

IF(VI1>1.0)
THEN
    VI1=1.0/VI1
    NI1=2
END_IF

IF(VI1>2-SQRT(3))
THEN
    VI1 = ((SQRT(3)*VI1)-1)/(SQRT(3)+VI1)
    NI1 = NI1 + 1
END_IF

VI2 = VI1*VI1
VI2 = VI2*(P0+P1*VI2)/(Q0+VI2)
VI1 = VI1*(1+VI2)

IF(NI1>1)
THEN
    VI1 = -VI1
END_IF

VI1 = VI1 + TAB_BIAS[NI1]

IF(E1<0.0)
THEN
    VI1 = -VI1
END_IF

S1 = VI1*RAD2DEG
    
```

Avec $TAB_BIAS = [0.0 ; \frac{\pi}{6} ; \frac{\pi}{2} ; \frac{\pi}{3}]$

Une valeur approchée nous est donnée pour chacune des constantes utilisées, c'est-à-dire ; $\frac{\pi}{6}$, $\frac{\pi}{2}$, $\frac{\pi}{3}$, SQRT(3), 2-SQRT(3), P0, P1, Q0 et RAD2DEG, cette dernière permettant de convertir une valeur exprimée en radian en une valeur exprimée en degré.

Le document SDD fait référence à deux exigences du document SRD, la première spécifie que l'interface du composant doit être la même que celle explicitée dans le document TOR. La seconde, quant à elle, spécifie que l'algorithme du symbole doit avoir le même résultat que celui explicité dans le document SLS.

Concernant le TOR, l'interface présentée est comme suit :

Nom	m_ATAN		
Paramètres	Type	Nature d'entrée ou Sortie	Optionnel ou N/A
NNN	SERIAL NUMBER	N/A	N/A
E1	Réel	Variable ou Constante	N/A
S1	Réel	Sortie	N/A

Figure 19 : Interface du symbole ATAN

Le symbole possède donc deux paramètres d'entrée, NNN et E1, et un paramètre de sortie, S1. E1 correspond au réel dont on veut connaître l'arc tangente et S1 au résultat attendu. NNN, quant à lui, est un identifiant donné au symbole.

Concernant le fichier SLS, l'algorithme spécifie uniquement que le symbole retourne l'arc tangente en degré d'un paramètre passé avec une précision inférieure à 10^{-3} .

Après relecture, Une première erreur est repérée. En effet, la fonction d'approximation est supposée retourner un résultat en degré. Cependant, l'algorithme du document SDD réalise une instruction pour convertir un résultat en radian en un résultat en degré après la complétion de la fonction d'approximation. Par conséquent, une erreur est présente dans l'algorithme ou dans l'unité de la sortie de la fonction.

Après cela, il reste une dernière étape, vérifier que la fonction d'approximation permet bien de retourner l'arc tangente avec la précision promise. Pour cela, on utilise le logiciel matlab. On obtient la courbe d'erreur suivante :

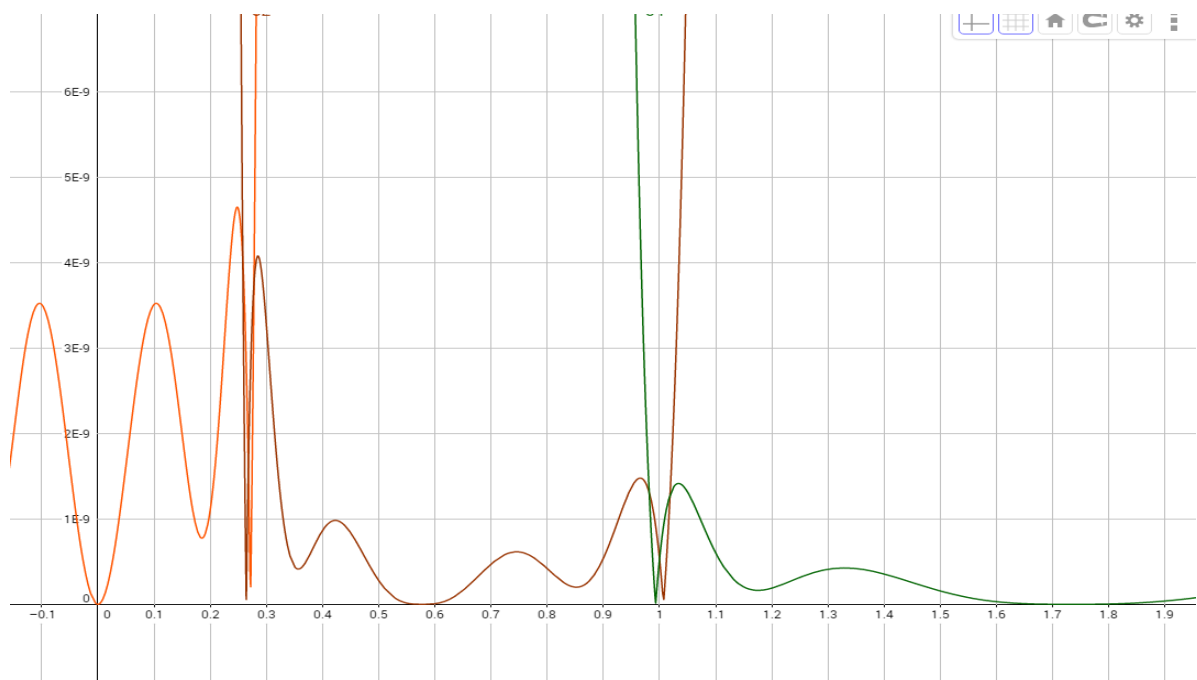


Figure 20 : Courbe d'erreur de la fonction d'approximation de la fonction atan

Lors de la création de cette courbe, on remarque que l'erreur précédemment trouvée venait de l'unité de la fonction car c'est uniquement en convertissant le résultat en degré que l'on obtient l'erreur attendue. Cependant, une deuxième apparaît. On remarque, en effet, que l'erreur relative est supérieure au $2 \cdot 10^{-9}$ promis entre 0.05 et 0.32.

Pour clore la relecture, il reste maintenant à écrire ces deux remarques dans le fichier PRF pour que les erreurs soient corrigées.

NMA	min	p 80 / 448	E_0H1_SDDSLI B_ATAN_02600	DE21	The result of AppArctan(x) is in radian instead of in degree
NMA	min	p 80 / 448	E_0H1_SDDSLI B_ATAN_02600	DE21	The relative precision is sometimes above 2E-9

Figure 21 : Inscription des remarques sur le symbole ATAN dans le fichier PRF

Une fois tous les symboles relus, le fichier PRF est envoyé à l'équipe écrivant le fichier SDD pour qu'il soit corrigé. Une fois ceci fait, l'équipe nous envoie un nouveau fichier SDD et inscrit les modifications apportées dans le fichier PRF. Ces modifications sont ensuite vérifiées et validées dans ce même fichier.

The result of AppArctan(x) is in radian instead of in degree	OK	DONE	corrected	OK	draft 7
The relative precision is sometimes above 2E-9	OK	DONE	relative precision corrected to $5 \cdot 10^{-9}$	OK	draft 7

Figure 22 : Prise en compte des remarques dans le fichier PRF

Une fois toutes les modifications validées, le codage des symboles peut commencer.

2.3.2. Le codage

Le codage des symboles se fait en utilisant l'interface IMPACT. Dans un premier temps, on utilise une commande MARCEL afin de lancer une fenêtre de création d'objet. On peut ainsi créer un nouvel objet nommé atan__ de type T_MACRO_ASM.

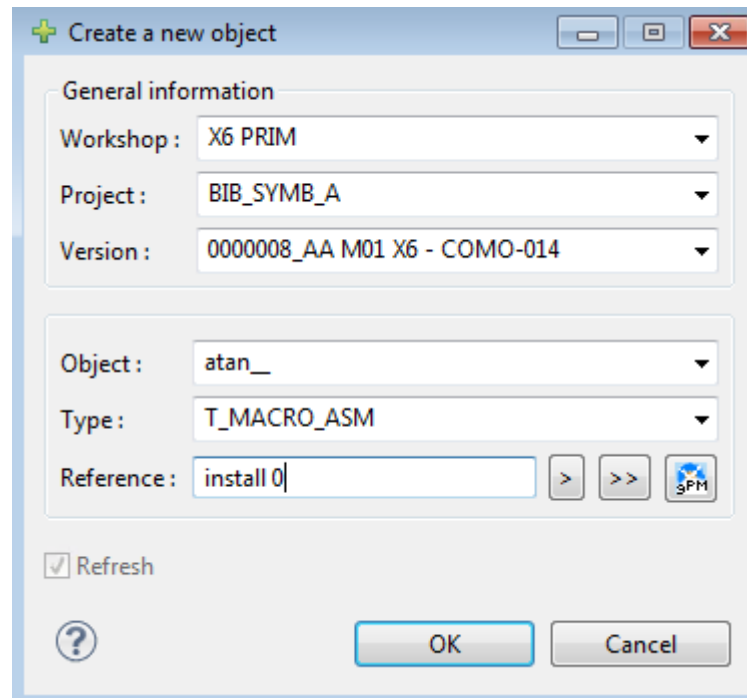


Figure 23 : Fenêtre de création d'objet avec IMPACT

À sa création, le nouvel objet comprend un fichier atan__.mac vide. La première étape dans la complétion du fichier de code est de créer l'en-tête. Celui-ci contient plusieurs champs :

- Le nom de la voie : la voie A
- Le nom de la macro utilisée par le symbole : m_ATAN
- Une courte description de l'objectif de la macro
- L'interface du symbole contenant
 - o Ses entrées avec leur type
 - o Ses sorties avec leur type
 - o Ses entrées constantes
 - o Ses rémanents, c'est-à-dire les variables utilisées pouvant être utilisées et modifiées lors de différents appels de la macro

```

;-----
;- LANE A
;-----
;- Macro-operation m_ATAN
;-----
;- Description :
;-     computing the arctangent of a real data
;-
;-----
;- Interface
;-   Input :
;-     NNN : Occurrence number
;-           (Parameter type: SERIAL NUMBER)
;-     E1  : Input E1
;-           (Parameter type: SLIB_t_NUM)
;-
;-   Output :
;-     S1  : Output S1
;-           (Parameter type: SLIB_t_NUM)
;-
;-   Constants :
;-     <None>
;-
;-   Long-lived :
;-     <None>
;-
;-----

```

Figure 24 : En-tête de la macro du symbole ATAN

Une fois ceci terminé, on peut commencer à implémenter le code. Cependant, il ne suffit pas de convertir l'algorithme du document SDD en langage assembleur pour obtenir la macro. En effet, de nombreuses règles liées à la norme DO178C et notamment à son niveau A. L'une des plus importantes est notamment l'obligation d'un code linéaire, c'est-à-dire que la macro doit toujours avoir le même temps d'exécution quelle que soit la valeur de ses paramètres. Or, ici l'algorithme présenté dans le document SDD n'est pas linéaire. En effet, la présence des conditions IF ne permet pas d'avoir toujours le même temps d'exécution.

Pour remédier à cela, il faut modifier ces conditions afin que les branches THEN et ELSE aient les mêmes temps d'exécution. Cela est permis grâce à l'instruction assembleur isel et ses dérivés qui déterminent la valeur à donner à un registre en fonction du résultat enregistré lors d'une condition. Cela nécessite cependant que les variables affectées soient présentes dans les deux branches mais aussi que les calculs effectués dans chacune de ces branches soient effectuées quel que soit le résultat de la condition.

Ainsi, la condition suivante :

```

IF(VI1>1.0)
THEN
    VI1=1.0/VI1
    NI1=2
END_IF

```


devient :

```
VI1_tmp = 1.0/VI1
IF(VI1>1.0)
THEN
    VI1 = VI1_tmp
    NI1=2
ELSE
    VI1 = VI1
    NI1=NI1
END_IF
```

Cependant, cette instruction pose un nouveau problème qui n'avait pas lieu auparavant. L'instruction $VI1_tmp = 1.0/VI1$ peut maintenant engendrer une erreur si $VI1$ a pour valeur 0.0. Ainsi, il faut maintenant tester la valeur de $VI1$ avant de pouvoir faire cette instruction.

L'algorithme devient alors :

```
IF(VI1==0.0)
    denom = 1.0
ELSE
    denom = VI1
END_IF
VI1_tmp = 1.0/denom
IF(VI1>1.0)
THEN
    VI1 = VI1_tmp
    NI1=2
ELSE
    VI1 = VI1
    NI1=NI1
END_IF
```

Il existe également de nombreuses règles de mise en forme de code, notamment en ce qui concerne les commentaires. Il en existe trois types :

- Les commentaires d'algorithme, ils doivent obligatoirement référencer l'algorithme du SDD ou, si celui-ci a été modifié en début de fichier de code, à ce nouvel algorithme. Il doit toujours commencer en début de ligne et être identifié par les caractères ;#
- Les commentaires de registres, ils permettent de déterminer la réservation et la libération des différents registres utilisés. Ils doivent obligatoirement débiter par les caractères ;\$. Les commentaires de réservation doivent se situer sur la ligne précédant l'instruction de réservation ou sur la ligne courante. Les commentaires de libération, quant à eux, doivent se situer sur la ligne suivant la libération ou sur la ligne courante.
- Les commentaires libres, permettant généralement d'introduire une instruction intermédiaire dans l'algorithme ou de justifier un changement dans ce dernier. Il doit avoir la même indentation que l'algorithme et commence toujours par les caractères ;;

Une autre règle importante concerne les labels utilisés. Ceux-ci se situent toujours et uniquement au début de la macro – sous la forme `_<nom du symbole>_&&NNN:` – ou avant une branche ou une condition – sous la forme `_<nom du symbole>_<nom de la branche>_<identifiant de la branche>_&&NNN`.

Ainsi, l'instruction précédemment citée devient en langage assembleur :

```

;#      LINEAR IF (VI1 == 0.0)
        _ATAN_LINEAR_IF_1_&&NNN:
        efscmpeq    cr0,r4,r12

;#      LINEAR THEN denom := 1.0 ELSE denom := VI1
        _ATAN_LINEAR_THEN_ELSE_1_&&NNN:
        iselgt      r6,r11,r4                                ;$ reserve r6 as denom

;#      LINEAR ENDIF
        _ATAN_LINEAR_ENDIF_1_&&NNN:

;#      pre-calculation of VI1_inv:=1.0/denom
        efsdiv      r3,r11,r6                                ;$ reserve r3 as 1.0/denom
                                                         ;$ free r6

        li          r5,2                                    ;$ reserve r5

;#      LINEAR IF (VI1>1.0)
        _ATAN_LINEAR_IF_5_&&NNN:
        efscmpgt    cr0,r4,r11

;#      LINEAR THEN VI1:=VI1_inv ELSE VI1:=VI1
        _ATAN_LINEAR_THEN_ELSE_5_&&NNN:
        iselgt      r4,r3,r4                                ;$ free r3

;#      LINEAR THEN NI1:=2 ELSE NI1:=NI1
        iselgt      r12,r5,r12                                ;$ free r5

;#      LINEAR ENDIF
        _ATAN_LINEAR_ENDIF_5_&&NNN:

```

Figure 25 : Exemple d'instructions assembleur pour le symbole ATAN

2.3.3. La création des tests

En parallèle du codage de la macro, un second développeur travaille sur la création de plans de tests. Ces plans de tests se font en quatre étapes majeures, la création de l'objet de test, la création du wrapper*, permettant la génération d'une planche composée uniquement du symbole testé, la création des tests eux-mêmes et la compilation et l'exécution.

La création de l'objet

La création de l'objet de tests se fait de la même manière que la création de l'objet de code. En utilisant une commande MARCEL, on ouvre la fenêtre de création. En revanche, cette fois-ci, on crée un objet de type T_TESTEUR_ASM que l'on appelle atan__1. L'objet créé contient deux fichiers vides, le premier est nommé atan__1_env.s et permet de concevoir le wrapper, le second, quant à lui, est appelé atan__1.ptu et permet de concevoir les tests eux-mêmes.

Le wrapper

Comme énoncé précédemment, le wrapper est un fichier utilisé afin de créer une planche composée de symboles. Le fichier est composé presque exclusivement de macros. La première étape consiste à déclarer les variables d'entrées et de sorties. Pour cela, il faut

positionner chacune d'entre elles dans la bonne zone mémoire dans la pile. Ainsi, une section de mémoire est préalablement créée afin de stocker un type précis de variable – réel, entier, booléen,... – et, à l'aide d'une macro nommée `m_BEGINNING_AREA_<INPUTS/OUTPUTS>_<TYPE>`, on peut directement accéder à cette section. Une fois dans la bonne section, on peut utiliser les macros `m_INPUT` et `m_OUTPUT` afin de respectivement déclarer des variables d'entrées et de sorties.

```
m_BEGINNING_AREA_INPUTS_M33
m_END_AREA_INPUTS_M33

m_BEGINNING_AREA_INPUTS_REAL
m_INPUT      E1,t_NUM
m_END_AREA_INPUTS_REAL

m_BEGINNING_AREA_INPUTS_VECTOR
m_END_AREA_INPUTS_VECTOR
```

Figure 26 : Déclaration de variables dans le wrapper du symbole ATAN

Une fois toutes les variables déclarées, on déclare un identifiant de planche, on démarre la section de code et on déclare un service nommé `SCAD_se_Nm_<id_planche>` grâce à la macro `m_BEGINNING_NODE` qui permet de créer un label. L'identifiant utilisé est un nombre à six chiffres écrits en base 36.

```
m_BEGINNING_AREA_IDENTIFICATION
m_ID_NODE_MODULE AB,CD,EF,C4
m_END_AREA_IDENTIFICATION

m_BEGINNING_AREA_CODE
m_BEGINNING_NODE AB,CD,EF
```

Figure 27: Identifiants dans le wrapper du symbole ATAN

Une fois ceci fait, on crée des variables, appelées connexions, dans la pile pour chaque entrée et sortie puis on donne aux registres `r1` et `r15` les valeurs des adresses dans la pile du début de section des variables et des rémanents. Ces registres sont ensuite utilisés dans le code assembleur pour charger et stocker ces différentes variables. On fait correspondre les différentes valeurs d'entrées dans la pile à celles des entrées passées à la planche.

```
m_CONNECTION      loc_E1,t_NUM
m_CONNECTION      loc_S1,t_NUM

m_NODE_ACTIVATION AB,CD,EF

;; Lecture des entrees
m_VvP             E1,loc_E1,t_NUM
```

Figure 28 : Initialisation des entrées dans le wrapper du symbole ATAN

Maintenant que les entrées sont prêtes, on peut lancer le symbole. Pour cela, on lance un compteur de temps puis on appelle la macro du symbole avec chacun de ses paramètres d'entrées et de sorties. Une fois la macro terminée, on arrête le compteur de temps afin de calculer le temps d'exécution de la macro. On affecte donc les valeurs des variables de sorties de la macro aux variables de sortie de planche puis afin de tester que les entrées n'ont pas été modifiées, on réalise la même opération avec ces dernières. Pour finir, on teste, avec la macro `m_END_NODE` que l'on n'ait pas dépassé la zone mémoire que l'on avait allouée puis on ferme la section de code.

```
;; Appel du symbole
ACTIV_COMPTEUR
m_ATAN      1,loc_E1,loc_S1
DEACTIV_COMPTEUR

;; Ecriture des sorties
m_PvV      loc_S1,S1,t_NUM

;; Reaffectation des entrees pour verifier qu'ils n'ont pas change
m_PvV      loc_E1,E1,t_NUM

m_END_NODE
m_END_AREA_CODE
```

Figure 29 : Appel de la macro et test dans le wrapper du symbole ATAN

Plan de tests

Les fichiers au format .ptu permettent de définir des plans de tests avant de pouvoir vérifier la fonctionnalité du code créé. Ils sont divisés en trois ou quatre parties selon le symbole et les exigences d'Airbus.

La première partie est la déclaration des variables et des fonctions de test. Il s'agit des variables d'entrées et de sorties du symbole ainsi que des fonctions `initTest()`, permettant d'initialiser la mémoire RAM, `finTest()`, actuellement vide mais tout de même obligatoire, et `SCAD_se_Nm_<id_planche>()`, décrite dans le wrapper. La déclaration des variables est précédée d'un pragma permettant de forcer le stockage de la variable dans la zone mémoire correspondant à son type. De plus, chacune des lignes de ces déclarations débute par un caractère '#' afin de spécifier à la compilation que ces lignes sont en langage C.

```

-----
-- declaration des variables globales (parametres de la macro testee)
-----
#pragma section SCAD_E_S_NUM ".scade_e_s_num" ".scade_e_s_num" far-absolute RW
#pragma use_section SCAD_E_S_NUM SCAD_re_rE1, SCAD_re_rS1

#SLIB_t_NUM SCAD_re_rE1;
#extern SLIB_t_NUM SCAD_re_rS1;

-----
-- Fonction generique du test
-----
#extern void InitTest();
#extern void FinTest();
#extern void SCAD_se_Nm_ABCDEF();

```

Figure 30: Déclaration dans le fichier ptu du symbole ATAN

La seconde partie du fichier de tests est exclusivement composée de commentaires. Ceux-ci sont présents afin d'expliquer les tests à venir. Les deux premières zones de commentaires permettent d'explicitier les variables d'entrées et de sorties.

La zone suivante est composée des classes d'équivalence de chaque variable d'entrées. Ces classes permettent de déterminer les intervalles à tester pour pouvoir vérifier tous les cas possibles et ainsi repérer toutes les erreurs que le codeur aurait pu faire. Il existe deux types de classes, les classes intrinsèques et les classes extrinsèques.

Les classes extrinsèques d'une variable sont uniquement déterminées par son type. Par exemple, les classes intrinsèques d'un réel sont $[-MAX_REEL; 0.0[$, $\{0.0\}$ et $]0.0; +MAX_REEL]$ avec MAX_REEL la borne maximale qu'un réel puisse atteindre. De la même manière un booléen a deux classes d'équivalence $\{TRUE\}$ et $\{FALSE\}$.

En revanche les classes extrinsèques sont à la fois déterminées par le type de la variable et par l'algorithme. Ainsi, à l'origine, les classes d'équivalence d'une variable sont identiques à ses classes intrinsèques, cependant, à chaque comparaison effectuée sur la variable et celles créées à partir d'elle, les classes se divisent pour en créer de nouvelles.

En prenant pour exemple l'algorithme du symbole atan tel qu'il est exprimé dans le document SDD, l'interface indique que la variable E1 est un flottant. Ainsi, ses classes intrinsèques sont $[-MAX_REEL; 0.0[$, $\{0.0\}$ et $]0.0; +MAX_REEL]$. La première opération effectuée sur E1 est une affectation avec une valeur absolue, ainsi, les classes d'équivalence créées dans les positifs le seront également dans les négatifs. Ensuite, la nouvelle variable est utilisée pour une condition : $IF(VI1>1.0)$ dans ce cas la classe $]0.0; MAX]$ est divisée en $]0; 1.0]$ et $]1.0; MAX]$. De même, à cause de la valeur absolue, $[-MAX; 0.0[$ est divisée en $[-MAX; -1.0[$ et $[-1.0; 0.0[$. Cette division se poursuit jusqu'à la fin de l'algorithme.

Ainsi, on peut créer les classes d'équivalence ci-contre :

```
--*****
--*                               CLASSES D EQUIVALENCE                               *--
--*****
-- Classes intrinseques:
-- E1 : { . < 0.0 } / { 0.0 } / { . > 0.0 }
--
-- Classes extrinseques:
-- E1 : [ 0.0, 0.26794919 ]
-- E1 : [ 0.26794919, 1.0 ]
-- E1 : [ 1.0, 3.73205080 ] ( 1/E1 in [ 0.26794919, 1.0 ] )
-- E1 : [ 3.73205080, inf[ ( 1/E1 in [ 0.0, 0.26794919 ] )
-- Et les memes pour E1 negatif
```

Figure 31 : Classes d'équivalence pour le symbole ATAN

La règle de choix de valeurs pour les tests est faite à partir de ces classes extrinsèques. Il faut choisir les bornes de chaque classe ainsi qu'une valeur intermédiaire. Sachant que plusieurs valeurs peuvent être testées dans un même plan de test, on peut alors les regrouper dans un unique test que l'on appelle Test fonctionnel. Ainsi, pour le symbole ATAN, deux tests fonctionnels sont créés, un pour les valeurs d'entrées négatives et un pour les valeurs positives ou nulles.

La troisième partie du fichier est l'écriture des tests fonctionnels. Ces tests sont divisés en Éléments qui permettent de faire varier plusieurs variables de test indépendamment les unes des autres dans un même test. Chaque élément est composé des différentes variables de la planche ainsi qu'un tableau nommé EABI et une autre variable appelée Piege_a_IT. Le tableau EABI est utilisé afin de vérifier que les macros de la planche respectent la norme EABI – pour Embedded Application Binary Interface – du processeur. Il s'agit d'une norme d'usage utilisée par de multiples processeurs tels que les ARM ou les PowerPC et portant sur les formats de fichiers, les types des données ou encore les utilisations de registres. La variable Piege_a_IT, quant à elle, permet de vérifier que la macro appelée ne déclenche pas d'interruption, par exemple à cause d'une division par 0 ou d'un débordement de la valeur des réels. Pour chaque variable, on donne une ou plusieurs variables d'initialisation et une ou plusieurs variables attendues à la fin de la macro. Pour les variables de sortie, on peut également donner une marge d'erreur sur les valeurs attendues.

Une fois toutes les valeurs déclarées pour chaque variable, on appelle le service créé dans le fichier atan__1_env.s

```

TEST TEST_FONC_1
FAMILLE fonctionnelle

ELEMENT
VAR SCAD_re_rE1, init dans { 0.0, 0.20, 0.5, 1.0, 3.0, 3.8, 50.0 }, VA=init
VAR SCAD_re_rS1, init = 99.9, VA(SCAD_re_rE1) dans { 0.0, 11.3099325, 26.5650512, 45.0, 71.5650511, 75.2564371, 88.8542371 }, DELTA=1.0E-4

-- Controle de la coherence Norme EABI :
TAB EABI, INIT = {0xFFFFFFFF,0xFFFFFFFF}, VA = {0,0}
-- Controle des interruptions :
VAR Piege_a_IT, INIT = IT_NO_IT, VA = INIT

#TESTER(SCAD_se_Nm_ABCDEF());
FIN ELEMENT

FIN TEST TEST_FONC_1

```

Figure 32 : Exemple de test pour le symbole ATAN

La dernière partie dépend des exigences d'Airbus. Dans le document SLS, une exigence est ajoutée sur certains symboles pour qu'un test dit 'nominal' soit ajouté. Il a le même format que les tests fonctionnels cependant, pour ce test, le document donne une liste de valeurs pour chaque entrée ainsi que les valeurs attendues en sortie.

La compilation et l'exécution

Une fois les tests et le code terminés, tous les fichiers peuvent être compilés afin de vérifier qu'il n'y a pas d'erreur dans aucun de ces fichiers et exécutés afin de vérifier que les résultats attendus dans les tests sont bien les mêmes que ceux obtenus à la sortie de la planche. L'avancement dans la compilation et l'exécution se font à l'aide de niveaux. L'objet de test est à l'origine au niveau 101 et on considère qu'il est correctement exécuté quand il atteint le niveau 1. Plusieurs commandes MARCEL sont utilisées pour compiler et exécuter les fichiers et ainsi faire descendre le niveau de l'objet.

- L'opération de niveau 100 correspond à la fabrication. Elle permet de compiler les antécédents de l'objet. Ces antécédents sont des objets IMPACT liés à l'objet de test afin que ce dernier puisse se servir des ressources décrites dans leurs fichiers. Un antécédent obligatoire est l'objet de code assembleur, pour que le wrapper puisse appeler la macro du fichier .mac. Elle fait passer ce dernier du niveau 101 au niveau 66.
- L'opération de niveau 65 est la compilation et le mappage. Elle permet ainsi de vérifier qu'il n'y a pas d'erreur dans les différents fichiers. Elle génère également les fichiers exécutables utilisés pour l'exécution. Elle fait passer le niveau de l'objet de 66 à 61.
- L'opération de niveau 61 est la compilation instrumentée. Elle génère des fichiers exécutables différents que ceux de l'opération précédente car ces derniers permettent de tester la bonne couverture structurelle des tests. Ainsi, on peut vérifier qu'après exécution des tests, toutes les branches du code ont bien été testées. Elle fait passer l'objet du niveau 61 au niveau 56. Aujourd'hui, cette étape est passée automatiquement sans génération de fichier.

- L'opération de niveau 56 est l'exécution. Elle utilise les fichiers générés par la compilation pour vérifier que les tests décrits dans les fichiers ptu sont correctement passés. Elle génère un fichier au format .ro qui résume cette exécution en donnant, pour chaque test, les valeurs de sortie attendues et les valeurs actuellement calculées. Elle fait passer l'objet du niveau 56 au niveau 51.
- La dernière opération est l'exécution instrumentée. Elle permet de tester la bonne couverture structurelle des tests. Comme les fichiers ne sont pas générés lors de la compilation instrumentée, cette opération échoue automatiquement. Elle fait passer l'objet du niveau 51 au niveau 1.
-

Une dernière opération est à effectuer après cela, il s'agit du passage du niveau de documentation 40. Il permet d'analyser la présence ou non d'avertissements lors de la compilation et de la compilation instrumentée. Si des avertissements sont détectés, alors cette opération échoue.

2.3.4. La relecture du code

La relecture du code assembleur permet au relecteur de déterminer si le code créé précédemment respecte bien toutes les règles énoncées dans le document SCS_ASM et est suffisamment clair. Il relit ainsi chaque instruction et chaque commentaire et les compare avec le document SDD. Il vérifie ainsi qu'aucune instruction n'a été retirée ou ajoutée par rapport à l'algorithme du document, que chaque commentaire d'algorithme y fasse bien référence, ou encore que les variables soient du bon type et que les indentations soient correctes, c'est-à-dire ici qu'une indentation corresponde à quatre espaces. Si une erreur est remarquée alors le correcteur l'écrit dans le document PRF associé au symbole.

VPZ	ed	Line 32	ATAN	Other	Delete the additionnal space before the first word "In" to align it on the rest of the comments
VPZ	ed	Line 225	ATAN	Other	Delete the additionnal space before the instruction parameters to align them on the rest of the code
VPZ	min	Line 114	ATAN	ASM_VERIF_60	"LINEAR_THEN_ELSE" label missing

Figure 33 : Remarque fait sur le code su symbole ATAN

Les remarques sont alors prises en compte par le codeur et les modifications insérées dans ce même document PRF. Les modifications sont ensuite vérifiées et le cas échéant validées par le correcteur.

Delete the additionnal space before the first word "In" to align it on the rest of the comments	OK	DONE	Corrected	OK	1.10
Delete the additionnal space before the instruction parameters to align them on the rest of the code	OK	DONE	Corrected	OK	1.10
"LINEAR_THEN_ELSE" label missing	OK	DONE	Corrected	OK	1.10

Figure 34 : Prise en compte des remarques sur le symbole ATAN

Une fois la relecture terminée, le relecteur peut exécuter l'opération de documentation de niveau 21. Cette opération permet de signifier la fin de la relecture en écrivant OK dans un fichier.

3. La conception de SYST

3.1. Les fichiers d'entrée

Les fichiers d'entrées de la partie SYST ont la même hiérarchie que ceux de la partie SLIB. Cependant, leur contenu peut légèrement ou entièrement varier. Également, certains documents ne sont ici pas utilisés dans le cadre de la conception de cette partie. Voici la liste des fichiers utilisés :

Le mémo de coordination

Tout comme pour la conception de SLIB, le travail sur SYST commence avec la réception d'un mémo de coordination. Ce document est très similaire à celui décrit précédemment, les différences étant les documents mentionnés, certains étant spécifiques au travail sur SYST et les listes de tâches.

Le document d'exigences logicielles

Tout comme le document SRD de SLIB, celui-ci contient une liste d'exigences à appliquer dans la conception du SDD de SYST. Ces exigences peuvent être couvertes par plusieurs documents SDD en même temps. Ces exigences peuvent correspondre à des connexions entre différentes parties du SDD ou à des règles d'implémentations des services. Chacune des exigences doit être justifiée par des traçabilités entre le document SRD et le document SGS de plus haut niveau.

Le document de conception logicielle

Le design de SYST est très différent du fait de l'utilisation des nouvelles méthodes de travail et de Flyworks. Ici, il est composé de deux fichiers, un fichier codda et un fichier dcsl.

Le fichier codda

Il permet de créer des machines possédant des types, des constantes, des ressources et des services. Ces machines fonctionnent comme des classes de langage objet car les machines peuvent être terminales ou non terminales et on peut donner une machine parente à une autre.

Chaque addition à ce fichier doit posséder une description et être justifiée. Cette justification doit être faite en utilisant la traçabilité des exigences du document SRD.

Le fichier dcsI

Il permet de formaliser les services déclarés dans les fichiers codda. Cette formalisation se fait à l'aide de prédicats et d'un contrat. Ce contrat est composé d'un ou plusieurs comportements. Un comportement peut atteindre l'ensemble du service ou seulement des cas particuliers. Dans ce premier cas, le comportement sera considéré global. Dans le second cas, il faut ajouter au contrat des comportements nominaux.

Les comportements nécessitent que certaines conditions, représentées par des prédicats, soient respectées en entrée et en sortie du service. Ainsi, si les prédicats d'entrées d'un comportement sont respectés, les comportements nominaux peuvent être activés et les prédicats de sortie doivent être respectés également. Dans le fichier, toutes ses conditions sont indiquées à l'aide de balises sous la forme :

```
contract {  
  behavior __nominal__ <nom comportement> {  
    assumes {  
      pre {  
        <prédicats d'entrées>  
      }  
    }  
    ensures {  
      outputs {  
        <nom des paramètres de sortie>  
      }  
      post {  
        <prédicats de sortie>  
      }  
      flow {  
        <liste des appels de services>  
      }  
    }  
  }  
}
```

Ces nouveaux fichiers couplés à l'outil Flyworks permettent la génération automatique des patrons de la plupart des livrables. Cet outil étant en cours de développement, il est cependant encore instable et génère de nombreuses erreurs dans les fichiers. À terme, il permettra d'améliorer considérablement la vitesse des processus de développement même si la relecture devra toutefois vérifier que ce qui est créé est bien en adéquation avec le design.

Le document de règles de codage

Ce document donne la liste des règles de codage à respecter lorsque l'on code en langage C. Tout comme pour le SCS-ASM, ces règles sont classées en trois catégories : les règles obligatoires, les règles recommandées et les règles non applicables, les premières étant les seules vérifiées lors des contrôles qualités.

3.2. Les fichiers de sortie

En ce qui concerne les fichiers de sortie, un seul document est identique à ceux utilisés pour la partie SLIB. Cependant, les objectifs de chacun d'entre eux restent les mêmes. En effet, le formulaire de relecture est ici strictement identique mais à la place des fichiers de test et de code assembleur, il y a un nouveau fichier de test dont la forme est différente et un fichier de code en langage C.

Le fichier de code

Le fichier de code C permet de regrouper les différents codes des services décrits dans les fichiers codda. Ce code doit suivre les règles décrites dans le document SCS-C. Ces fichiers, à l'exception des corps des services, sont générés automatiquement avec l'outil gradle de Flyworks. Il est accompagné d'un header qui contient les déclarations des variables globales et les prototypes des services à implémenter.

Le fichier de test

Ces fichiers permettent de créer les plans de tests unitaires correspondant à un service. Ces plans sont générés automatiquement avec l'outil gradle de Flyworks à raison d'un test par comportement décrit dans le fichier dcs1 du service correspondant. Ces tests devront respecter les règles fixées dans un fichier SUTS mais ce fichier n'a pas encore été créé au moment de l'écriture de ce rapport.

3.3. Le travail effectué

De la même manière que pour le travail effectué sur la partie SLIB, le travail sur SYST s'effectue en quatre étapes :

- La relecture des fichiers SDD
- Le codage des services
- L'élaboration des tests
- La relecture du code

Cette dernière est identique à celle décrite dans la description de la partie SLIB et n'a pas encore été réalisée lors de l'écriture de ce rapport. Ces étapes sont faites pour chaque machine décrite dans les documents SDD et une machine nécessite au moins deux développeurs pour respecter l'indépendance codeur-testeur. Par conséquent, pour l'écriture de ce rapport, je vais utiliser un exemple avec la machine scheduler_lib, qui permet d'implémenter les différents modes des commandes de vol, et plus particulièrement sur le service YCLL_se_SelectCyclicMode.

La relecture SDD

Le travail sur SYST commence avec la réception d'un COMO indiquant un numéro de révision SVN indiquant la version du projet avec les dernières modifications des fichiers SDD ainsi que le numéro de version du SRD. La première étape consiste donc à récupérer cette révision. Ensuite, il faut relire les fichiers codda et dcsI pour vérifier qu'il n'y a pas d'erreur dans leur syntaxe et qu'ils respectent les exigences SRD. La relecture se fait en deux étapes, dans un premier temps les fichiers codda et dcsI sont lus et les justifications sont vérifiées en fonction de traçabilités écrites dans les fichiers.

```
/** @service(YCLL_se_SelectCyclicMode)
    This service requests the selection of the next mode of the mode : INIT, OPER, PFBIT, IBIT or NOSYNC, which will be activated during the next major cycle.

    @return void

    @traceability
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17700
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17710
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17720
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17730
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17740
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17750
    #Link_to E_OH1_SRDSYST_SelectCyclicMode_17780
*/
```

Figure 35 : Déclaration du service dans le fichier codda

La deuxième étape, quant à elle consiste à regarder chaque exigence du SRD et à vérifier qu'elles sont toutes entièrement appliquées dans les documents SDD. Si une erreur est repérée, elle est annotée dans le fichier PRF correspondant à la relecture SDD.

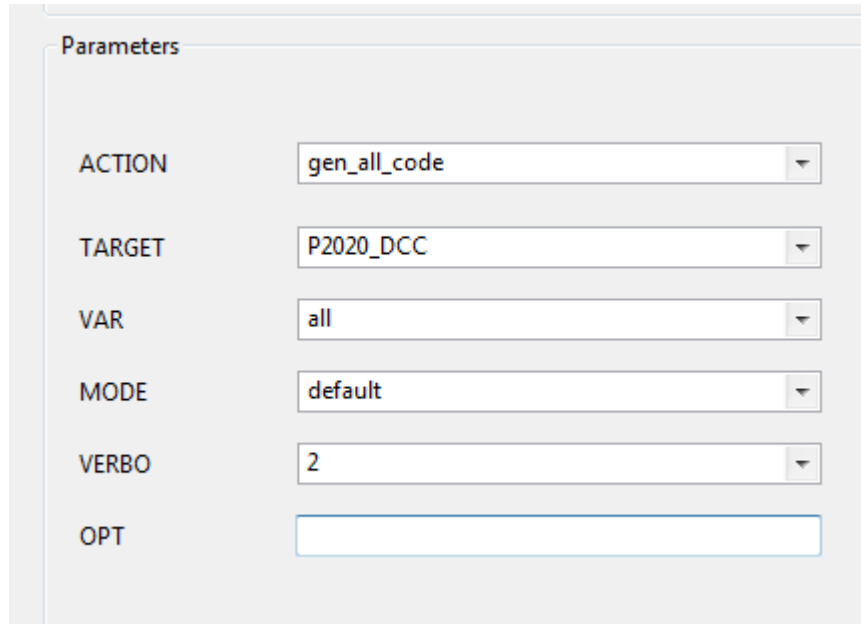
VPZ	Maj	YCLL_se_SelectCyclicMode.dcsI	Line 8	DE001	The behaviour of this service does not match the requirement E_OH1_SRDSYST_SelectCyclicMode_17750. The mode OPER shall be set if the MareaToBSP_SELECTED_MODE variable is not set at INIT, NOSYNC, OPER, PFBIT or IBIT. Or the default mode implemented in the dcsI file does not contain the "not(OPER)" possibility.
-----	-----	-------------------------------	--------	-------	--

Figure 36 : Remarque PRF du fichier codda et dcsI

Une fois toutes les étapes de relecture effectuées, le document PRF est envoyé au concepteur du SDD, qui doit corriger les erreurs trouvées, et envoyer sur SVN la nouvelle révision. Nous vérifions les corrections faites et validons le document SDD.

3.3.1. Le codage

Avant de commencer le codage des fichiers C, il faut dans un premier temps générer les squelettes de ces derniers. Pour cela, il faut utiliser une commande gradle appelée `fwk_optimases_generate` associée à l'action `gen_all_code`. Cela permet de convertir toutes les données inscrites dans les fichiers `codda` en des fichiers `headers` et de code.



Parameters	
ACTION	gen_all_code
TARGET	P2020_DCC
VAR	all
MODE	default
VERBO	2
OPT	

Figure 37 : Fenêtre pour les commandes gradle

Après cela, les fichiers sont répartis entre les développeurs. Dans un fichier `.c` se trouvent tous les services implémentés dans les fichiers `codda` et le code des services à écrire doit respecter les exigences des fichiers `dcs1` correspondant à ses services, la seule exception étant les balises *requires* des comportements globaux, ces dernières étant vérifiées séparément. Chaque comportement nominal est alors permis grâce à des conditions déterminées par des structures – `if` ou `switch` selon le nombre de comportements. Ces conditions sont définies grâce aux prédicats d'entrée définis dans la balise *requires* des comportements. Dans ces structures sont ensuite appliqués les exigences définies dans les balises `flow` et `post` de ces mêmes comportements. Ainsi, le service `YCLL_se_SelectCyclicMode` possède plusieurs comportements nominaux, chacun étant spécifique à un mode donné.

```

let MODE_INIT      = (YRES_re_MareaToBsp.SELECTED_MODE  ≡ YRES_ct_INIT);
let MODE_NOSYNC    = (YRES_re_MareaToBsp.SELECTED_MODE  ≡ YRES_ct_NOSYNC) ;
let MODE_IBIT      = (YRES_re_MareaToBsp.SELECTED_MODE  ≡ YRES_ct_IBIT) ;
let MODE_PFBIT     = (YRES_re_MareaToBsp.SELECTED_MODE  ≡ YRES_ct_PFBIT) ;
let MODE_OPER      = (YRES_re_MareaToBsp.SELECTED_MODE  ≡ YRES_ct_OPER) ;
let MODE_DEFAULT   = (~MODE_INIT ∧ ~MODE_PFBIT ∧ ~MODE_IBIT ∧ ~MODE_NOSYNC ∧ ~MODE_OPER) ;

contract {
  global {
    requires {
      pre {
      }
    } //end of requires

    ensures {
      outputs {
        YCLL_re_ActiveMode
      }
      flow {
        observer ≡ ();
      }
    } //end of ensures
  } //end of global

  behavior __nominal__ select_mode_INIT {
    // selection of the next mode INIT
    assumes {
      pre { MODE_INIT ; }
    }
    ensures {
      post { YCLL_re_ActiveMode ≡ YCLL_ct_MODE_INIT; }
    }
  } //end of behavior
}

```

Figure 38 : Extrait du fichier d'un service dcs1

En conséquence, il faut créer une structure switch afin de tester le mode courant et ainsi appliquer le bon comportement.

```

void YCLL_se_SelectCyclicMode(void)
{
  switch (YRES_re_MareaToBsp.SELECTED_MODE)
  {
    /* Selection of the next mode INIT */
    case YRES_ct_INIT:
      YCLL_re_ActiveMode = YCLL_ct_MODE_INIT;
      break;

```

Figure 39 : Extrait du code C d'un service

Une fois le code terminé, il faut utiliser deux nouvelles commandes gradle, la première est `fwk_make` afin de vérifier que le code compile bien et la seconde est `fwk_optimases_generate` avec l'action `check_all_code` afin d'utiliser l'outil CheckC qui vérifie que le code respecte bien les règles définies dans le SCS-C.

3.3.2. Les fichiers de test

De la même manière que pour le codage, il faut dans un premier temps générer les fichiers contenant les squelettes des tests en utilisant la commande gradle `fwk_optimases_generate` avec la commande `generate_all_tests`. Cette fois-ci cependant, ces squelettes sont créés à partir des fichiers `dcs1`. En effet, les différents cas de test correspondent chacun à un comportement nominal décrit dans le fichier `dcs1` du service testé.

Les tests sont divisés en trois parties, la première permet de donner les valeurs des variables d'entrées du service. La deuxième permet de simuler services appelés par celui qui est actuellement testé ou de donner des valeurs aux différentes variables globales. La troisième est l'association du comportement décrit dans le fichier `dcs1` avec le test et l'appel du service du fichier `.c`. Lors de la conception des tests, il faut respecter les classes d'équivalence évoquées précédemment. Ainsi, pour le service `YCLL_se_SelectCyclicMode`, il faut vérifier chacune des valeurs que la variable `YRES_re_MareaToBsp.SELECTED_MODE` et ainsi vérifier que le mode donné à la variable `YCLL_re_ActiveMode` soit le bon en fonction de cette première. Ici, la première partie restera vide car le service ne possède pas de variable d'entrée. En revanche, il faut donner la valeur équivalente au comportement attendu à la variable globale `YRES_re_MareaToBsp.SELECTED_MODE`. Contrairement aux tests réalisés sur la partie `SLIB`, il n'est pas utile de donner aux tests les valeurs attendues en retour de service, celles-ci étant apportées par les comportements des fichiers `dcs1`.

À l'heure de l'écriture de ce stage, le document `SUTS` n'est pas encore terminé et la commande gradle utilisée pour vérifier les tests n'est pas encore stable. Par conséquent, les tests n'ont pas encore été créés.

Conclusion

L'objectif de ce stage était d'aider à la réalisation de composants de commande de vol hélicoptère. Ce développement est réalisé en deux parties, le développement de la partie système et celui d'une librairie de symboles appelés par le système. Ce développement a pu se faire en utilisant des logiciels fournis par Airbus, Flyworks pour la partie système et IMPACT et MARCEL pour la librairie, ainsi que deux langages de programmation, le langage assembleur pour le codage des symboles et le langage C pour les tests des symboles et toute la conception de la partie symbole. Ce stage s'inscrit dans un projet lancé par Airbus Helicopters dans le cadre de la conception d'un nouvel hélicoptère, le X6. Ce projet doit se terminer en 2020, par conséquent, à l'issue du stage, il reste encore beaucoup de travail à réaliser.

Plusieurs difficultés ont été rencontrées durant le stage. Premièrement d'un point de vue technique car j'avais peu d'expérience dans le langage assembleur avant le début de ce stage. Ensuite, d'un point de vue logiciel, puisque les outils utilisés sur la partie système sont en cours de développement, ils sont donc encore assez instables, ce qui a pu ralentir le développement. Enfin, d'un point de vue organisationnel, l'enchaînement des tâches nécessaires au bon déroulement du projet a posé quelques problèmes. En effet, la plupart des tâches confiées à SII dépendent de plusieurs paramètres dont l'avancement des outils et des documents d'entrée. Un retard dans la livraison de ces entrées pouvait retarder de beaucoup notre équipe et ainsi créer des trous de charge importants dans notre occupation sur le projet.

Ce stage m'a beaucoup appris sur le travail en équipe mais aussi sur le travail avec des contraintes, celui-ci devant suivre de nombreuses règles de codage dues à la norme qualité DO178C. Cela m'a également permis d'en apprendre plus sur le travail au sein des ESN et du domaine avionique plus généralement.

Le travail effectué est encourageant car chaque tâche demandée par Airbus EYY a été livrée dans les temps et je pense que SII et Airbus ont été satisfaits des résultats. Cela s'est traduit notamment par une proposition de contrat en CDI par mon manager SII. Je pourrai donc continuer mon travail au sein de l'équipe par la suite.

Lexique

COMO : **Mémo de Coordination**, document indiquant la liste des tâches à effectuer

Flyworks : Extension du logiciel Eclipse utilisée pour la conception de SYST permettant la gestion de versions de fichiers, leur modification et leur compilation

IMPACT : **Integrated Management Process And Configuration Technic**, Extension du logiciel Eclipse utilisée dans la conception de SLIB et permettant la gestion de versions de fichiers

MARCEL : **Mécanisme d'Aide à la Réalisation et au Contrôle d'État Logiciel**, Extension du logiciel Eclipse utilisée dans la conception de SLIB et permettant la modification des fichiers et leur compilation

PRF : **Proof-Reading Form**, fichier tableur réalisé pour tracer les erreurs ainsi que les corrections apportées dans les documents SDD et les fichiers de code

SCADE : **Safety Critical Application Development Environment**, environnement de développement permettant de modéliser des systèmes temps réel critiques sous forme de planches

SCS : **Software Coding Standard**, document regroupant les règles de codage dans un langage particulier

SDD : **Software Design Data**, document regroupant les exigences bas niveau d'un logiciel

SLIB : **SCADE Library**, librairie composée de symboles utilisés dans la conception des planches SCADE

SLS : **SCADE Library Specification**, document regroupant la liste des symboles SCADE et leurs fonctionnalités

SRD : **Software Requirements Document**, document donnant la liste des exigences haut niveau d'un logiciel

SYST : ensemble des fonctions système des commandes de vol

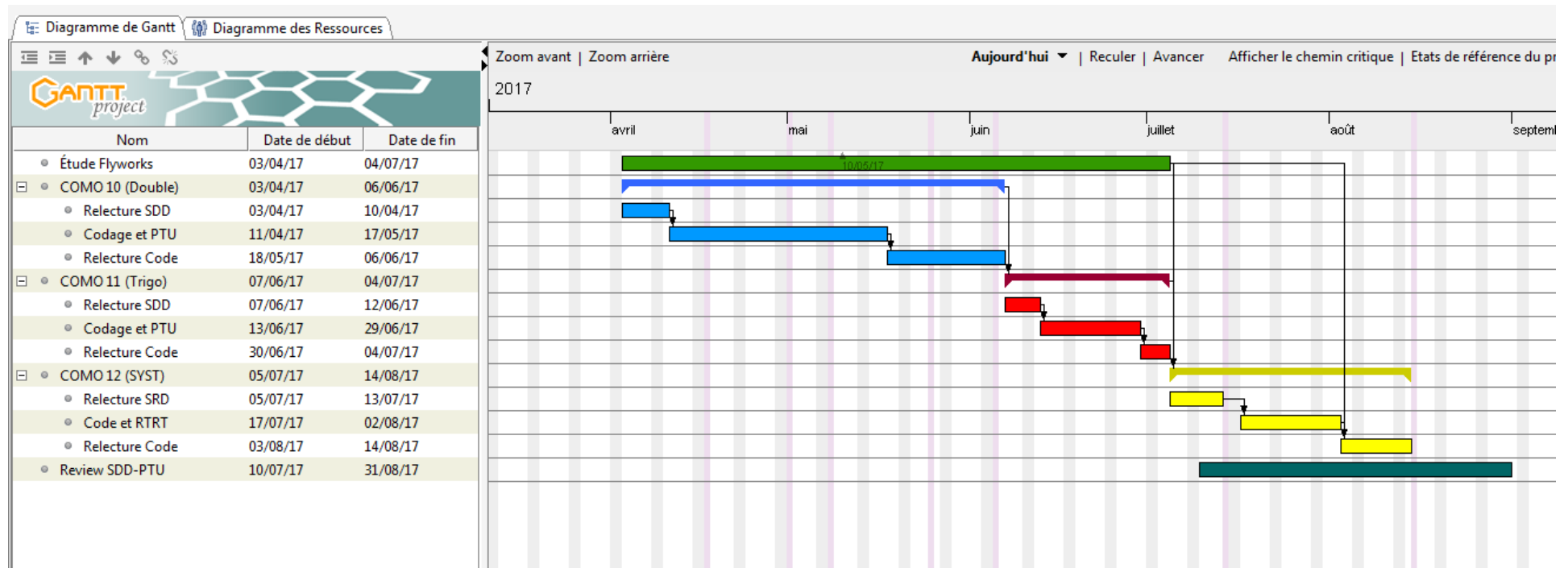
TOR : **Tool Operationnal Requirements**, document regroupant les exigences utilisées dans la génération automatique de code

Wrapper : fichier utilisé pour créer des planches SCADE

Bibliographie

- [1] <http://www.groupe-sii.com/fr>
(Consulté le 04 août 2017)
- [2] https://fr.wikipedia.org/wiki/Airbus_Helicopters_X6
(Consulté le 08 août 2017)
- [3] <http://www.helicopassion.com/fr/02/tech01.htm>
(Consulté le 08 août 2017)
- [4] [https://fr.wikipedia.org/wiki/Pipeline_\(architecture_des_processeurs\)](https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs))
(Consulté le 10 août 2017)
- [5] https://fr.wikipedia.org/wiki/Cycle_en_V
(Consulté le 10 août 2017)
- [6] LEDRU Chrystophe, « Présentation DO-178C / ED-12C » [Présentation Powerpoint]
(Faite le 28/06/2017)

Annexe 1 : Diagramme de Gantt Prévisionnel



Annexe 2 : Diagramme de Gantt Définitif

