Rendre agile les tests d'intégration des systèmes avioniques par des langages dédiés

Robin Bussenot

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *16/07/2018* **par :**

Robin Bussenot

Rendre Agile les tests d'intégration des systèmes avioniques par des langages dédiés

---

JURY

Ana Rosa Cavalli	Professeur, Institut Télécom Sud Paris	Rapporteur
Hervé Leblanc	Maître de conférence, Université Paul Sabatier	Co-directeur
Christian Percebois	Professeur, Université Paul Sabatier	Directeur
Laurent Réveillère	Professeur, Université de Bordeaux	Rapporteur
Sébastien Salva	Professeur, Université d'Auvergne	Examinateur
Hélène Waeselynck	Directrice de recherche CNRS, LAAS	Examinatrice
Jean-Pierre Gaubert	Ingénieur, Airbus	Invité

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse
Directeur(s) de Thèse :
Christian Percebois et *Hervé Leblanc*
Rapporteurs :
Ana Rosa Cavalli et *Laurent Réveillère*

Résumé

Dans l'ingénierie avionique, les tests d'intégration sont cruciaux : ils permettent de s'assurer du bon comportement d'un avion avant son premier vol, ils sont nécessaires au processus de certification et permettent des tests de non-régression à chaque nouvelle version d'un système, d'un logiciel ou d'un matériel. La conception d'un test d'intégration coûte cher car elle mêle la réalisation de la procédure, le paramétrage de nombreux outils couplés au banc de test ainsi que l'adressage des interfaces du système testé. Avec des procédures de test écrites en langage naturel, l'interprétation des instructions d'un test lors de son rejeu manuel peut provoquer des erreurs coûteuses à corriger, en raison notamment des actions précises à entreprendre lors de l'exécution d'une instruction de test. La formalisation et l'automatisation de ces procédures permettraient aux équipes de testeurs de se concentrer sur la réalisation de nouveaux tests exploratoires et sur la mise au point de tels systèmes au plus tôt. Or, un système avionique est composé de plus d'une centaine de systèmes embarqués, chacun concernant des compétences spécifiques.

Notre contribution est alors un *framework* orchestrant les langages de test dédiés à l'intégration de systèmes avioniques dans une vision Agile. Nous introduisons tout d'abord le concept de langage spécifique à un domaine (*Domain Specific Language* ou DSL) et montrons comment nous l'utilisons pour la formalisation des procédures de test dédiées à un type de système particulier. Ces langages devront pouvoir être utilisés par des testeurs avioniques qui n'ont pas forcément de compétences en informatique. Ils permettent l'automatisation des tests d'intégration, tout en conservant l'intention du test dans la description des procédures. Puis, nous proposons l'approche BDD (*Behavior Driven Development*) pour valider l'intégration de systèmes par scénarios comportementaux décrivant le comportement attendu de l'avion.

Nous nommons *Domain Specific Test Languages* (DSTL) les langages utilisés par les testeurs. A chaque système (ATA ou *Air Transport Association of America*) correspond un DSTL métier. Un premier DSTL concernant les systèmes de régulation de l'air a été développé entièrement en tant que preuve du concept à partir de procédures existantes pseudo-formalisées. L'expérimentation s'est poursuivie avec les calculateurs standardisés IMA (*Integrated Modular Avionic*) pour lesquels les procédures de test sont décrites en langage naturel et sont donc non automatisables. A partir d'un corpus de procédures, nous proposons un premier processus empirique d'identification des patrons de phrases peuplant

un DSTL. Le corpus fourni est composé de dix procédures totalisant 108 chapitres de test et 252 tests ou sous-tests comportant au total 3708 instructions pour 250 pages Word. Rendre Agile ces tests d'intégration consiste à proposer une approche collaborative pour formaliser un DSTL que ce soit pour les patrons de phrases de la grammaire concrète ou pour les patrons de transformations vers des langages exécutables.

Abstract

In avionics engineering, integration tests are crucials : they allow to ensure the right behavior of an airplane before his first flight, they are needed to the certification process and they allow non-regression testing for each new version of a system, of a software or of a hardware. The design of an integration test is expensive because it involves the implementation of the procedure, the configuration of tools of the bench and the setup of the interfaces of the system under test. With procedure written in natural language, the interpretation of statements of a test during the manual execution can lead to mistakes that are expensive to fix due to accurate actions needed to perform a statement. The formalization and the automation of those procedures allow testers team to focus on the implementation of new test cases.

First of all, we introduce Domain Specific Language (DSL) and show how we use it to formalize tests procedures dedicated to a kind of system. Those languages should be able to be use by avionic testers which do not necessarily have programming skills. They allow test automation, while maintaining test intention in the test description. Then, we proposed a BDD (Behavior Driven Development) approach to validate the integration of systems thanks to behavioral scenarios describing the expected behavior of the airplane. Our contribution is a framework which orchestrate DSLs dedicated to integration test of avionic systems in an Agile vision.

We named Domain Specific Test Languages (DSTL), languages used by expert testers. For each system (ATA ou Air Transport Association of America) corresponds a DSTL business. A first DSTL about the validation of airflow control systems has been developed as a proof of concept from existing procedures pseudo-formalized. The experimentation has been continued with IMA (Integrated Modular Avionic) calculators for which test procedures are written in natural language and thus are not automatable. From a corpus of procedures, we propose a first empirical process to identify sentence patterns composing the DSTL. The corpus provided is composed by ten procedures totaling 108 test chapters and 252 tests or subtests involving 3708 statements for a total of 250 Word pages. Make Agile integration tests in this context consist to propose a collaborative approach to formalize a DSTL and to integrate it in the orchestration framework to generate automatically the glu code.

Remerciements

En premier lieu, je tiens à adresser mes remerciements à Monsieur Percebois et Monsieur Leblanc, respectivement directeur et encadrant de cette thèse. Ils ont su me soutenir, m'aider et me guider tout au long de ce travail. J'associe à ces remerciements toutes les personnes du jury qui ont permis la validation de cette thèse.

Je tiens également à remercier Jean-Pierre Gaubert, responsable d'une équipe d'intégration Airbus, de m'avoir fait confiance et de m'avoir donné l'opportunité d'expérimenter mes travaux dans un contexte industriel. De même, j'aimerais remercier tous les acteurs du projet ACOVAS qui m'ont aiguillés grâce à leurs expériences et qui m'ont permis d'appréhender l'industrie avionique et ses besoins.

J'aimerais remercier toutes les personnes présentes à mes côtés tout au long de mes études. J'adresse toute ma reconnaissance à Tariq qui a su être là quand j'en avais besoin, à Xavier qui m'a aidé à garder le cap pour atteindre mes objectifs et à Vanessa, sa compagne, pour sa bonne humeur et les relectures effectuées lors de la rédaction de ce manuscrit.

Mes remerciements vont aussi à ma famille et principalement à mes parents qui ont été présents dans les bons comme dans les mauvais moments et qui m'ont supporté à chaque instant. Je les remercie de leur patience et de m'avoir donné la chance d'accomplir toutes ces années d'études.

Mes derniers remerciements vont à ma compagne, Charlène, qui a toujours été présente pour moi et qui m'encourage et me motive dans tout ce que j'entreprends. La patience et le soutien dont elle a fait preuve durant cette thèse ont été infaillibles et m'ont donné la force pour achever ce travail.

Table des matières

Introduction

Les systèmes composant un aéronef sont critiques, embarqués, réactifs et temps-réel. Le découpage de l'aéronef en systèmes, sous-systèmes, puis en équipements est établi lors de la phase de spécification du système global. Une fois les fonctions de l'avion identifiées, elles sont regroupées en plusieurs domaines. Ces domaines correspondent à la classification ATA (*Air Transport Association of America*) qui propose un découpage des systèmes d'un avion en 100 chapitres différents. A titre d'exemple, le système de l'aéronef inclut les domaines suivants : air conditionné et pressurisation, vol automatique, communications, génération électrique, commandes de vol, avionique modulaire intégré ... La criticité des systèmes avioniques implique qu'ils doivent être préalablement certifiés pour qu'un avion soit commercialisable. La vérification et la validation des systèmes composant un domaine, puis de l'intégration correcte de ces systèmes, sont prépondérantes pour garantir la sûreté de fonctionnement d'un avion en toutes circonstances.

Le test est l'activité essentielle du processus de vérification et de validation lors de l'étape d'assemblage des sous-systèmes d'un avion. Tout au long du cycle de développement, chacun des systèmes est testé en isolation pour vérifier son adéquation aux spécifications. Ces systèmes sont ensuite assemblés par domaine dans une phase dite d'intégration. Nous nous sommes intéressés aux tests réalisés durant cette phase. Ces tests sont cruciaux car ils permettent les dernières vérifications avant le premier vol. Les erreurs qui ne sont pas détectées à ce moment peuvent alors être extrêmement coûteuses car elles sont susceptibles de causer des pertes humaines, matérielles et des dommages environnementaux.

Les tests d'intégration sont exécutés à travers un banc de test. Un banc interagit avec le système testé en s'appuyant sur ses interfaces de communication. Il lie le système à une simulation temps-réel qui exécute artificiellement les conditions environnementales et les comportements des systèmes qui ne sont pas encore intégrés. Un test d'intégration nécessite de configurer les interfaces du banc pour établir la connexion avec le système testé et de coordonner la simulation temps-réel et les actions à réaliser pour obtenir le comportement désiré. La plupart du temps, ces comportements sont validés *a posteriori* par un expert à partir de données d'exécution sauvegardées par le banc.

L'exécution manuelle de ces tests est fastidieuse car elle demande de réaliser des actions minutieuses ou dans un bref délai, comme c'est le cas pour la collecte des résultats au cours de l'exécution. De plus, une grande partie des procédures de test d'intégration sont rédigées

en langage naturel, moyen d'expression porteur d'ambiguïté où l'interprétation de chacun entre en jeu. Les erreurs pouvant intervenir lors de l'interprétation d'une instruction, lors de la manipulation du banc ou du système et lors de la collecte de résultats peuvent rendre caduque l'exécution d'un test, voire d'une campagne de tests. Le coût d'exécution élevé de ces tests implique qu'ils sont rejoués un minimum de fois. De plus, la conception d'une procédure de test nécessite un savoir-faire d'expert et du temps pour rendre explicite les procédures en langage naturel. La formalisation et l'automatisation de ces procédures permettraient aux équipes de testeurs de concevoir de manière plus rapide et plus sûre des procédures de test, de se concentrer sur la réalisation de nouveaux tests exploratoires et sur la mise au point de tels systèmes au plus tôt.

La formalisation des procédures a pour but de lever toute ambiguïté. C'est la première étape pour les automatiser. Les experts concevant ces procédures connaissent les systèmes testés et les bancs de tests utilisés, mais n'ont cependant pas de compétences en programmation. Peu de langages de programmation sont dédiés aux tests des systèmes embarqués et ils ne permettent pas aux experts de participer activement à la conception des procédures de test automatisables. Dans un premier temps, nous avons proposé d'utiliser les DSL (*Domain Specific Languages*) pour formaliser les procédures de test spécifiques à chaque ATA. Puisque ces langages sont proches du niveau du langage naturel utilisé des testeurs experts et proches des exigences, nous avons nommé ces langages DSTL (*Domain Specific Test Languages*). Ce concept sera présenté au chapitre 2 de la thèse. L'élaboration d'un DSTL à partir de la donnée de dizaines de procédures de test en langage naturel fera l'objet du chapitre 4.

Comme écrit précédemment, les tests d'intégration permettent de s'assurer du bon comportement d'un avion avant son premier vol. Ils sont nécessaires au processus de certification et permettent des tests de non-régression à chaque nouvelle version d'un système, d'un logiciel ou d'un matériel. L'automatisation des tests d'intégration représente alors un enjeu fondamental, que ce soit dans leur exécution et dans leur verdict. L'automatisation de l'exécution des tests réduit considérablement les coûts d'intégration des systèmes. Elle a aussi pour conséquence de faciliter le rejeu des tests pour augmenter la confiance lors de la mise en production d'un ensemble de systèmes complexes. Les experts en charge de ces tests disposeront ainsi de plus de temps pour rejouer un *bug* et en découvrir les causes. Nos DSTL sont alors orchestrés dans un *framework* en charge de leur automatisation. Ce *framework* offre trois niveaux de langage, du plus ubiquitaire aux langages de script spécifiques à un banc de test. La chaîne de transformations de programmes est accélérée et simplifiée grâce à la donnée d'un langage dit pivot accessible aux programmeurs testeurs. Le *framework* est présenté au chapitre 2. Le langage pivot combiné à des patrons de transformation vers des automates temporisés facilite la production de scripts exécutables comme démontré au chapitre 3.

Le premier chapitre de cette thèse décrit les processus de développement nécessaires à la conception d'un avion et la place prépondérante des tests dans ces processus. Différentes approches concernant la formalisation et l'automatisation des tests sont présentées

et discutées. Nous avons choisi l'ingénierie des langages pour formaliser et orchestrer les différents langages de test dédiés aux ATA. Le deuxième chapitre détaille cette contribution en présentant, tout d'abord, le concept de DSTL, puis leur orchestration. Le troisième chapitre montre la preuve du concept sur le développement complet d'un premier langage de haut niveau dédié au système de régulation de l'air en cabine, jusqu'à la génération de code exécutable pour plusieurs types de banc. Le dernier chapitre expérimente et ébauche un processus de développement dédié à la conception de DSTL à partir d'un corpus de procédures existantes en langage naturel pour un nouvel ATA.

Ces travaux ont été conduits dans le cadre du projet FUI ACOVAS (outil Agile pour la COnception et la VAlidation Système). Ce projet a pour objectif de concevoir une plateforme de validation et d'intégration système inspirée des méthodes agiles. Cette agilisation concerne aussi bien les fonctionnalités d'un banc que les processus de développement dédiés aux procédures de test. Rendre Agile les tests d'intégration des systèmes avioniques par des langages dédiés constitue notre contribution à ce projet. Le BDD (*Behavior Driven Development*) propose que les experts d'un domaine puissent décrire leurs besoins dans un langage ubiquitaire que les programmeurs testeurs transforment en tests automatisés. Les tests d'acceptation automatisés deviennent alors le vecteur de communication pour les parties prenantes. Nous avons transposé cette approche issue de l'ingénierie logicielle à l'ingénierie système pour la conception des tests d'intégration de systèmes avioniques.

Chapitre 1

Tests pour l'ingénierie des systèmes avioniques

Nous abordons dans ce chapitre la manière dont le développement et les tests entrent en jeu dans le processus de conception d'un avion. Un avion est composé d'un certain nombre de systèmes hétérogènes produits par des fournisseurs différents et intégrés par un avionneur. Ces systèmes sont embarqués, temps-réel, réactifs et critiques. Ces caractéristiques demandent un effort et une rigueur accrus sur les tests pour des besoins de vérification et de validation, mais aussi de certification car une seule défaillance peut entraîner des pertes humaines. Ces systèmes doivent respecter des normes et des standards qui sont définis par des organisations telles que EUROCAE, (*EURopean Organisation for Civil Aviation Equipment* [33]) ou encore RTCA (*Radio Technical Commission for Aeronautics* [64]).

Le test est une activité prépondérante du processus de vérification et de validation lors de la réalisation d'un avion. Pour autant, un effort d'automatisation des tests reste à faire, en particulier sur les tests d'intégration. Leur automatisation permettrait de réduire les coûts d'intégration des systèmes. L'automatisation a aussi pour conséquence de faciliter le rejeu des tests pour augmenter la confiance lors de la mise en production d'un ensemble de systèmes complexes.

Avant d'entrer dans le vif du sujet, nous présentons les spécificités du développement avionique, en particulier les évolutions des architectures matérielles et logicielles permettant aux différents systèmes composant un avion de communiquer. Une fois les différentes phases du processus de développement présentées (section 1.1), nous nous concentrerons sur notre champ d'étude qui sont les activités liées aux tests d'intégration (section 1.2). Nous positionnerons notre travail qui est un *framework* BDD (*Behavior Driven Development*) pour la conception des campagnes de tests d'intégration (section 1.4), après avoir présenté différents travaux sur la transposition du génie logiciel à l'ingénierie système (section 1.3).

1.1 Système avionique : production, caractéristiques et nouvelles infra-structures

Le processus de développement d'un avion implique des dizaines de milliers de personnes venant de sites, d'entreprises et même très souvent de pays différents. Dans un tel contexte, les erreurs de conception ou d'implémentation sont souvent dues à des erreurs de communication. Le vocabulaire utilisé diffère en fonction des métiers et des disciplines. On peut, par exemple, citer le cas du projet Mars Surveyor 98 visant à envoyer sur l'orbite de Mars deux sondes spatiales. Un problème logiciel s'étant produit lors du calcul de l'altitude a causé la perte de la sonde Mars Climate Orbiter. Le rapport [78] produit quelques semaines après, montre que la perte de la sonde est due à une erreur de communication entre l'équipe principale du projet basée dans le Colorado et l'équipe chargée du système de navigation basée en Californie. Ces deux équipes utilisaient deux systèmes d'unités de mesure différents. L'équipe en Californie utilisait le système anglo-saxon pour produire le système de navigation alors que le reste du système était codé avec les unités du système métrique. Tout ceci montre que la communication est un point des plus importants dans le développement de systèmes complexes. C'est d'autant plus vrai pour les systèmes critiques car leurs défauts peuvent causer la perte de vies humaines, engendrer des pertes matérielles importantes ou avoir des conséquences environnementales graves [54].

Dans un autre rapport [45], Edward Weiler, qui était à l'époque directeur du pôle espace à la NASA disait "Les gens font parfois des erreurs". Cependant, selon lui le problème ne vient pas de l'erreur en elle-même mais plutôt du fait qu'elle n'ait pas été détectée en amont durant le développement. Edward Weiler met aussi en avant l'importance du processus de vérification et de validation qui aurait dû révéler cette erreur si les activités de vérification et de validation avaient été réalisées de manière plus rigoureuse. Plus une erreur est identifiée tardivement, plus elle est coûteuse à corriger.

Cette section aborde les aspects du processus de développement et les caractéristiques d'un système avionique. Nous étudions ensuite les différentes générations d'architectures qui peuvent prendre place dans un avion. Pour qu'une erreur dans le comportement d'un système avionique soit détectée au plus tôt, des activités de test prennent place tout à long du cycle de développement.

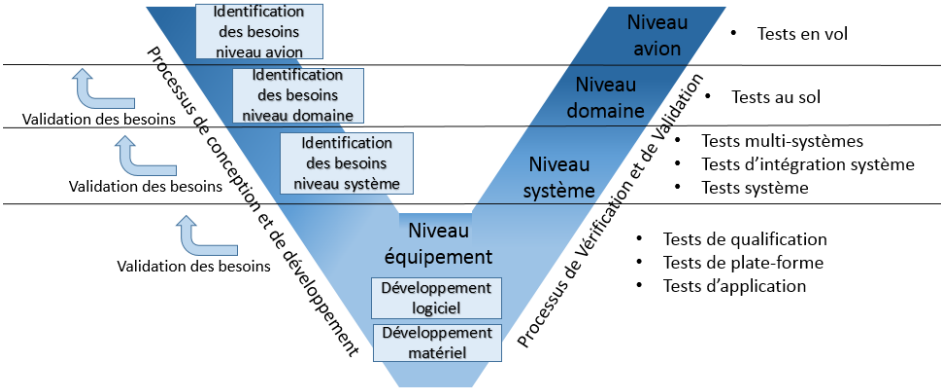
1.1.1 Processus de développement : les cycles en V

Le processus de développement d'un avion suit le modèle du cycle en V, comme présenté à la figure 1.1. Ce modèle permet de mettre en opposition le processus de développement qui suit une approche descendante et le processus de vérification et de validation qui suit une approche ascendante.

Le processus de développement d'un avion commence par une phase de spécification et de conception de l'avion dans sa globalité. Il se poursuit par un raffinement des spécifications et de la conception de chaque système le composant. Ce raffinement a pour but de découper

ce système en entités plus petites et moins complexes, jusqu'à arriver au bon niveau de granularité. Les systèmes qui composent un avion sont divers ; ils incluent le plus souvent une partie matérielle et une partie logicielle. Ces deux parties sont développées séparément puis sont assemblées durant la phase dite d'intégration. Le logiciel permettra de piloter et de fournir la logique pour alimenter en données et contrôler les équipements matériels qui à leur tour produiront des données qui seront fournies en entrée aux calculateurs.

FIGURE 1.1 – Développement et activités de tests



Des documents sont rédigés tout au long de la phase descendante pour rendre compte des spécifications du système et des choix de conception réalisés.

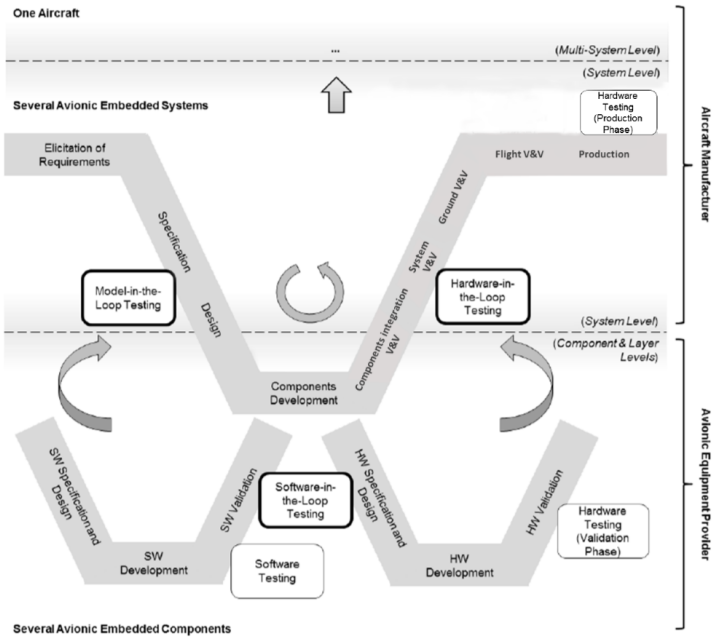
Cette représentation du processus de développement pourrait laisser penser que les activités de vérification et de validation ne sont réalisées qu'une fois le développement complet du système réalisé. Ce n'est pas le cas. Pour anticiper les changements dus à des choix erronés de conception ou à des exigences mal définies, d'autres processus itératifs entrent en jeu à chaque niveau de détail de la conception. Ces cycles itératifs sont représentés par les flèches de validation des besoins sur la figure 1.1.

Cette figure montre aussi les activités de test réalisées durant la phase ascendante du cycle en V. Ces activités sont regroupées selon les différents niveaux d'intégration d'un avion.

- Au niveau équipement, on retrouve les activités de test validant les applications logicielles et les plates-formes d'exécution. Finalement, les tests de qualification garantissent que les applications logicielles s'exécutent correctement lorsqu'elles sont hébergées par une plate-forme matérielle préalablement testée.
- Au niveau système, les tests s'assurent que les systèmes produits sont corrects et qu'ils s'interfacent correctement avec les autres systèmes de l'avion. Le niveau domaine intègre les systèmes d'un domaine et permet de réaliser les premiers tests au sol.
- Au niveau avion, tous les systèmes, préalablement certifiés, sont présents dans une première version de l'avion dans l'optique de réaliser des tests en vol.

Au niveau système, on retrouve un cycle en V par système à développer. La figure 1.2 représente le cycle en V pour un système donné. Dans ce cycle en V, on distingue deux autres processus respectant aussi le modèle en V : un pour le développement du matériel et un pour le développement du logiciel. Ces deux cycles sont représentés en bas de la figure et forment un modèle de développement spécifique appelé modèle en W.

FIGURE 1.2 – Modèle en W



Cette approche découpe le développement d'un système selon deux niveaux : un niveau système où l'intégrateur réalise la phase de spécification des besoins et des exigences et commence la conception globale du système (haut de la figure) et le niveau composant où les phases de développement de chaque partie d'un système sont réalisées par un fournisseur (bas de la figure). L'intégrateur réalise aussi les activités de vérification et de validation lors de l'intégration du système développé. Au-dessus du niveau système, on trouve le niveau multi-système qui vise à l'intégration de plusieurs systèmes préalablement testés, comme c'est le cas dans l'*iron bird* par exemple. De plus amples informations sur l'*iron bird* sont disponibles à la section 1.1.6.

Chacun des systèmes sont testés séparément pour vérifier que les spécifications sont bien respectées. Ces activités sont représentées par les phases de *Software testing* et de *Harware testing* en bas de la figure. Une fois le système développé et livré, l'intégration du système dans l'avion est à la charge de l'avionneur. Cette étape est représentée sur la figure 1.2 par l'activité *Components Integration V&V*. D'autres tests visent à valider au plus tôt,

la conception du système grâce à de la modélisation temps-réel (*Model-in-the-Loop*), le logiciel grâce à la simulation de la plate-forme matérielle d'exécution (*Software-in-the-Loop*) et l'intégration du matériel dans un environnement simulant les communications avec les autres systèmes de l'avion (*Hardware-in-the-Loop*). Ces types de test d'intégration seront détaillés dans la section 1.2.2.

1.1.2 Caractéristiques d'un système avionique

Un système avionique est un système temps-réel, critique, embarqué, réactif et tolérant aux pannes.

- **Temps-réel** : Les systèmes temps-réel sont divisés en deux catégories. Le temps-réel strict implique que la réponse du système lors d'une sollicitation doit toujours arriver dans un laps de temps donné. Le temps-réel souple autorise un temps de réponse supérieur à une contrainte temporelle, mais ce temps de réponse ne doit pas dépasser un certain nombre de cycles d'exécution. Dans le contexte avionique, les systèmes temps-réel sont majoritairement stricts. Cette caractéristique impacte très largement les choix de conception et d'implémentation que ce soit sur le logiciel ou sur le matériel.

- **Critique** : La défaillance d'un système critique a un impact direct sur la sécurité des personnes l'utilisant. Des méthodes de vérification et de validation sont donc mises en place pour assurer un fonctionnement fiable. De plus, des normes de certification existent pour assurer un niveau de sécurité en fonction de la criticité du système. La norme pour les systèmes complexes est la norme ARP4754 (*Guidelines For Development Of Civil Aircraft and Systems*) [67]. Elle permet de classifier, en fonction de leur criticité, les systèmes selon cinq niveaux : catastrophique, dangereux, majeur, mineur, sans impact. Elle propose aussi d'assigner un niveau de confiance dans le développement (*Development Assurance Level* - DAL) d'un système en fonction de son niveau de criticité (DAL A pour les plus critiques jusqu'à DAL E pour les moins critiques). La norme ARP4754 est complétée par la norme DO-178C (*Software Considerations in Airborne Systems and Equipment Certification*) [66] qui vise à certifier le logiciel embarqué à l'intérieur d'un système critique et la norme DO-254 (*Design Assurance Guidance for Airborne Electronic Hardware*) pour le matériel [39].

- **Embarqué** : Les systèmes qui composent un avion sont embarqués dans celui-ci ; ils sont composés d'une partie matérielle et d'une partie logicielle. Cette caractéristique implique qu'ils sont contraints au niveau spatial, car ils doivent être contenus dans le système qui les embarque. Ils sont également contraints au niveau énergétique car ils sont couplés au système d'alimentation de l'avion. Ils sont aussi contraints par la puissance de calcul et la place mémoire qui leur sont attribuées. L'augmentation de la capacité de calcul de ces systèmes a un effet direct sur la consommation en énergie.

- **Réactif** : Ces systèmes sont dits réactifs car ils sont en permanence en interaction avec leur environnement. Ils doivent réagir aux changements et s'ajuster continuellement. Ils échangent des données en temps réel pour réaliser les fonctions nécessaires à un avion. La limite de temps durant laquelle ces systèmes doivent réagir à un évènement pour que leur comportement ne soit pas jugés défaillant, montre le lien étroit entre le fait qu'ils sont réactifs et qu'ils sont temps réel. L'aspect réactif de ces systèmes demande de mettre en place des tests particuliers.

- **Tolérance aux pannes et propagation des erreurs** : L'avion doit, en toutes circonstances, réagir à son environnement en temps réel et la défaillance d'un de ses systèmes ne doit pas entraîner sa panne. Pour répondre à cette problématique, des mécanismes de redondance sont élaborés lors de la conception générale de l'avion. Ces mécanismes répondent aux contraintes de disponibilité des équipements et des fonctions avioniques. Ils permettent, par exemple, aux fonctions logicielles critiques d'être hébergées par deux calculateurs différents de l'avion. Ces calculateurs disposent d'un système de communication inter-modules et intra-module qui permet aux applications de coopérer dans la réalisation de certaines tâches. Ces systèmes de communication donnent la possibilité de changer d'application concourant à la réalisation d'une tâche lorsqu'une erreur provenant d'une des applications est détectée. Une plate-forme d'exécution logicielle est partitionnée en plusieurs entités. Elle peut donc héberger plusieurs fonctions. La mémoire n'est pas partagée entre les différentes partitions pour éviter la propagation d'erreurs pouvant être critiques.

Les systèmes embarqués au sein d'un avion ont une complexité toujours plus élevée de par les caractéristiques citées ci-dessus. L'état de ces systèmes dépend étroitement de l'état des systèmes avec lesquels ils communiquent. Les types de communication et les données transférées sont hétérogènes.

Toutes ces caractéristiques impliquent un effort accru pour la conception et le test des systèmes avioniques.

1.1.3 Découpage d'un système avionique

Le découpage de l'avion en domaines, systèmes, sous-systèmes, puis en équipements est établi lors de la phase de spécification du système global comme le montre la figure 1.1. Il permet de raffiner les spécifications du système depuis les fonctionnalités générales de l'avion jusqu'aux spécifications les plus détaillées de chaque équipement. A chaque étape de spécification, des documents de conception et d'exigences sont produits. Le bureau d'étude explicite les exigences de haut niveau et les besoins fonctionnels de l'avion et réalise la conception de l'architecture physique de l'avion. Des documents d'exigences et de description du système sont produits et sont liés aux exigences du niveau avion pour permettre la traçabilité.

Une fois que les fonctions de l'avion sont identifiées, elles sont regroupées en plusieurs

domaines. Ces domaines correspondent à la classification ATA (*Air Transport Association of America*) qui propose un découpage des systèmes d'un avion en 100 chapitres [79]. Les tableaux 1.1 et 1.2 présentent quelques-uns de ces chapitres et équipements. Il s'agit ensuite d'identifier et de concevoir les systèmes qui répondront aux fonctions de l'avion.

TABLE 1.1 – Exemple de chapitres ATA et des sous-systèmes correspondants (1/2)

Domaine système de l'aéronef		Équipements
21	Air conditionné et pressurisation	Compression
		Distribution
		Pressurisation
		Chauffage
		Climatisation
		Contrôle de la température
22	Vol automatique	Pilote automatique
		Correction de la vitesse
23	Communications	Communication vocale
		Communication satellite
		Informations pour les passagers et divertissements
24	Génération électrique	Contrôle du générateur
		Génération de courant alternatif
		Génération de courant continu
		Alimentation externe
		Distribution du courant alternatif
		Distribution du courant continu
27	Commandes de vol	Aileron et tab
		Gouvernail de direction et tab
		Gouvernail de profondeur et tab
		Stabilisateur horizontal
		Dispositif hypersustentateur
42	Avionique modulaire intégré	Système principal
		Composants réseau
44	Systèmes de la cabine	Système de cabine principal
		Système de divertissement en vol
		Système de communication externe
		Système de gestion de la cabine

Les deux cas d'études que nous avons développé au cours du projet FUI ACOVAS (outil Agile pour la COnception et la VAlidation Système) sont tirés de l'ATA 21 (Air conditionné et pressurisation) et de l'ATA 42 (Avionique modulaire intégré). Le premier cas d'étude correspond aux tests réalisés par un fournisseur de systèmes alors que le deuxième

correspond aux tests réalisés par l'avionneur lors de l'intégration des systèmes produits par les fournisseurs.

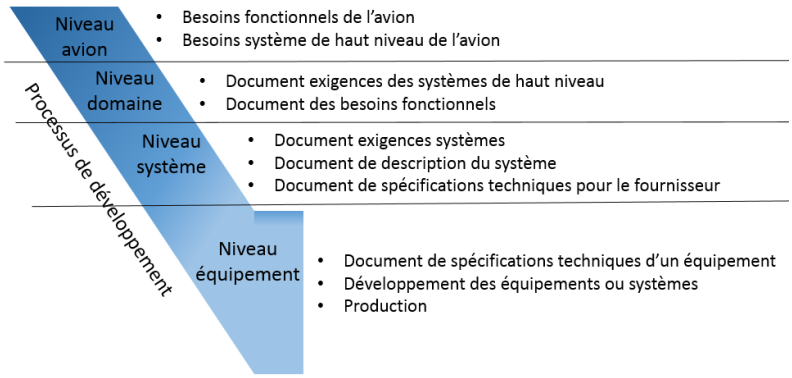
TABLE 1.2 – Exemple de chapitres ATA et des sous-systèmes correspondants (2/2)

Domaine structure		Équipements
52	Portes	Portes pour les passagers et l'équipage
		Sortie d'urgence
		Porte de la soute
		Porte pour la maintenance des systèmes
		Escaliers d'entrées
57	Ailes	Aile centrale
		Aile externe
		Bout de l'aile
		Bord d'attaque de l'aile
		Bord de fuite de l'aile
		Ailerons et élevons
Domaine moteur		
72	Moteurs	Compresseur
		Chambre de combustion
		Turbine
73	Carburant moteur et régulation	Distribution
		Diviseur de débit
		Contrôle du carburant
80	Démarrage	Système de démarrage
		Moteurs

Un domaine est composé de plusieurs systèmes. L'architecture d'interconnexion des systèmes définit les interfaces de communication interne et externe auxquels les systèmes devront se conformer. Le choix de l'architecture a aussi un impact sur la structure interne de chaque système. Il doit prendre en compte les exigences en matière de sécurité, les possibles propagations d'erreurs et principalement les exigences métier. Le bureau d'étude produit un document de spécification technique d'acheteur (*Purchaser Technical Specification* ou PST) du système pour le transmettre au fournisseur afin qu'il réalise le système souhaité. La figure 1.3 résume l'ensemble des documents produits lors de la conception d'un avion.

Le niveau le plus bas dans ce découpage est le niveau équipement. A ce niveau, les spécifications techniques du système sont détaillées pour chaque équipement qui le compose. Un document PST est aussi produit au niveau équipement. Un équipement comprend une partie matérielle (contrôleur, capteur, actionneur) et le logiciel le pilotant (système d'exploitation, système applicatif). Pour chacun de ces éléments matériels et logiciels, des documents de spécification et de conception ainsi qu'un PST sont également produits. Les

FIGURE 1.3 – Documents pour la conception et activités de développement d'un avion



PST servent ici aussi de cahier des charges qui sont délivrés aux fournisseurs pour qu'ils réalisent le développement du matériel et du logiciel.

1.1.4 Différents types d'architecture avionique

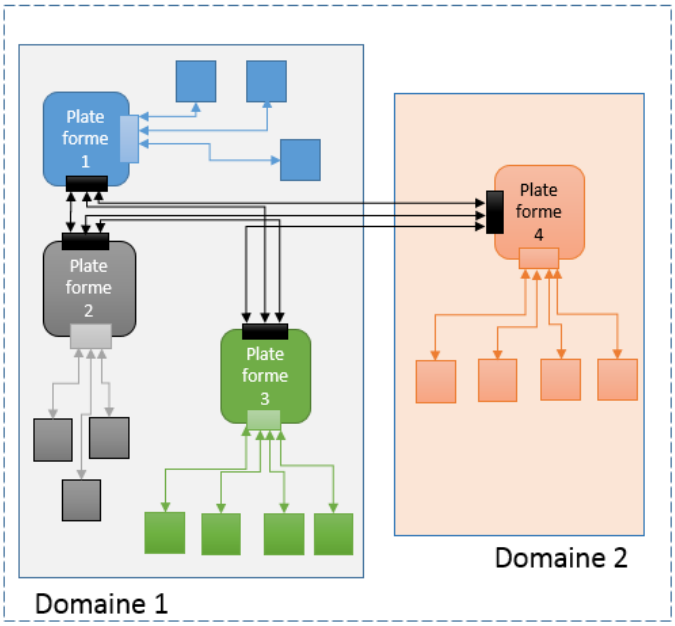
Une architecture avionique définit comment les unités de calcul et les différents capteurs et senseurs de l'avion sont connectés entre eux. Une étude des différentes architectures avioniques a été réalisée par Bill Filmer [35]. Généralement, on les classe en trois grandes catégories : les architectures indépendantes, les architectures fédérées et les architectures intégrées.

Les architectures indépendantes proposent une forte séparation entre les systèmes appartenant à des domaines différents. Elles ont pour but de séparer chaque système de contrôle composé d'un calculateur, des capteurs correspondants et d'autres équipements associés. Chacun de ces systèmes réalise les fonctions avioniques d'un même domaine. Les systèmes de contrôle n'ont pas de connexions entre eux ce qui implique qu'il ne peut pas y avoir de propagation d'erreurs d'un système à un autre.

La logique autour des architectures fédérées consiste à séparer fortement chaque système tout en autorisant la communication inter-systèmes par des connexions point à point entre certains systèmes. La complexité de l'architecture est augmentée en partageant des données entre plusieurs fonctions de systèmes différents. Avec ce type d'architecture, une erreur peut se propager d'un système à l'autre ; sa détection et sa tolérance sont gérées *via* des mécanismes logiciels et grâce à la redondance physique d'équipements. La figure 1.4 montre un exemple d'une telle architecture.

Une nouvelle génération d'architecture a été développée, remplaçant les calculateurs spécifiques à chaque système par des calculateurs standardisés pouvant héberger plusieurs fonctions logicielles venant de plusieurs systèmes. Ces plates-formes sont appelées des mo-

FIGURE 1.4 – Architecture fédérée

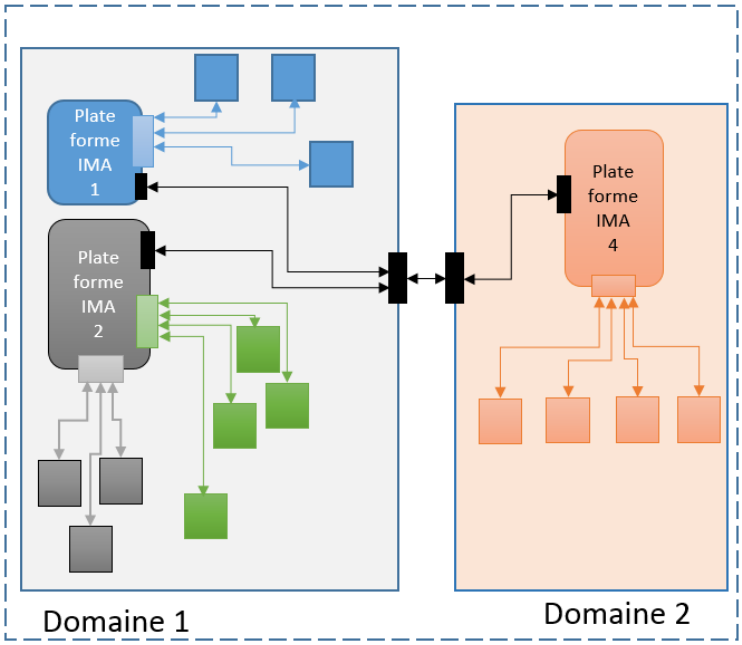


dules IMA (*Integrated Modular Avionic*) ou calculateurs IMA. Ces modules sont interconnectés par des bus de données et des *switchs* de communication. Ce type d'architecture met l'accent sur la modularité et la flexibilité de la conception des nouveaux systèmes embarqués ; elle est aussi, de par sa nature, plus tolérante aux fautes même si le traitement des erreurs nécessite une certaine complexité de mise en œuvre. La figure 1.5 montre un exemple d'une telle architecture. La standardisation des modules de calcul IMA les rend interchangeables : l'avionneur peut donc changer de fournisseur. Cette architecture permet de réduire les coûts de développement car les activités d'intégration de ces calculateurs standardisés et conçus pour les besoins particuliers de l'avionique (*special-to-purpose*) sont grandement simplifiés.

L'évolution autour des architectures concerne essentiellement les moyens de communication entre les différents systèmes et la standardisation des calculateurs. Les communications entre les calculateurs standardisés d'une architecture IMA se font par des *switchs* de communications représentés en noir sur la figure 1.5, ce qui permet de réduire le nombre de connexions physiques dans l'avion, en connectant les systèmes non co-géolocalisés à un même bus de données.

Ces architectures permettent d'héberger des fonctionnalités de différents systèmes sur un même calculateur. Un calculateur doit alors partager ses ressources pour des fonctionnalités de différents systèmes : ressources de calcul mais aussi ressources mémoire. Afin de retrouver le niveau d'encapsulation des architectures indépendantes, un partitionnement

FIGURE 1.5 – Architecture IMA



des ressources est réalisé [51].

1.1.5 *Interface Control Document* (ICD)

Ces nouveaux types d'architecture demandent un effort supplémentaire lors de la définition des interfaces de chaque système. Chaque système consomme et produit des données *via* ses interfaces. La définition des interfaces d'un système nécessite la mise en place d'un document de contrôle des interfaces (*Interface Control Document*). Un ICD est un document contenant les interfaces d'un système embarqué. Le nom d'ICD est un nom générique et sa structuration peut varier en fonction des entreprises, et même au sein d'une même entreprise. Le changement le plus notoire est la façon d'identifier de manière unique les différentes interfaces d'entrée et de sortie du système. Ces interfaces d'entrée et de sortie seront le support pour l'envoi et la réception des paramètres avioniques. Un paramètre avionique correspond à un signal contenant un message qui sera délivré ou consommé par le système. Ce document permet aussi de connaître quel est le type d'un paramètre d'application (entier, réel, booléen ...), ses valeurs minimale et maximale, ou encore le taux de rafraichissement de sa valeur. La transmission des messages peut se faire selon deux modes de communication [41]. Le premier, le mode *sampling* transmet des messages de structure homogène, où seule la valeur des paramètres contenus dans le message change. Les ports

d'entrée et de sortie conservent uniquement la dernière occurrence du message à envoyer ou du message reçu car seules les valeurs les plus récentes sont pertinentes. Le mode *queuing* transmet des messages de structure hétérogène : deux messages reçus ou envoyés par un même port ne contiennent pas obligatoirement les mêmes paramètres. L'information transportée par les messages du mode *queuing* est variable demandant aux ports de stocker toutes les occurrences de messages reçus ou à envoyer. Le mode *queuing* lève une alarme lorsque la file de transmission est pleine ou quand la file de réception est vide.

Plusieurs niveaux de détail sont fournis. Au plus bas niveau, on retrouve les identifiants des connecteurs du système sous test. Chaque connecteur comprend plusieurs broches (ou *pins*). Il permet de renseigner les bus de communication qui sont rattachés au connecteur physique. A un niveau logique, on retrouve les messages qui transitent sur ces bus. Puis, au niveau logique le plus haut, les paramètres d'application et les signaux qui lui sont rattachés sont décrits.

FIGURE 1.6 – Exemple de paramètre ICD

```
Application Name: PRIM1A Bus Name: A01_IN Variable name: Temperature_cabine1
   type: Float unit: °C Media: AFDX
```

La figure 1.6 est l'exemple d'un paramètre contenu dans un fichier ICD. Dans cet exemple, le paramètre est unique grâce au triplet composé du nom d'application, du nom du bus et du nom de variable qui lui sera attribué. A ce paramètre sera aussi associé un type de données, une unité de mesure et un média de communication. Le paramètre décrit à la figure 1.6 correspond à la température de la zone nommée `Temperature_cabine1` qui sera reçue sur le bus `A01_IN` par l'application `PRIM1A` qui sera hébergée sur le système auquel l'ICD appartient.

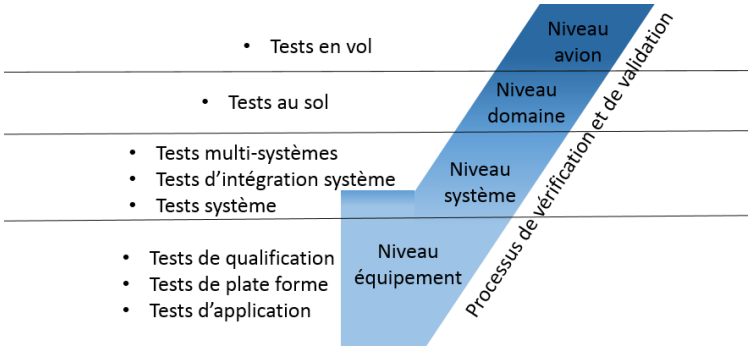
Ces paramètres sont transmis sur le réseau *via* des média de communication qui sont de plusieurs types : analogique, discret, AFDX (*Avionics Full-Duplex Switched Ethernet*) [2], conforme au protocole ARINC 429 [6], CAN (*Controller Area Network*), ou MIL-STD-1553B [28].

1.1.6 Les phases de test

Cette section s'attache à décrire les phases de test d'un système avionique complet depuis les tests unitaires portant sur le logiciel et le matériel jusqu'aux tests en vol. Pour le développement de systèmes avioniques, le processus suivi est un processus dirigé par les plans de test. Il est décrit par Ian Sommerville d'un point de vue logiciel [76]. Ce processus est adapté à chaque entreprise en fonction de ses besoins et de ses stratégies de test. La figure 1.7 schématise les activités de test dans la branche ascendante du cycle en V.

Les premières phases de test sur les éléments réels sont réalisées sur le logiciel et les plates-formes qui l'exécuteront ainsi que sur les équipements annexes à ces systèmes, comme les capteurs et les actionneurs, les câbles ... Chaque équipement ou logiciel intégré dans

FIGURE 1.7 – Phases de test



l'avion sera préalablement testé en isolation. Le développement des systèmes se faisant en parallèle, il est donc possible de réaliser les tests en parallèle. On peut considérer un système embarqué comme étant composé d'une application exécutée sur un système d'exploitation grâce à une plate-forme matérielle. La plate-forme matérielle héberge un système d'exploitation temps-réel. La plate-forme et le système d'exploitation sont testés séparément dans un premier temps, puis en intégration lors de la phase de test de la plate-forme.

Les fonctions logicielles et les algorithmes sont testés dans un environnement d'exécution temps-réel simulé : ce sont les tests d'application. Puis, cette simulation est remplacée par une modélisation de la plate-forme qui l'accueillera réellement dans l'avion. Ces tests correspondent aux tests d'application.

Concernant la plate-forme, le test s'établit en plusieurs étapes :

– Au niveau matériel, les équipements (capteurs, contrôleur, ...) subissent deux types de test : des tests fonctionnels et des tests environnementaux permettant de vérifier que le matériel peut assurer son service dans les conditions de vol normales et exceptionnelles (vibrations, décompression, pressurisation, variations de températures, humidité ...). Les tests environnementaux sont définis par les standards et normes qui permettent la certification.

– Puis le système d'exploitation est aussi testé, en se concentrant sur son aspect fonctionnel. Les pilotes de communication sont testés *via* un environnement simulé qui produit les messages entrants et sortants.

– Enfin, des tests d'intégration du système d'exploitation sur le matériel sont réalisés. Une simulation logicielle du comportement et des caractéristiques du matériel permet de tester le système d'exploitation dans des conditions proches de l'intégration sur le matériel réel. Le système d'exploitation est aussi testé sur un matériel réel qui aura été vérifié au préalable par les divers tests cités ci-dessus. L'intégration du système d'exploitation sur le matériel constitue la plate-forme qui pourra ensuite héberger les

applications avioniques.

Une fois ces trois types de test réalisés, la plate-forme matérielle pilotée par le système d'exploitation, peut alors accueillir les applications logicielles qu'elle hébergera réellement dans l'avion. Les tests d'intégration des applications logicielles sur la plate-forme matérielle correspondent aux tests de qualification.

On obtient un système lorsqu'une plate-forme contrôlée par un système d'exploitation permettant d'exécuter une application avionique est intégrée à des équipements externes additionnels. Ce système est testé et correspond à l'activité tests système de la figure 1.7.

Pour les tests d'intégration système, des modèles permettent de simuler les autres systèmes interagissant avec le système testé ainsi que l'environnement extérieur. A ce niveau, le concept de redondance est mis en place. De la même manière que les tests d'intégration sont réalisés, des tests multi-systèmes sont possibles lorsque plusieurs systèmes réels sont intégrés ensemble.

Les systèmes sont alors intégrés domaine par domaine, et les systèmes appartenant à un autre domaine sont simulés lors de cette phase de test. Au niveau domaine, les premiers tests au sol sont réalisés.

Une des dernières phases d'intégration assemble les systèmes de la plupart des domaines. Le but est de monter une première version de l'avion en taille réelle (*iron bird*) ou non (*mini-bird*) et de vérifier que cette version est opérationnelle i.e. les systèmes fonctionnent correctement lors d'une exécution individuelle et simultanée. Dans un *iron bird*, les systèmes, réseaux informatiques, hydrauliques, de ventilation, ... sont intégrés et pilotés par les calculateurs et contrôleurs réels. L'*iron bird* est donc la version finale de l'intégration de l'avion qui permet de réaliser les tests au sol puis de réaliser les premières simulations de vol. Tant que ces batteries de tests ne sont pas validées, aucun test en vol ne sera réalisé. Lors de la réalisation de la première version de l'avion, les premiers tests en vol sont enfin réalisés.

1.2 Un focus sur les tests d'intégration

Le développement d'un avion peut être divisé en deux grandes phases : la phase de conception et de développement et la phase d'intégration et de test. Durant la première phase, la vérification et la validation sont effectuées au niveau des spécifications, et durant la deuxième elles sont réalisées au niveau implémentation. Le processus de Vérification et de Validation (V&V) intervient principalement dans la phase montante du cycle en V qui liée à l'intégration des systèmes [5]. La vérification détermine si une partie ou l'ensemble du système respecte bien ses spécifications. La validation s'assure que le système reflète les besoins et attentes des utilisateurs (opérateurs, personnels de maintenance, agences de règlementation et les utilisateurs finaux). Plus précisément :

– **La vérification au niveau élicitation et spécification des besoins** garantit que les spécifications et besoins de bas niveau sont conformes aux spécifications de

plus haut niveau.

- **La validation au niveau élicitation et spécification des besoins** concerne les éléments de décision pris lors de la phase de conception.

- **La vérification au niveau implémentation** montre la conformité des systèmes, sous-systèmes, composants matériels ou logiciels avec les spécifications et détails de conception.

- **La validation au niveau implémentation** s'assure que les besoins de l'avionneur ont bien été implémentés dans le système embarqué livré.

Les activités de vérification comprennent : l'analyse statique, la vérification de modèles et la preuve formelle. Des revues et des inspections sur les documents de V&V ont pour but d'améliorer leur qualité.

Le test est le principal moyen de validation. Un test interagit avec le système sous test (*System Under Test* - SUT) pour le stimuler avec des données d'entrée dans le but d'obtenir des informations en sortie qui seront ensuite analysées pour les comparer aux comportements attendus. Un test comporte plusieurs cas de test. Un cas de test est composé par un ensemble de valeurs en entrée, des conditions d'exécution et la description du résultat attendu. Le processus dédié aux tests commence par leur conception, i.e. les choix des cas de test et la description de la logique de chaque cas de test. La deuxième phase correspond à l'implémentation qui prépare et configure l'environnement et les équipements nécessaires à la réalisation du test ; elle inclue la production du code exécutable des tests automatisés.

Il n'est pas possible d'assurer un comportement correct pour l'ensemble des cas d'exécution d'un système complexe. La vérification et la validation servent donc à augmenter le niveau de confiance des systèmes embarqués.

Dans cette section, nous nous concentrons principalement sur les activités de test d'intégration (section 1.2.1) qui sont au cœur des préoccupations de cette thèse. Nous nous intéressons aux tests dits *In-the-Loop* qui sont une façon de réaliser des tests d'intégration pour les systèmes embarqués (section 1.2.2). Nous aborderons pour finir l'implémentation des procédures de test (section 1.2.4) sur un moyen d'essai dédié : le banc de test (section 1.2.3). La phase d'intégration, portant sur le système complet, implique des tests très coûteux à exécuter en ressources et personnels. Les tests d'intégration sont en général non automatisés.

1.2.1 Les phases de test d'intégration

La phase d'intégration permet de s'assurer que l'ensemble des composants d'un avion ont le comportement prévu lors de leur exécution conjointe. Lors de cette phase, certains comportements doivent être simulés pour se rapprocher au maximum des conditions de vol réel. Les tests d'intégration accompagnent cette phase et se font grâce à des bancs de test qui permettent de connecter les SUT avec les outils de test et de simuler, grâce à des

modèles, l'environnement extérieur et les comportements des composants non-disponibles lors du test. Ces simulations sont utilisées pour stimuler le SUT dans un environnement le plus proche possible du contexte d'exécution réel. Les bancs de test transmettent donc des informations vers le SUT et reçoivent des données qui seront sauvegardées pour vérifier ultérieurement leur conformité.

L'intégration de ces systèmes suit alors une stratégie d'intégration. Généralement, l'intégration peut se faire de deux manières :

- L'approche non-incrémentale ou *big bang* intègre tous les composants et systèmes en même temps.
- L'approche incrémentale intègre un ou plusieurs composants à chaque nouvelle étape d'intégration. En ce qui concerne l'approche incrémentale, on distingue :
  - **L'approche d'intégration ascendante** est un processus démarrant depuis le niveau le plus bas, c'est-à-dire le niveau équipement.
  - **L'approche d'intégration descendante** est un processus démarrant depuis le niveau le plus haut, c'est-à-dire le niveau avion.
  - **L'approche d'intégration en fonction des composants disponibles** vise à intégrer uniquement les composants dont le développement est terminé. Les composants en cours de développement sont simulés.
  - **L'approche d'intégration par fonction de l'avion** a pour but de réaliser l'intégration des fonctionnalités séparément conçues.
  - **L'approche d'intégration de l'extérieur vers l'intérieur** vise à créer deux processus d'intégration, un ascendant partant du niveau le plus bas et un descendant partant du niveau le plus haut. Le but étant de faire converger ces deux processus.
  - **L'approche d'intégration de l'intérieur vers l'extérieur** vise à créer deux processus d'intégration, un ascendant et un descendant commençant tous deux à un niveau moyen de détail. Il divergent ensuite pour atteindre respectivement le niveau le plus bas et le niveau le plus haut.
  - **L'approche par processus métier** vise à créer un processus d'intégration par type de métier. Les systèmes seront intégrés en fonction de leur place dans la réalisation des cas d'utilisation du métier.
  - **L'approche par complexité décroissante** lève les incertitudes liées à une criticité d'intégration élevée.

Les approches incrémentales demandent le développement de simulations ou de *stubs* des composants qui ne sont pas encore intégrés. Une approche non incrémentale est souvent trop contraignante du fait que tous les composants doivent être développés avant l'intégration. Cette stratégie n'est pas viable pour l'intégration de ce type de système.

Dans la plupart des cas, plusieurs stratégies incrémentales sont utilisées conjointement pour répondre aux retards de livraison des fournisseurs. La stratégie la plus usitée est l'approche ascendante qui commence au niveau équipement par l'assemblage de la plate-forme et des applications qui vont s'exécuter sur cette plate-forme. Le processus d'intégration vise

à intégrer tous les composants de chaque système au niveau domaine et ensuite au niveau avion.

L'intégration suit globalement une approche ascendante, mais d'autres approches sont utilisées parallèlement. Par exemple, des stratégies par processus métier pourront être envisagées en complément au niveau domaine ou avion alors que des approches par complexités décroissantes ou par composants disponibles peuvent être imaginées au niveau équipement.

Parce que ces systèmes sont très difficiles à intégrer et que la phase d'intégration est proche de la fin du cycle de développement d'un avion, il s'avère nécessaire de faire au plus tôt des tests d'intégration avec les autres systèmes simulés dans un type de test particulier appelé *In-the-Loop testing*.

1.2.2 *In-the-Loop testing*

Les tests d'intégration utilisent un banc de test pour interagir directement avec le matériel et permettre la simulation d'un modèle d'environnement. Pour les tests sans interaction avec le matériel, le simulateur peut être hébergé sur un ordinateur classique. Le modèle d'environnement simule, en partie ou complètement, la présence des autres systèmes de l'avion nécessaires à la réalisation du cas de test souhaité. Par exemple, pour les tests des différentes configurations d'un calculateur IMA, le modèle d'environnement simule le comportement des systèmes permettant la réception centralisée des messages d'erreur des calculateurs et la réaction du système par rapport aux différentes erreurs. Ce modèle permet de produire les données d'entrée dans le but de stimuler le SUT. Celui-ci réagit à ces stimuli en produisant des données en sortie qui sont réinjectées dans le modèle d'environnement. Le modèle pourra alors prendre en compte ces données pour ensuite produire les nouvelles données qui seront de nouveau utilisées en entrée du SUT. Ces mécanismes permettent de simuler les échanges de données cycliques et ininterrompus des systèmes testés comme décrit à la figure 1.8.

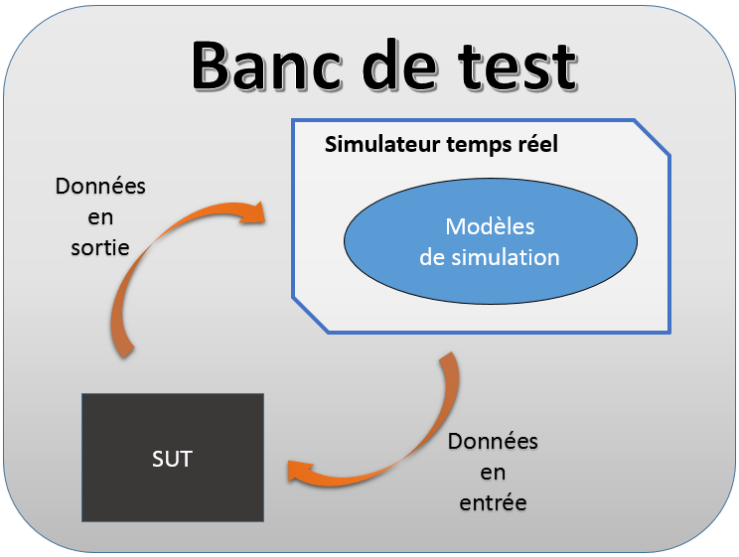
On peut constater que le SUT est vu comme une "boîte noire". En effet, les tests *In-the-Loop* ne servent pas à vérifier si les mécanismes internes du SUT sont corrects, mais à vérifier que son comportement en interaction avec les autres systèmes de l'avion l'est.

Le modèle de simulation est exécuté par un simulateur temps-réel et la connexion physique entre le simulateur et le SUT est prise en charge par le banc de test. La simulation temps-réel est utile pour remettre le SUT dans des conditions d'exécution proches des conditions réelles. L'architecture physique du banc permet de reproduire la connexion du matériel à son environnement simulé ou émulé.

Il existe trois types de test *In-the-Loop* : *Model-in-the-Loop* (MiL), *Software-in-the-Loop* (SiL) et *Hardware-in-the-Loop* (HiL).

- Le MiL est l'activité qui se produit le plus tôt dans le cycle de développement. Les tests sont réalisés sur les modèles comportementaux des systèmes. Ces modèles sont produits en fonction des besoins fonctionnels et des caractéristiques des systèmes en cours de conception. Il existe un modèle de simulation par catégorie de test. Un

FIGURE 1.8 – Boucle des données de test pour du test *In-the-Loop*



modèle peut, par exemple, permettre d'optimiser la longueur des câbles pour alimenter électriquement tous les composants d'un avion. Un autre modèle peut simuler le réseau informatique produit par l'interconnexion des calculateurs IMA. Ces modèles peuvent donc simuler l'architecture interne des composants, tout comme les interactions entre eux. Ils sont développés initialement, avant tout développement de matériel et de logiciel. Ces tests ont pour but de valider les exigences et les choix de conception des modèles comportementaux.

- Le SiL prend en compte le logiciel réel et l'exécute sur une plate-forme simulée. Dans ce cas, le modèle d'environnement émule le comportement du matériel et du système d'exploitation du calculateur qui contrôlera ce logiciel. Ces tests ont pour but de valider que le logiciel s'exécutera correctement sur la plate-forme matérielle, celle-ci étant encore simulée par des modèles.

- Les tests HiL sont réalisés sur le logiciel réel qui sera exécuté sur le matériel tel qu'il sera configuré dans l'avion final. Cette phase intervient au début de la phase d'intégration lorsque le matériel et le logiciel ont été développés et testés. Le modèle d'environnement ne simule que les interactions avec les systèmes concourants à l'exécution d'un ensemble de cas de test pour un SUT donné. Les tests HiL visent à s'assurer que l'assemblage logiciel-matériel représentant un système s'intègre correctement aux systèmes de l'avion.

1.2.3 Le banc de test

Le but d'un banc de test est de mettre en place des moyens de communication entre les systèmes que l'on souhaite tester pour en vérifier le comportement. Les bancs d'essai sont principalement utilisés pour valider des systèmes composés de matériels et de logiciels. Un banc de test comprend un séquenceur, une interface physique permettant de connecter le SUT aux ressources de test, un système d'alimentation du SUT, un simulateur temps-réel et un ensemble d'outils externes.

Le séquenceur est là pour exécuter les tests automatiques et semi-automatiques. Son but est d'exécuter la liste d'instructions contenues dans les tests formalisés. Ces tests formalisés prennent la forme de procédures de test écrites dans un langage informatique et sont donc exécutables.

Le SUT est connecté physiquement aux ressources de test grâce à une partie du banc de test appelée baie de test qui fait le lien entre le séquenceur, le modèle de simulation et le SUT lui-même. Cette baie permet de simuler les communications avec le monde extérieur. Le SUT est aussi connecté à un module d'alimentation souvent intégré à la baie de test. Une baie de test est modulaire car les cartes de communication qui la composent peuvent être adaptées au système que l'on teste. Il existe plusieurs types de carte en fonction du moyen de communication utilisé (ARINC, AFDX ...).

Le simulateur temps-réel permet de simuler les interactions entre le SUT et les éléments interagissant avec lui. Ce simulateur sera alors utile pour intégrer de manière artificielle le SUT dans un contexte d'exécution le plus proche possible des conditions d'exécution réelles.

Les outils externes sont soient des outils développés spécifiquement pour les besoins particuliers du test, soient des outils du marché (COTS [1]). Ils permettent de réaliser des actions spécifiques sur le SUT ou d'enregistrer des données produites par le SUT. On peut par exemple citer l'analyseur de réseau Wireshark qui permet d'enregistrer des trames circulant sur le réseau entre le SUT et le banc. L'analyse des trames réseau par Wireshark procure des informations sur l'état du SUT au cours d'un test. Les outils permettront d'analyser les données pour vérifier *a posteriori* si le comportement du SUT correspond au comportement attendu.

Un banc de test peut exécuter des tests automatiques, semi-automatiques ou manuels en fonction du niveau d'automatisation du pilotage des outils externes au test et en fonction du niveau d'effort de formalisation engagé pour la procédure de test. Les bancs de test sont de plusieurs types. On peut citer par exemple les bancs d'essai utilisés en phase de développement d'un système ou encore le banc de validation qui permet de valider le système avant la mise en production.

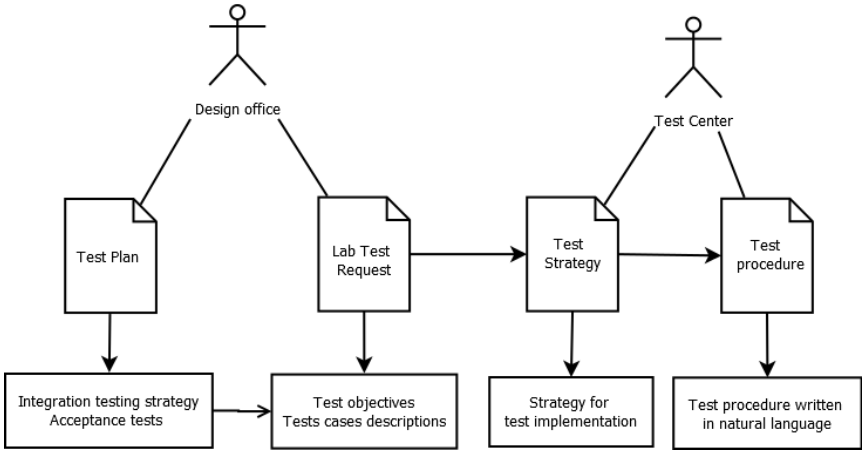
1. Les équipements *commercial off-the-shelf* sont des équipements facilement disponibles sur le marché n'ayant pas été développés pour un domaine d'application particulier.

1.2.4 Des exigences aux procédures de test

Après avoir présenté les spécificités des tests système pour des systèmes complexes en termes de stratégie d'intégration et de moyen d'essais, nous nous intéressons maintenant au flux de données menant aux procédures de test et caractérisant là aussi un processus particulier.

Les procédures de test prennent la forme de documents qui permettront d'exécuter un test sur un banc de test pour un système donné. Elles sont exécutées soit manuellement par un opérateur de test soit automatiquement dans un langage interprétable par le banc de test. Elles sont souvent manuelles et réalisées au travers d'interfaces graphiques représentant le SUT. Toutes les données observées sont loguées afin d'être analysées, de façon plus détaillée par un *data-viewer*. Cette façon de faire peut s'avérer nécessaire dans le contexte de certification. Les procédures de test ont différents niveaux de formalisation : le niveau informel, le niveau semi-formel et le niveau formel. Le niveau informel correspond aux procédures de test en langage naturel. L'utilisation du langage naturel pour la description des procédures de test peut entraîner des erreurs d'interprétation et d'incompréhension dues au fait qu'un même concept peut être décrit de multiples façons. Les procédures semi-formelles utilisent des techniques de structuration pour réduire la variabilité et les ambiguïtés des descriptions informelles. Les procédures formalisées quant à elles, sont produites à partir d'un langage informatique formel, qui possède donc une syntaxe et une sémantique bien définies. On peut par exemple citer le langage TTCN-3 (*Testing and Test Control Notation*) [32] pour la spécification des tests en boite noire dans un système distribué [69] ou des langages de programmation classiques comme Python, C, scripts shell, dialectes XML, ...

FIGURE 1.9 – Flux de documents d'une procédure



Dans le cas des procédures de test d'intégration, nous présentons à la figure 1.9 le *workflow* de documents associé à la production d'une procédure de test HiL informelle. Ce

workflow est un exemple prenant place pour la création de procédures de test d'intégration chez l'avionneur Airbus, partenaire du projet ACOVAS. Deux acteurs entrent en jeu dans la réalisation des procédures de test, le bureau d'étude et le centre de test.

Les procédures de test se basent sur le respect d'un plan de test. Ces plans de test sont élaborés en fonction des spécifications du système et des contraintes liées à sa certification, ils sont produits par le bureau d'étude. Le plan de test définit la stratégie globale d'intégration et décrit les critères d'acceptation pour un ensemble de tests. A partir de celui-ci, le bureau d'étude fournit les cas de test à l'équipe de testeurs dans un *Lab Test Request* (LTR). Le LTR se concentre sur la définition d'objectifs de tests concrets pour une version spécifique du système. Il contient aussi une description succincte de chaque objectif que devra implémenter un cas de test de la procédure développée. Un objectif de test correspond à un besoin, une fonctionnalité, ou à un comportement spécifique du système qu'il faut vérifier. Il est décrit par un nom, son intention, les conditions initiales du système, une description simple des actions à réaliser pour mettre le système dans les conditions nécessaires à la vérification de l'objectif et enfin, les résultats attendus. L'objectif de test est décrit indépendamment du moyen de test. Enfin, le centre de test produit la procédure de test à partir de la stratégie de test et des objectifs du LTR. Une procédure de test est un ensemble de cas de test. Un cas de test est rattaché à un ou plusieurs objectifs de test et est composé d'un ensemble d'instructions décrites en langage naturel (principalement en anglais) et stockées dans un fichier texte. Ces procédures seront ensuite exécutées manuellement sur un banc de test, instruction par instruction.

1.2.5 Les langages de test

Les langages de test ne sont pas forcément spécifiques aux besoins des tests dans un contexte avionique, mais ils répondent aux problématiques de vérification de systèmes composés d'une partie matérielle et d'une partie logicielle. Nous retrouvons dans cette liste des langages de modélisation comme TestML [43], ou encore des langages dont la forme est plus classique comme par exemple TTCN-3 [32].

TestML est un langage visant à rendre un cas de test compatible aux différents niveaux d'abstraction qui existent lors des tests de systèmes embarqués. Il a été spécialement conçu pour répondre aux problèmes industriels que l'on retrouve lors du développement de systèmes automobiles. Comme nous l'avons vu, les tests dits *In-the-Loop* sont présents sous trois formes dans le développement de systèmes embarqués. Pour rappel, ces formes sont *Model-In-the-Loop*, *Software-In-the-Loop* et *Hardware-In-the-Loop*, chacune correspondant à un niveau de développement ou d'intégration différent. Le langage TestML a pour avantage de couvrir les différentes étapes des tests d'intégration *In-the-Loop* à lui seul. Il a donc pour but de rendre les cas de test d'intégration indépendants du niveau d'abstraction du test. Actuellement, ces tests ont des langages de spécification et des moyens de test différents, ce qui ne rend pas possible leur réutilisation lors d'une autre étape de validation que celle pour laquelle ils ont été conçus. Le but de TestML est de fournir un format de mo-

délisation commun qui permettra ensuite la transformation de ces modèles vers l'ensemble des langages d'exécution utilisés dans le contexte du développement automobile.

Il existe aussi d'autres formes de langages de test dont TDL (*Test Description Language*) [82]. TDL est un nouveau langage pour la spécification des tests et la présentation des résultats d'exécution d'un test. Il permet de réduire le fossé entre les spécifications de test, c'est-à-dire les descriptions de haut niveau des cas de test, et le code complexe qui exécutera automatiquement ce test grâce au langage TTCN-3. Il donne la possibilité de créer une liste d'objectifs et de les rattacher à des cas de test. Ces cas de test décrivent le comportement du SUT alors vu comme une boite noire. Ils sont composés d'un nom, d'un ensemble de blocs de comportement (nommés `CompoundBehaviour`) décrivant les interactions avec les autres systèmes. Ces interactions sont décrites : le système émetteur, le système récepteur et le message transmis durant l'interaction. Les systèmes récepteurs et émetteurs sont des instances de systèmes spécifiés grâce au langage TDL. TDL permet d'établir le lien avec des outils de modélisation des exigences et des notations comme *User Requirements Notation* [4], *Use Case Maps* [3] pour la définition des scénarios ou *Goal-oriented Requirement Language* [84] pour la gestion des exigences non-fonctionnelles. Une configuration est aussi affectée à chaque cas de test. Ces cas de tests sont ensuite traduits dans un langage bas niveau, plus proche du niveau exécutable, dont la syntaxe se rapproche d'un script de test. Ce langage est TTCN-3 (*Testing and Test Control Notation version 3* [32]), un langage de scripts pour le test des communications entre systèmes embarqués temps-réel. En TTCN-3, un cas de test est vu comme une séquence d'envoi ou de réception de données entre plusieurs composants d'un système. Ces messages sont transmis grâce aux ports des composants. Un verdict est délivré pour chaque cas de test.

1.3 Introduction du génie logiciel dans les tests avioniques

Cette section est une synthèse des travaux et des contributions sur l'automatisation des tests dans le contexte avionique (sections 1.3.1 et 1.3.2). Nous verrons quelles sont les pratiques issues du génie logiciel dont l'utilisation dans un contexte avionique a déjà été étudiée. Nous considérons essentiellement celles qui introduisent des pratiques agiles dans leur cycle de développement (section 1.3.3). Les propositions étudiées sont principalement orientées pour le développement du logiciel embarqué dans les avions. Peu d'entre elles visent à améliorer le développement des équipements matériels et des calculateurs.

1.3.1 Pour le test de plates-formes IMA

La contribution de Von Aliki Ott [62] porte sur le test des plates-formes IMA qui sont des calculateurs standardisés pour l'avionique. La standardisation de ces calculateurs les rend indépendants des systèmes qu'ils contrôlent. L'utilisation de ces nouveaux calculateurs standardisés à la place des calculateurs spécifiques est possible grâce aux nouvelles générations d'architectures dites modulaires. Ce nouveau type d'architecture implique donc de

revoir le développement et les moyens de vérification et de validation des systèmes avioniques de nouvelle génération.

Ces travaux étudient deux nouvelles activités de test liées à l'utilisation des architectures IMA. Ces deux activités sont le test d'une plate-forme IMA en isolation et le test des communications dans un réseau de calculateurs IMA. L'automatisation de ces deux cas de test reprend des méthodes du génie logiciel et constitue l'enjeu de l'approche.

Aliki Ott développe un *template* générique pour une suite de tests pouvant être instancié pour de multiples configurations de modules différents. Elle montre que les tests automatiques *Hardware-in-the-Loop*, sont bien adaptés aux tests de plates-formes IMA en isolation. Elle propose aussi un algorithme permettant de générer tous les types de communication dans un réseau composé de plates-formes IMA.

Ces méthodes rendent possible la génération de code exécutable pour l'automatisation de deux cas de test étudiés. La production d'un script de test exécutable pour l'automatisation des tests d'intégration avionique est très coûteuse. Cette production demande de manipuler des langages informatiques. Ces langages ne sont pas triviaux à utiliser pour des experts en tests de systèmes avioniques. Après la production d'un test, il faut s'assurer que celui-ci s'exécute sans erreur et qu'il valide bien le comportement souhaité. Toute cette démarche prend beaucoup de temps car la production d'un script sans *bug* est très rare au premier essai. Ceci est dû à la complexité des systèmes testés et à l'éloignement entre les concepts d'un langage informatique utilisé et les besoins et habitudes des testeurs pour exprimer les cas de test. Les méthodes proposées à partir de *templates* ou d'algorithmes de génération de code facilitent la production de scripts fiables. Ces méthodes ont été développées spécifiquement pour les activités de test avionique et s'éloignent d'une utilisation des langages informatiques traditionnels. Elles permettent d'envisager une forme de réutilisation et elles augmentent la confiance dans les tests produits.

Ces travaux se concentrent sur des tests de bas niveau étant plus proches des tests en isolation des modules IMA que des tests d'intégration que nous étudions dans nos travaux. Nos objectifs sont pourtant similaires : générer du code dans un langage exécutable pour automatiser des cas de test dans un contexte avionique. Les travaux proposés ici se concentrent uniquement à automatiser deux cas de test bien identifiés et ne proposent pas de piste pour généraliser ces pratiques à d'autres domaines avioniques.

1.3.2 Par l'utilisation de l'ingénierie des modèles

Dans le cadre des tests avioniques, Guduvan et al. [44] proposent un méta-modèle composé d'éléments structuraux et comportementaux. *Eclipse Modeling Framework Ecore* [29] (EMF Ecore) a été envisagé comme langage de méta-modélisation pour produire ce méta-modèle de test. L'environnement de développement est composé d'un éditeur de modèles créé à partir de GMF (*Graphical Modeling Framework*) [30] et d'un éditeur textuel basé sur Xtext. Ce méta-modèle est couplé à un générateur de code permettant de produire du code exécutable à partir de modèles de test définis par ce méta-modèle.

Pour développer un test en utilisant ce méta-modèle, le testeur doit donc manipuler ces abstractions structurelles. Il peut créer une liste des cas de test (`TestCase`) qui seront regroupés dans une `TestSuite` ou un `TestGroup`. Ces cas de test doivent être décomposés en une suite de blocs de comportement (`Behavior`). Enfin, un bloc de comportement contient un ensemble de concepts comportementaux qui seront les instructions du test (`Statement`). La complexité est alors cachée par la méta-classe `Statement`. La structure d'un test peut-être considérée comme commune à tous les métiers de l'avionique, alors que les instructions seront spécifiques à chaque métier.

Deux grandes catégories de concepts se retrouvent dans ce méta-modèle : les concepts pour le fournisseur du moyen d'essai et les concepts pour l'utilisateur du moyen d'essai. Le fournisseur du moyen d'essai fournit un modèle pré-instancié contenant les informations sur le SUT et les types de connexion qui le composent. Ceci est réalisé par la modélisation des fichiers ICD contenant des informations sur les interfaces du système testé. Les outils externes au test, ainsi que les actions qu'ils proposent, seront aussi modélisés par les ingénieurs de test. Ces actions de test pourront ensuite être appelées dans des concepts `TestActionCallStatement` ; il s'agit d'instructions spécifiques. Les appels à ces actions se font dans des blocs de comportement qui composeront les cas de test, tout comme les instructions basiques (affectation, déclaration de variables ...) ou les instructions de contrôle du flot d'exécution (condition, répétition ...). Les instructions basiques, spécifiques et de contrôle du flot d'exécution représentent donc les abstractions comportementales du méta-modèle.

Ce méta-modèle a pour but de regrouper les meilleures pratiques que l'on retrouve dans les langages propriétaires utilisés actuellement dans le contexte avionique. Il permet de modéliser des tests avioniques mais ne masque que partiellement la complexité informatique qu'induit son utilisation. Il contient plus d'une centaine de concepts que l'utilisateur devra connaître et apprendre à organiser selon la logique du méta-modèle. Ce qui le rend complexe à utiliser.

Ce méta-modèle contient des abstractions spécifiques au contexte du test de systèmes avioniques. Il permet de produire des tests de manière homogène pour la plupart des métiers impliqués dans la réalisation d'un avion. Cependant, ce méta-modèle vise à couvrir les besoins d'un maximum de métiers qui prennent place dans la production d'un avion. Vouloir répondre à tous ces besoins amène à un méta-modèle complexe qui demande un effort d'apprentissage par les experts testeurs. Il n'a pas été envisagé de définir un méta-modèle par métier spécifique, ou de définir des profils à la manière d'UML pour cibler plus précisément les besoins. Nous pensons qu'il est préférable de proposer un langage pour chacun des domaines avioniques, dans le but de produire des procédures de test formalisées.

Proposer une syntaxe concrète se concentrant sur l'utilisabilité des notations est tout aussi important que la découverte des concepts métier constituant le méta-modèle du langage spécifique au métier ciblé. L'utilisation du *Model-Based Testing* (MBT) pour créer des procédures formalisées ne semble pas être un choix pertinent car l'utilisation de modèles au travers d'une syntaxe *via* des éditeurs textuels libres est très complexe pour des testeurs ayant très peu de connaissances en automatisation. Nous cherchons à proposer une syntaxe

concrète proche de la syntaxe utilisée actuellement dans les procédures non-formalisées. Cette syntaxe devra être supportée par des éditeurs spécifiques guidant l'utilisateur du langage dans le but de simplifier l'écriture des procédures.

1.3.3 Par l'utilisation des méthodes agiles

Les contributions visant à introduire des pratiques agiles dans le développement de systèmes avioniques sont peu nombreuses. La plupart du temps, les méthodes agiles ont pour objectif de réduire le fossé qui existe entre la conception et les tests durant le cycle de vie d'un produit. Elles sont principalement utilisées pour le développement logiciel dans l'industrie. Les contributions majoritaires concernent le logiciel embarqué dans les systèmes avioniques comme par exemple Karlesky et al. [52] ou Manhart et al. [57].

Une enquête réalisée sur 13 entreprises réparties sur huit pays et prenant en compte 35 projets de développement de logiciels embarqués [68], montre quelles sont les pratiques agiles les plus utilisées dans le monde de l'industrie. L'étude montre aussi que le taux d'adoption de tout ou partie des pratiques agile dans les entreprises développant du logiciel embarqué est très fort. Les pratiques utilisées dans ces projets viennent principalement de deux méthodes agiles : Scrum et XP [9]. Environ 60% des entreprises qui ont répondu trouvent que l'utilisation des pratiques XP leur est grandement bénéfique. Le développement du logiciel pour des systèmes embarqués implique des contraintes spécifiques. Karlesky et al. [52] mettent en avant que le test du logiciel embarqué nécessite d'émuler le matériel supportant l'exécution du logiciel.

Certains travaux visent à introduire des méthodes agiles comme Scrum uniquement pour le développement de logiciels applicatifs soumis à la certification IEC 61508 [77]. Ils proposent un processus de développement basé sur Scrum nommé Safe Scrum qui met en place un cycle itératif pour développer et certifier de manière incrémentale les fonctionnalités d'une application avionique. De la même façon que Scrum, ils utilisent un *backlog* produit pour identifier et découper les besoins du système. Dans le cas de Safe Scrum, le *backlog* est divisé en deux, un *backlog* fonctionnel et un *backlog* pour les besoins et critères de sécurité en lien avec la certification. Les éléments du *backlog* fonctionnel sont liés aux éléments du *backlog* sécurité, afin d'assurer la traçabilité des besoins en sécurité tout au long du développement d'une fonctionnalité. Cette méthode donne la possibilité de replannifier et d'avoir des retours des équipes de testeurs et des spécialistes de la certification à chaque itération. Il est donc possible de prendre en compte les corrections d'erreurs et les propositions d'amélioration le plus tôt possible dans le développement. Il permet aussi de ne pas programmer toutes les activités de validation et de certification en fin du cycle de développement. La certification est réalisée incrément par incrément, ce qui donne de meilleurs résultats.

D'autres méthodes agiles ont été testées dans un contexte avionique comme le TDD (*Test Driven Development*). Le but du TDD vise à anticiper le processus de vérification car le manque de temps et de budget oblige très souvent à se limiter aux tests les plus basiques

qui sont souvent insuffisants. Le TDD souhaite que le code des tests soit écrit avant même de produire le code réalisant les fonctionnalités attendues. E-TDD (*Embedded Test Driven Developpment*) [72] a été développé pour répondre aux problèmes de conception conjointe du matériel et du logiciel lors du développement de processeurs de signal numérique qui prennent place dans les appareils numériques de nouvelle génération. Ils ont fait évoluer un outil permettant de concevoir des tests en TDD pour des systèmes embarqués. Les systèmes embarqués étant des systèmes temps-réel, il faut donc prendre en compte les contraintes temporelles lors de l'exécution de ce type de système. L'amélioration de cet outil permet donc de mesurer le temps d'exécution de chaque instruction, ainsi que la mémoire utilisée. L'outil prend en compte les contraintes temporelles et les contraintes de ressources qu'implique l'exécution de systèmes embarqués.

Tous ces travaux montrent que l'utilisation des méthodes agiles dans un contexte avionique est possible. Ils se concentrent principalement sur l'introduction de ces pratiques dans un contexte logiciel et peu de travaux visent à les introduire plus largement, c'est-à-dire dans tout le cycle de développement d'un système complexe. Les pratiques étudiées visent à réduire le nombre de *bugs* et d'erreurs possibles principalement dans la composante logicielle d'un système. Nous ne souhaitons pas nous concentrer uniquement sur les aspects de validation du logiciel embarqué mais plus sur la validation d'un système complet et ce, principalement lors de la phase d'intégration.

1.4 Un *framework* BDD pour la conception de test d'intégration

Dans le but d'automatiser les tests dans un contexte avionique, nous souhaitons introduire des pratiques Agile lors de leur conception et de leur implémentation. Nous nous concentrons particulièrement sur les tests d'intégration car c'est à ce niveau que le manque d'automatisation est le plus flagrant. Les méthodes agiles se concentrent principalement sur le test et l'intégration continue du logiciel. Le BDD (*Behavior Driven Development*) est une de ces pratiques. Il permet de définir des scénarios de comportement pour les fonctionnalités attendues par un logiciel durant la phase de spécification. Ces scénarios seront ensuite utilisés comme tests d'acceptation lors de la validation de ces fonctionnalités. Ces méthodes favorisent la création des tests avant la phase de développement. Elles sont déjà utilisées dans le cadre de développements informatiques et nous pensons qu'elles pourraient être reprises et introduites dans un contexte avionique afin d'améliorer la communication entre les personnes impliquées dans le processus de développement. Cela permettrait aussi d'augmenter la confiance dans les systèmes produits grâce à, d'une part une meilleure couverture des tests et d'autre part, une automatisation des tests permettant un rejeu plus fréquent.

Nous avons vu, dans ce chapitre, que le développement d'un avion implique beaucoup de personnes et d'entreprises venant parfois de pays différents et que les besoins de cer-

tification complexifient le processus de développement. Il n'est pas envisageable pour le moment de remettre en cause radicalement le processus de développement qui a été mis au point et optimisé pendant des années. Nous souhaitons toutefois nous inspirer de certains concepts des tests agiles pour améliorer la conception et l'implémentation des tests d'intégration automatisés. Le BDD permet de formaliser les cas de test de chaque fonctionnalité (ou *User story* dans le monde agile) dans un langage proche du langage naturel. Cette formalisation est un bon moyen de communication entre les experts du domaine et l'équipe de développement d'un logiciel. Le but est de décrire le comportement que l'on teste de manière compréhensible par un être humain et donc que l'utilisateur final d'un logiciel puisse écrire dans un langage simple et formalisé les scénarios et résultats attendus. Notre contribution s'appuie sur la vision du BDD pour concevoir de nouveaux langages de test dans un contexte avionique.

Nous faisons ici la proposition d'un *framework* pour faciliter l'écriture des tests d'intégration et automatiser leur exécution [17] (section 1.4.1). Ce *framework* s'appuie sur une famille de langages de haut niveau, chacun spécifique à un métier, prenant place dans le processus de réalisation d'un avion (section 1.4.2).

1.4.1 Un *framework* de langages de test d'intégration

Notre proposition est un *framework* à plusieurs niveaux d'abstraction pour la conception des procédures de test d'intégration. Le but est de fournir un moyen simple de formaliser des procédures de test dans un contexte avionique. La formalisation d'un test passe par le pilotage d'un séquenceur qui permettra de stimuler le SUT par une séquence d'actions. Cette séquence d'actions sera définie grâce à un langage exécutable comme C, Python ou Lua ... Ces langages sont appelés *Domain Specific Test Languages* (DSTL). Ils reprennent les concepts des *Domain Specific Languages* (DSL) avec comme but spécifique la réalisation de tests.

Nous souhaitons produire des langages qui seront manipulables simplement par l'utilisateur. Deux types d'éditeur existent pour manipuler des langages informatiques. Les éditeurs textuels sont très répandus pour les langages de programmation classiques tel que Java, C, Python, etc. Les éditeurs projectionnels sont beaucoup moins répandus et sont plus orientés pour une utilisation dédiée à un domaine. Les procédures de test, malgré la formalisation, doivent rester un moyen de communication. Le but est de conserver un niveau d'expressivité dans le jeu d'instructions des langages de haut niveau proches du langage naturel utilisé dans les procédures non-formalisées. Pour répondre à ces deux besoins au travers de la syntaxe concrète des langages, nous avons préféré nous orienter vers des éditeurs projectionnels. Ils se veulent simples à utiliser et guident efficacement les utilisateurs dans la production de code. Ils permettent de garder une syntaxe verbeuse, afin de s'approcher de phrases en langage naturel car, à l'inverse des éditeurs textuels, ils ne demandent pas à l'utilisateur de connaître parfaitement la grammaire du langage.

Notre *framework* est composé de trois niveaux d'abstraction : les langages DSTL, le

langage pivot et les langages exécutables. Les langages manipulés par les essayeurs correspondent aux langages du niveau DSTL. Les autres niveaux sont masqués à l'utilisateur, mais sont quand même accessibles pour des besoins de *debug*. Une procédure de test réalisée avec un langage de haut niveau, i.e. DSTL, n'aura qu'une seule traduction possible dans le langage pivot. Le langage pivot quant à lui permettra de réaliser la transformation dans plusieurs langages cibles d'exécution, en fonction du type de banc par exemple. Ces mécanismes de transformation permettront de réaliser l'automatisation des procédures de test d'intégration.

1.4.2 Une famille de langages de haut niveau

La conception d'un avion implique des dizaines de métiers différents, où les besoins pour exprimer les tests sont différents. Nous pensons qu'il est important de séparer les préoccupations de chaque métier dans un langage propre.

Pour le moment, les testeurs produisent des procédures rédigées en anglais qui sont ensuite exécutées manuellement. Les procédures actuelles sont souvent stockées de manière hétérogène, parfois sous forme de documents texte, parfois sous forme de tableurs. Elles suivent des patrons de construction réguliers. Ces patrons de construction ont pour but de réduire les redondances à l'intérieur des procédures et de définir une structuration par étape du test. Le but des langages de haut niveau est d'être le plus proche possible des habitudes des testeurs. Les testeurs pourront, de plus, se concentrer principalement sur les aspects de description du comportement du système que l'on souhaite tester, et non sur les actions de test à réaliser. Plus simplement, notre objectif est de permettre aux testeurs de se concentrer sur l'intention du test plus que sur la manière de tester.

Chaque langage de haut niveau est composé d'un ensemble d'instructions spécifiques répondant aux nécessités de test d'un métier en particulier. Ces instructions doivent être compréhensibles par les experts du domaine ciblé et ont pour objectif de respecter les habitudes des testeurs. La nature de la syntaxe des langages de haut niveau pourra être différente pour chacun d'eux afin de mieux s'adapter aux habitudes des experts testeurs. Par exemple, des syntaxes tabulaires peuvent prendre place pour remplacer les procédures décrites sous la forme de tableaux Excel. Produire un langage par domaine métier nous permet de mieux prendre en compte les problématiques de chacun de ces domaines.

Les instructions de ces langages donneront la possibilité d'interagir avec les paramètres avioniques d'entrée et de vérifier l'état des paramètres de sortie. Elles permettront aussi de manipuler des outils externes au test, proposant d'espionner l'état du SUT ou de le stimuler. Elles auront pour but de masquer la complexité du code informatique et des constructions algorithmiques mises en œuvre pour réaliser l'automatisation de ces actions sur un banc de test. Tout ceci, simplifie leur utilisation par des non informaticiens ayant très peu de connaissances en algorithmique. La conséquence attendue sera la réduction du nombre d'erreurs et de *bugs* possible lors de la production des cas de test.

1.5 Conclusion

L'élaboration et la construction d'un avion impliquent un processus de développement complexe n'évoluant que très peu et impliquant par ailleurs des milliers de personnes qui doivent communiquer. Chaque partenaire a un rôle bien défini dans ce processus. Les documents de communication entre ces personnes demandent beaucoup d'efforts à leur écriture et à leur validation. Les systèmes produits par les fournisseurs sont assemblés par le constructeur au fur et à mesure de leur disponibilité. Ce processus suit un modèle en V qui commence par une conception de plus en plus détaillée des fonctionnalités que l'avion devra mettre en œuvre, pour finalement atteindre les activités de développement des équipements matériels et du logiciel les utilisant. Ensuite, vient l'intégration de ces équipements pour vérifier et valider leur comportement lors de leurs exécutions conjointes avec d'autres systèmes de l'avion. L'intégration est terminée lorsque tous les systèmes de l'avion sont intégrés.

Les architectures deviennent de plus en plus complexes de par la centralisation des unités de calcul et l'augmentation des communications inter-systèmes. Tout ceci implique une augmentation croissante de la complexité, notamment depuis l'émergence des architectures modulaires intégrées. Dans ces architectures, des fonctionnalités de systèmes différents sont implantées dans un même calculateur. Ces calculateurs sont standardisés et doivent obligatoirement gérer le partitionnement des ressources qu'ils proposent pour chacune des fonctions qu'ils hébergent.

La criticité des systèmes avioniques implique un effort accru des activités de test. Une certification des composants les plus critiques est obligatoire pour commercialiser l'avion. La batterie des tests réalisés permet quant à elle d'améliorer la sureté de fonctionnement de l'appareil dans la plupart des conditions, même les plus extrêmes.

Nous pensons que les tests, qu'ils soient formels ou non, doivent aussi être un moyen de communication entre les testeurs et le bureau d'étude et au sein même de l'équipe des testeurs. Pour rester un moyen de communication, les tests formels ne doivent pas uniquement prendre la forme d'un script de test décrit en langage informatique.

Des travaux ont été réalisés dans le but d'introduire les méthodes agiles dans un contexte avionique. Malheureusement, ces travaux sont surtout dédiés au développement du logiciel embarqué dans l'avion et non à l'élaboration d'un système complet.

Après avoir étudié le processus actuel de développement d'un avion, nous proposons un *framework* de langages reprenant plusieurs concepts de la démarche BDD et caractérisé par les deux points suivants :

- un fort niveau sémantique de la description des cas de test proche du langage naturel dans le but de décrire les comportements testés pour chaque cas de test. Nous voulons obtenir des cas de test manipulables par toutes les parties prenantes du développement (experts du domaine, experts en développement informatique, architectes, experts testeurs).

- un niveau intermédiaire d'abstraction entre le langage formel de haut niveau et le code exécutable. A chaque instruction d'un test de haut niveau correspond une ou plusieurs actions dans le langage intermédiaire et chaque instruction en langage intermédiaire établit une correspondance dans un langage exécutable sur un banc de test d'intégration. Ce niveau intermédiaire servira de niveau de *debug* d'un cas de test.

Ce *framework* permettra de formaliser les procédures de test dans un contexte avionique, sans pour autant demander aux testeurs d'acquérir des compétences en informatique. Les langages de haut niveau ne contiendront qu'un nombre très réduit d'instructions. Ces dernières auront une syntaxe proche de celle des fichiers de procédures de test qui sont actuellement encore exécutées manuellement. Comme nous l'avons vu, un grand nombre de métiers est impliqué dans la conception d'un avion. Chaque métier a des besoins particuliers. C'est pour cela que nous voulons proposer un langage de haut niveau par métier. Ces langages sont des *Domain Specific Test Languages* (DSTL).

Chapitre 2

Orchestration de langages de test métier dans une approche comportementale

Ce chapitre introduit tout d'abord le concept de langage spécifique à un domaine (*Domain Specific Language* ou DSL) et montre comment nous les utilisons pour résoudre une classe de problèmes particuliers : la formalisation des procédures de test dédiées à un type de système particulier. Dans notre contexte d'étude, la classe de problèmes que nous adressons est la conception et l'implémentation des procédures de test d'intégration. Nous avons choisi de produire des langages spécifiques pour répondre à cette problématique. Ces langages devront pouvoir être utilisés par des testeurs avioniques qui n'ont pas forcément de compétences en informatique. Ils permettent l'automatisation des tests d'intégration, tout en gardant l'intention du test dans la description des procédures. Ces procédures pourront alors servir de moyen de communication au sein de l'équipe des testeurs et entre les testeurs et les ingénieurs du bureau d'étude.

Dans un deuxième temps, ce chapitre introduit l'approche BDD (*Behavior Driven Development*), qui nous semble adaptée aux tests d'intégration. Le BDD valide des systèmes par scénarios comportementaux décrivant le comportement attendu du logiciel. Ces scénarios sont formalisés par des langages proches du langage naturel et du domaine métier considéré. Cette approche nous a guidés lors du développement des langages de test. Même si le BDD a été initialement conçu pour répondre à des problèmes de développement logiciel principalement, nous l'avons ré-appliqué au contexte d'ingénierie des systèmes avioniques, les tests d'intégration avioniques s'y prêtant bien.

Notre contribution globale est alors un *framework* orchestrant les langages de test dédiés à l'intégration de systèmes avioniques. Ce chapitre présente, tout d'abord, le concept de DSL (section 2.1) puis l'approche BDD (section 2.2). Pour implémenter les langages produits dans la thèse, nous avons choisi l'outil adéquat après une expérimentation concernant

un exemple de langage ubiquitaire en BDD (section 2.3). Puis, nous unissons les concepts de DSL et de BDD dans un *framework* orchestrant les langages de test métier dans une approche comportementale (section 2.4).

2.1 *Domain Specific Language* (DSL)

Les *Domain Specific Languages* (DSL) sont des langages informatiques dont l'objectif est de se concentrer sur un domaine particulier [37]. Un domaine est défini par le problème ou la classe de problèmes auquel il répond, par exemple la manipulation de données stockées en mémoire secondaire en faisant abstraction de la structuration de ces données d'un point de vue système. Les DSL sont opposés aux langages de programmation classiques ou généralistes (*General-Purpose programming Language* ou GPL) tels que Java, C ou encore Python. Parmis les DSL les plus connus citons par exemple SQL initialement nommé SEQUEL (*Structured English Query Language*) pour interroger et modifier une base de données relationnelle, HTML pour la structure de documents contenant des liens hypertextes, Make pour la compilation de fichiers sources ... Par la suite, l'utilisation du sigle GPL fera référence aux langages de programmation classiques (ou généralistes) alors que le sigle DSL désignera les langages dédiés à un domaine.

Le concept de DSL émerge au cours des années 1980 [59]. A cette période, le besoin de prendre en compte le domaine métier grandit et a fait naître de nombreux langages de domaines. Cette appellation peut être rattachée à la notion de petit langage proposé par Jon Bentley [10]. Il montre que plus les langages sont focalisés sur un domaine et donc restreints syntaxiquement, plus il est facile de les concevoir, de les implémenter, de les maintenir, de les documenter ; de plus, ils sont plus faciles à apprendre et à utiliser. Deursen et al. ont réalisé une bibliographie annotée sur les DSL et établissent une définition du concept comme suit [83] :

> « Un langage dédié à un domaine est un langage de programmation ou un langage de spécification exécutable qui offre, *via* des notations et des abstractions, un pouvoir d'expressivité, habituellement restreint, se concentrant sur les problèmes d'un domaine particulier. »

Une autre définition dans la littérature est celle de Martin Fowler [37]. Selon lui, un DSL est défini par quatre caractéristiques :

1. **Langage de programmation informatique :** Tout comme un GPL, un DSL est conçu pour être utilisé par des humains pour fournir des instructions exécutables à un ordinateur. Son but est de faciliter une tâche de codage.
2. **La nature du langage :** Tout comme un GPL, un DSL est conçu pour produire des programmes informatiques combinant un ensemble d'expressions. Alors qu'un GPL propose des opérations génériques de programmation pour combiner des expressions, un DSL possède des mécanismes de combinaisons d'expressions compréhensibles pour un expert du domaine.

3. **Expressivité limitée :** Contrairement à un GPL, un DSL est doté d'une expressivité limitée couvrant un domaine particulier. Cette contrainte ne permet pas de réutiliser facilement un DSL pour un autre domaine métier. Par contre, elle permet à un expert métier de répondre à un problème sans notion de programmation superflue, le minimum nécessaire étant couvert par les possibilités du DSL.
4. **Dédié à un domaine :** Un DSL est utile s'il est orienté sur un domaine réduit. Plus le domaine est réduit, plus l'utilisation d'un langage dédié aura de la valeur.

Le but de ces langages est de réduire le fossé qui existe entre les développeurs et les experts d'un domaine en particulier. Dans certains cas, il est préférable pour des tâches particulières de codage de laisser la main aux experts. Il s'agit de produire de manière plus concise et avec une plus grande confiance, les programmes souhaités pour répondre aux problèmes du domaine. L'implication des experts dans les activités de développement aura pour conséquence d'améliorer la communication entre les experts et les développeurs. L'objectif final est de produire des applications répondant aux attentes et aux besoins des experts.

Après avoir précisé plus en détail les caractéristiques des DSL, nous montrons les bénéfices attendus de l'utilisation de tels langages et pointons les différences entre DSL et GPL. Les DSL étant conçus pour être utilisés à différentes étapes d'un processus de développement, nous présentons les différentes catégories de DSL ayant leur utilité à certains moments clés d'un processus de développement. Nous finissons cette présentation par des considérations sur l'implémentation de tels langages et proposons leur utilisation dédiée aux tests d'intégration de systèmes complexes avioniques.

2.1.1 Caractéristiques d'un DSL

La bonne conception d'un DSL demande d'étudier plusieurs de ses caractéristiques comme l'expressivité, la complétude, la couverture du domaine, la sémantique, la séparation des préoccupations, la modularité des langages ou encore la syntaxe concrète [89] :

Expressivité : Le but premier d'un DSL est d'augmenter l'expressivité que fournissent les langages classiques. Ceci est réalisé grâce à l'intégration dans le langage d'expressions grammaticales reflétant l'intention du métier et contenant la connaissance du domaine. Une expression dans un DSL est traduite en plusieurs expressions dans un GPL ; un DSL utilise moins de code qu'un GPL pour exprimer les mêmes besoins d'un domaine métier.

Complétude : La complétude est le degré avec lequel un langage peut exprimer un ensemble d'actions contenant toutes les informations nécessaires pour qu'elles soient exécutées. Un langage incomplet implique l'ajout d'informations supplémentaires, comme des fichiers de configuration ou du code développé *via* un langage de plus bas niveau.

Couverture du domaine : Dans le cas idéal, un langage répondant aux problématiques d'un domaine couvrira entièrement ce domaine si tous les programmes élaborés

pour ce domaine peuvent être spécifiés dans ce langage. Dans la plupart des cas, les DSL sont généralement définis pour répondre uniquement à un sous-ensemble des problèmes d'un domaine particulier afin d'obtenir un langage ne comportant pas de concepts algorithmiques ou informatiques dépassant les compétences d'un expert du domaine.

Sémantiques et exécution : La sémantique statique est en charge de vérifier les contraintes et les règles de typage portant sur les expressions du langage. Les programmes respectant ces règles peuvent alors être exécutés en minimisant le nombre d'erreurs possibles à l'exécution. La sémantique opérationnelle désigne le comportement attendu des expressions du langage. L'exécution d'un programme est rendue possible par son interprétation ou par sa transformation en code exécutable. De manière générale, la sémantique opérationnelle d'un DSL s'obtient en explicitant les règles de traduction dans un langage GPL, lui-même pourvu de sa propre sémantique.

Séparation des préoccupations : Un domaine peut regrouper plusieurs préoccupations. Soit le DSL prendra en compte toutes les préoccupations d'un domaine, soit ces préoccupations peuvent elles mêmes être séparées en plusieurs sous-langages. Ceci donne lieu à un développement où chaque fragment du programme répondra à une préoccupation spécifique. Cette séparation peut s'avérer utile pour des besoins de réutilisation, lorsque certaines de ces préoccupations sont transversales à plusieurs domaines.

Modularité des langages : La modularité des langages a pour objectif de simplifier la conception de nouveaux langages comme c'est le cas pour le développement logiciel lorsque une décomposition fonctionnelle est utilisée afin de ne pas se répéter [86]. La combinaison de différents DSL peut alors se faire selon quatre façons : réutiliser, référencer, embarquer ou étendre. Ces quatre façons de composer les langages seront plus amplement détaillées au chapitre 4.

Syntaxe concrète : L'utilisabilité de la syntaxe concrète est importante pour faire en sorte que le langage soit accepté par la communauté des experts du domaine sans qu'ils soient experts en programmation. Le but de cette syntaxe est de fournir une notation simple et concise pour exprimer les préoccupations d'un domaine. Elle a aussi pour objectif de simplifier la lecture du code. La syntaxe concrète peut prendre plusieurs formes : textuelle, graphique, symbolique, tabulaire ou matricielle. Il convient donc choisir la forme convenant le mieux aux besoins des experts du domaine.

2.1.2 Bénéfices attendus

Le principal avantage des DSL est d'améliorer la productivité d'une partie ou de l'ensemble du code exécutable réalisant une tâche précise. Cette productivité accrue est la résultante des autres avantages apportés par les DSL détaillés ci-dessous [37] [89].

La conception d'un langage dédié à un domaine permet d'augmenter la connaissance

sur ce domaine. L'étude préalable nécessaire à la conception permet de clarifier les concepts étant rattachés à ce domaine tout en permettant aux utilisateurs métier de s'accorder sur une vision commune du domaine ciblé. De cette étude découleront les concepts métier du domaine et les liens entre ces concepts. Cette étude peut prendre la forme de cartes conceptuelles ou mieux encore d'ontologies décrites dans des IDE spécialisés. La conception d'un DSL permet donc de décrire et de rendre opérationnelle la connaissance d'un domaine.

Un DSL garantit que les utilisateurs finaux du logiciel ne se concentrent que sur les problèmes liés à leur domaine. Les utilisateurs ont à faire correspondre du mieux possible la logique du domaine métier avec la logique de l'application développée. La production de code avec un GPL est réservée à des programmeurs et non à des experts métier. L'utilisation d'un DSL masque la complexité induite par du code écrit en GPL. Les utilisateurs développant tout ou partie d'une application avec un DSL répondant à leurs besoins n'auront plus à se concentrer sur les problèmes d'implémentation et sur les contraintes liées à l'utilisation d'un langage de programmation classique.

L'utilisation d'un DSL facilite l'implication des experts du domaine car ils peuvent lire et comprendre le code produit. Le code DSL décrit explicitement les comportements souhaités grâce au vocabulaire du domaine métier. L'utilisation de DSL améliore la communication entre les experts du domaine et le reste de l'équipe de développement. L'utilisation des DSL comme moyen de communication est une caractéristique particulièrement recherchée pour les DSL dits métiers qui sont principalement des langages *stand-alone*. Ces derniers sont conçus pour être utilisés indépendamment. Les abstractions et les notations d'un DSL sont alignées avec le domaine ciblé rendant les programmes DSL plus concis et plus lisibles que leurs équivalents en GPL.

Enfin, l'expressivité d'un DSL étant concentrée sur un métier, les possibilités fournies sont plus limitées que celles fournies par les GPL. Puisque les DSL sont moins permissifs, le nombre d'erreurs qu'il est possible de faire dans un programme est alors réduit. Ces erreurs seront aussi plus facilement identifiables et corrigées. Ceci permet d'augmenter la confiance dans le code exécutable produit en réduisant le nombre de *bugs* possibles.

Ces langages de programmation dédiés à un domaine peuvent, comme les GPL, être manipulés par des environnements d'édition intégré (IDE) pour faciliter leur conception et leur utilisation. Cependant, sans environnement de développement spécifique, concevoir un DSL peut s'avérer aussi fastidieux que de concevoir un GPL avec des analyseurs lexicaux et syntaxiques. Heureusement des outils et *frameworks* existent pour développer et utiliser des DSL.

2.1.3 Frontière avec les langages de programmation classiques

La limite entre les langages GPL et DSL est assez floue. Cette limite dépend des définitions existantes dans la littérature et du contexte d'utilisation du langage.

Un GPL [23] est un langage utilisé pour résoudre une large variété de problèmes, contrairement aux langages spécifiques à un domaine. Les langages de programmation courants

(C, C++, Java, COBOL, etc.) en sont des exemples. Un GPL est dit turing-complet, lorsqu'il peut répondre à n'importe quel type de problème informatique pouvant être résolu par une machine de Turing. De ce fait, les GPL sont interchangeables en théorie. Cependant, l'implémentation de certaines fonctionnalités peut varier d'un langage généraliste à l'autre. Par exemple, les fonctionnalités de gestion de la mémoire en C permettent une meilleure intégration des logiciels embarqués sur des équipements matériels n'ayant pas beaucoup de ressources. Ces fonctionnalités ne sont pas accessibles en Java, où la gestion de la mémoire s'appuie sur un algorithme de ramasse-miettes (*garbage collector*). L'évolution des langages informatiques a aussi été induite par la création de nouveaux paradigmes de programmation : programmation orientée objets, programmation impérative, programmation fonctionnelle, programmation en logique, programmation orientée aspects, programmation par contrats ou encore des formalismes de méta-programmation que nous verrons par la suite (programmation réflexive).

Les DSL sont quant à eux très rarement turing-complets. Ils sont conçus pour répondre à un certain nombre de problèmes contrairement aux GPL. Dans de nombreux cas, les DSL sont conçus dans l'optique de répondre à une classe de problèmes spécifiques. Au cours du temps, les multiples demandes d'évolution peuvent les transformer en langages plus vastes, parfois turing-complets et sont de fait des GPL.

Les DSL ont pour objectif de simplifier le développement des applications. Un DSL ne permet généralement pas l'utilisation de concepts algorithmiques tels que les boucles, les structures conditionnelles et ne permet pas non plus de construire de nouvelles abstractions. De même, il ne permet pas facilement de gérer une table des variables du programme ou la création de sous-programmes ou de fonctions.

Dans les aspects qui rentrent en jeu dans la caractérisation d'un DSL selon Martin Fowler [37], les principales caractéristiques pour matérialiser la frontière entre un DSL et un GPL sont l'expressivité et la nature du langage. Cependant, ces caractéristiques dépendent du domaine d'application considéré. Il est même possible de considérer un DSL comme un GPL lorsqu'il répond à d'autres problèmes que celui pour lequel il a été conçu. Par exemple, le langage XSLT est dédié à la transformation de documents XML. Cependant il possède toutes les fonctionnalités attendues par un GPL et le problème des huit reines par exemple est programmable en XSLT. Cependant, il est généralement considéré comme un DSL parce qu'il est principalement utilisé pour transformer des documents XML. Utilisé pour une autre catégorie de problèmes, il peut alors être considéré comme un GPL.

2.1.4 Domaines d'application

Les DSL peuvent prendre place sous plusieurs formes dans un processus de développement d'un logiciel. Pour chacune de ces formes, les DSL répondent à un besoin particulier. Ils peuvent être utilitaires, architecturaux, techniques ou encore métiers ...

Les DSL utilitaires ont pour but de faciliter le travail des développeurs pour une tâche bien précise. Le but n'est pas de recouvrir les besoins de l'ensemble du processus de déve-

loppement d'un logiciel, mais de créer un DSL spécifique à la réalisation d'une tâche bien identifiée pour simplifier son développement. Par exemple, le langage Make est un langage déclaratif aidant à la construction de programmes exécutables. Il permet, entre autres, à un programme d'être automatiquement compilé en fonction d'options spécifiques à différentes plates-formes d'exécution.

Les DSL architecturaux fournissent des concepts nécessaires à la mise en place d'une architecture logicielle donnée. Ces langages ne sont pas des langages de modélisation généralistes comme UML [14] qui eux permettent de modéliser n'importe quel type d'architecture. Selon Markus Voelter, UML est un langage de modélisation généraliste comparé aux DSML (*Domain Specific Modeling Language*) [56]. Tout comme en UML, un programme écrit à partir d'un DSL architectural génère le squelette de code d'une partie de l'application. Le programmeur devra insérer du code dans ce squelette. Il permet de structurer l'application. Par contre, le squelette de code généré a pour but de faire en sorte que les développeurs respectent une architecture particulière (architecture en couches, architecture orientée objets, ... [40]). Ces DSL sont très utiles pour développer des systèmes dont l'architecture est multi-tiers (architectures Modèle-Vue-Contrôleur) ou pour la mise en place de patrons de conception par exemple. AUTOSAR (*AUTomotive Open System ARchitecture*) est un DSL architectural proposant de créer des modules logiciels dont l'architecture est standardisée pour l'automobile. L'architecture en couches des modules développés avec AUTOSAR les rend utilisables par des véhicules de différents constructeurs et dans des composants de différents équipementiers [38].

Les DSL techniques sont conçus pour être utilisés par des programmeurs. Ils permettent de traiter la totalité de la logique d'une l'application tout comme un GPL, mais en proposant des expressions permettant d'accélérer la production de code. Ils complémentent en général un GPL existant. Par exemple, Ruby est un GPL permettant d'écrire efficacement des applications à objets ; Ruby on Rails est un DSL technique construit à partir de Ruby permettant de rendre transparente l'utilisation du patron d'architecture Modèle-Vue-Contrôleur (MVC).

Les DSL métier quant à eux ont pour but de construire des applications pour un métier très spécifique. La logique métier et les aspects du domaine métier seront décrits grâce à ce type de DSL. Le but est que ces langages soient utilisés par les experts du domaine. L'implication des experts dans le développement d'un logiciel est un avantage qui permet principalement de faire en sorte que l'application produite réponde bien aux besoins des ces experts. Pour autant, le fait que les langages métier ne soient pas utilisés par des informaticiens, mais bien par des experts, demande des efforts particuliers pour faire en sorte qu'ils soient conformes aux attentes et aux pratiques des experts métier. Ces efforts devront rendre l'utilisation de ces langages triviale pour des personnes ayant principalement des connaissances du domaine ciblé. Le langage nommé Quant DSL est un exemple de DSL métier conçu pour des besoins liés à l'analyse quantitative dans le milieu de la finance [18]. Nous pouvons aussi citer les travaux de Peyton Jones et al. proposant un langage de formalisation des contrats d'assurance ou d'instruments financiers [63].

2.1.5 Implémentation d'un DSL

Il existe deux types de DSL : les DSL internes et les DSL externes. Un langage interne possède une syntaxe qui se base sur un langage hôte. Un langage externe, quant à lui, propose de créer des programmes à partir d'une syntaxe qui lui est propre. L'implémentation d'un DSL interne s'appuie sur un langage hôte ayant des capacités d'extension. L'implémentation d'un DSL externe nécessite soit d'utiliser les outils à disposition pour la création de langages soit d'utiliser des ateliers de développement de DSL.

Le choix d'implémenter un langage interne ou externe n'est pas forcément guidé par des besoins ou des contraintes liés à son développement ou à son utilisation. Souvent, ces choix dépendent des habitudes, coutumes et connaissances de chaque équipe de développement [37].

2.1.5.1 Implémentation d'un DSL interne

Les DSL internes sont construits à partir d'un langage hôte GPL. On réutilise alors la syntaxe, la structure et les possibilités d'extensibilité de ce langage. La difficulté est que ce type de DSL aura alors une syntaxe contrainte par la syntaxe du langage dans lequel il est embarqué. Les expressions du domaine métier ne se retrouveront pas telles quelles dans un programme écrit dans ce type de DSL.

Le premier niveau de réutilisation est de ne réutiliser que les constructions de base du langage hôte. L'implémentation d'un DSL interne est alors proche du développement d'une API dédiée à un domaine. Un exemple classique d'extension de la syntaxe d'un langage hôte est celui de l'utilisation d'un patron *builder* pour construire des objets complexes. Le constructeur de calendrier est un objet dédié permettant de simplifier la syntaxe utilisée pour créer un évènement du calendrier (listing 2.2). Les listings 2.1 et 2.2 montrent le changement de syntaxe induit par le chaînage d'appels de méthodes grâce au constructeur d'expressions (`CalendarBuilder`).

Listing 2.1 – Utilisation classique d'objets pour l'ajout d'évènements dans un calendrier

```
Calendar cal = new Calendar();
Event event = new Event("DSL workshop");
event.on(2017, 05, 22);
event.from("9:00");
event.to("12:00");
cal.add(event);
```

Listing 2.2 – Utilisation de classe `CalendarBuilder` avec chaînage des méthodes

```
CalendarBuilder calBuilder = new CalendarBuilder();
calBuilder.add('DSLworkshop').on(2017,05,22).from('9:00').to('12:00');
```

Si de nouvelles constructions syntaxiques sont nécessaires à l'automatisation de certaines tâches non supportées par le langage hôte, celui-ci doit supporter des fonctionnalités

d'extensibilité : constructeurs d'expressions, macros, fermetures (*closures*), annotations, etc., souvent à base de méta-programmation. La méta-programmation donne la capacité à un programme de manipuler d'autres programmes dans le même langage. Les langages les plus populaires pour la mise en place de langages internes sont Lisp [55], Ruby [25], ou encore Scala [22]. Ces langages sont principalement des langages interprétés. Il est aussi possible de faire de la méta-programmation *via* des langages compilés, même si cela demande plus de travail.

Ces méthodes permettent d'ajouter de nouveaux mots-clés au langage hôte. Ce changement de syntaxe a pour but d'augmenter l'expressivité du DSL par rapport à l'utilisation du langage hôte en introduisant des éléments du vocabulaire métier dans sa syntaxe. Le code DSL sera alors plus lisible. Par exemple, la version 3 de JUnit utilise l'introspection fournie par le langage Java pour exécuter les méthodes de test d'un *test case* étendant la classe `junit.framework.TestCase`. Dans ce cas de figure, chaque méthode doit commencer par le mot `test`. Depuis la version 4, JUnit utilise les annotations pour désigner de façon plus explicite les méthodes de test à exécuter (`@Test`). Il en est de même pour identifier les méthodes devant être exécutées avant et après chaque test (`@Before` et `@After`). L'ajout de ces éléments syntaxiques permet à JUnit d'être plus facile à prendre en main pour des développeurs ayant en charge d'écrire des tests unitaires. Les tests sont exécutés automatiquement et entrelacés avec des méthodes permettant d'isoler les tests. Un rapport de test est généré à chaque exécution d'un ou plusieurs cas de test.

Les avantages des DSL internes sont la facilité et la rapidité de la programmation de tels langages, mais puisqu'ils utilisent les fonctionnalités d'un GPL, ils sont d'un usage réservé à des développeurs. Par exemple, les messages d'erreurs ne seront pas spécifiques au DSL car produits par le compilateur ou l'interprète du langage hôte.

2.1.5.2 Implémentation d'un DSL externe par des outils dédiés

La syntaxe d'un langage externe est indépendante du langage avec lequel les applications sont exécutées. Un programme décrit dans ce langage devra alors ensuite être compilé vers un langage GPL. Classiquement, l'implémentation nécessite le développement d'un analyseur lexical et d'un analyseur syntaxique [1]. Cette étape préliminaire crée un arbre de syntaxe abstraite à partir d'un programme qui sera consommé par le générateur de code.

La conception d'un langage externe se base sur une syntaxe spécifique qui sera définie dans un méta-langage syntaxique, le plus connu de ces langages étant Backus-Naur Form (BNF) ou sa nouvelle version EBNF (*Extended Backus-Naur Form*) [70]. Les syntaxes produites à partir de ces langages peuvent être consommées par plusieurs outils de génération d'analyseurs comme Bison [42], Lex et Yacc ou ANTLR [80] pour les plus connus. L'analyse lexicale est la première phase de compilation d'un programme. Cette phase réalise la reconnaissance des lexèmes du langage grâce à des expressions régulières. Cette suite d'entités lexicales sera ensuite transmise dans une deuxième phase à l'analyseur syntaxique. Ce dernier, quant à lui, organisera cette liste de lexèmes pour produire un arbre syntaxique

abstrait (*Abtract Syntax Tree* - AST). Une troisième phase vérifie que l'arbre représentant le programme DSL est sémantiquement correct, c'est-à-dire que la structure et les entités qui le composent font sens au regard de la sémantique du domaine. Enfin, il est possible pour le générateur de produire du code dans un langage GPL, le code produit étant lui-même transformé en code exécutable par les outils du langage GPL.

Des outils dédiés à l'implémentation de langages permettent d'intégrer toutes les étapes de développement d'un DSL textuel au sein d'un IDE. Ils intègrent la génération d'analyseurs lexicaux et syntaxiques, des outils pour aider à la génération de code GPL et des éditeurs textuels supportant l'édition d'un programme en DSL. Ces éditeurs proposent la coloration syntaxique, l'auto-complétion, la gestion des erreurs syntaxiques, des opérations de *refactoring*, voire même des fonctions de *debug*. Le plus populaire de ces outils est Xtext, il s'intègre avec les IDE Eclipse et IntelliJ grâce à des *plugins*.

2.1.5.3 Implémentation d'un DSL externe par des ateliers dédiés

De nouveaux outils nommés ateliers de conception de langages [36] sont développés pour supporter le développement de nouveaux langages de programmation et de nouveaux langages dédiés. Ce sont des outils de méta-programmation. Ils reprennent les concepts issus du *Language Oriented Programming* [91]. Martin Fowler propose alors un troisième type de DSL : les DSL construits à partir d'un atelier de conception de langages [36] (ou environnement de développement de langages). Si ces ateliers s'apparentent à des IDE classiques comme Eclipse ou InteliJ, ils ne servent pas uniquement à définir la structure d'un langage dédié mais aident aussi au développement des environnements d'édition adaptés et personnalisés de programmes DSL.

La principale différence entre un *framework* pour le développement de langages et un atelier de conception de langages est le moyen d'édition d'un programme DSL. Dans le premier cas, l'utilisateur produit du code *via* un éditeur de texte classique. Le code sous forme textuelle devra ensuite être analysé pour être transformé en AST. Dans l'autre cas, des éditeurs projectionnels servent à l'édition des programmes [90]. L'éditeur projectionnel rattaché à un concept donnera la structure d'agencement des mots-clés de la syntaxe concrète et des sous-concepts qui le composent. Les sous-concepts auront, eux aussi, chacun leurs propres éditeurs. Les concepts et sous-concepts d'un langage forment la hiérarchie des concepts du méta-modèle permettant de construire les AST des programmes. L'AST est donc l'élément central des ateliers de développement de langages. Les phases d'analyses lexicales et syntaxiques sont alors grandement simplifiées.

Grâce aux éditeurs projectionnels, l'utilisateur manipule une représentation de l'AST du programme développé. Des règles de projection affichent ces concepts selon la syntaxe concrète définie par les éditeurs. L'utilisateur interagit avec cette syntaxe concrète et les changements se répercutent directement sur les AST. Un programme développé avec un langage conçu dans un atelier ne peut être édité en dehors de l'environnement d'édition projectionnel proposé par l'atelier. Tandis que la création de code DSL dans un *framework*

classique comme Xtext [47] est supportée par n'importe quel éditeur textuel.

Dans une vision projectionnelle, il est possible qu'un concept de l'AST ait plusieurs représentations, chacune d'entre elles rattachée à un éditeur. Il est envisageable de représenter un même programme sous forme textuelle dans certains cas et sous forme graphique dans d'autres. Les modifications des informations contenues dans un éditeur auront un impact direct sur les informations stockées dans le modèle sémantique sous la forme d'un AST.

Martin Folwer propose une première définition de ce que sont les ateliers de conception de langages [36] :

- Les utilisateurs peuvent créer librement de nouveaux langages qui peuvent s'intégrer les uns avec les autres.
- Les programmes créés sont directement persistés sous forme d'arbres de syntaxe abstraite.
- Les développeurs de langages définissent un DSL en trois parties : le schéma (ou le méta-modèle), les éditeurs et les générateurs.
- Les utilisateurs du langage manipulent les concepts du DSL *via* des éditeurs projectionnels.
- Un atelier de conception de langages peut faire persister des informations incomplètes ou contradictoires dans sa représentation abstraite.

L'utilisation de l'approche *Language Oriented Programming* [91] implique que le développement d'un DSL soit découpé en deux étapes. La première vise à réaliser le développement de la syntaxe et des éditeurs. La deuxième s'intéresse à la transformation d'un programme en DSL vers un programme en GPL. Le principal inconvénient est qu'il n'existe pas de standard pour définir les schémas, éditeurs et générateurs. Il n'existe pas non plus de standard pour migrer un langage d'un atelier de conception à un autre. Le choix du bon atelier de méta-programmation est alors primordial pour la réussite du développement d'un DSL.

Les notations proposées par les langages GPL sont génériques et leur composition rend complexe leur apprentissage par des non-informaticiens. Dans ces ateliers, tout programme est guidé par l'assemblage des éditeurs projectionnels correspondant aux concepts métier hiérarchisés dans une méta-syntaxe. Les utilisateurs finaux n'ayant aucune syntaxe ou notation particulière à apprendre, nous pensons diminuer l'effort cognitif nécessaire à l'appréhension d'un tel langage [13]. La contrepartie est un guidage excessif que les éditeurs projectionnels imposent aux utilisateurs finaux [88]. Par exemple, l'utilisation de la touche retour chariot (*enter*) aura un comportement différent en fonction de la position du curseur dans le programme représenté par un AST. Son utilisation dans une liste de concepts réalisera l'instanciation d'un nouveau concept. Il ne sera donc pas possible de réaliser des sauts de lignes grâce à cette touche à ce moment-là dans l'arbre syntaxique.

Deux techniques sont principalement rencontrées pour la génération de code : la génération par *template* ou la génération par conversion. La génération par *template* se concentre sur la forme du code produit par le générateur et fournit des mécanismes pour rempla-

cer des parties du *template* par des éléments de l'AST du programme en DSL. C'est bien plus qu'une transformation *model to text* en ingénierie dirigée par les modèles. En effet le *template* est lui-même vérifié par l'analyse statique. De son coté, la génération par conversion parcourt les arbres de syntaxe abstraite à la manière d'un compilateur pour lui faire correspondre le code cible GPL.

2.1.6 Apport des DSL aux tests d'intégration avionique

Le développement d'un avion implique une centaine de domaines différents (ATA) ayant des préoccupations, des vocabulaires et des langages d'action différents. Nous souhaitons introduire des langages répondant le mieux possible aux préoccupations de chacun de ces métiers. Dans un premier temps, ces langages formaliseront les procédures de test d'intégration des différents systèmes avioniques. Ils constituent une famille de langages de test répondant chacun à un domaine avionique spécifique. Ces DSL sont classés dans la catégorie des DSL métier.

L'analyse préalable à l'implémentation d'un langage spécifique à un domaine demande une implication de tous les acteurs susceptibles d'être confrontés aux procédures produites grâce à ces langages. L'effort produit lors de l'analyse du domaine permet aux différents acteurs de s'accorder sur une vision consensuelle du domaine. Ainsi, les procédures de test formalisées seront plus facilement lisibles et interprétables par l'ensemble de l'équipe chargée des tests.

Lors de la phase d'intégration, les procédures de test s'exécutent sur un banc de test pour s'interfacer avec le matériel sous test et pour simuler les modèles environnementaux. Les instructions d'une procédure de test sont exécutées par le banc de test *via* un séquenceur temps réel qui lui est propre. En général, un séquenceur est pilotable par un langage de programmation. Les procédures produites lors de la phase d'intégration devraient être exécutées sur plusieurs types de banc de test. C'est pourquoi nous avons proposé un langage intermédiaire pour traduire les procédures de test et les rendre indépendantes du langage de programmation utilisé par un banc. Ce langage intermédiaire est un DSL technique dédié aux programmeurs de test devant traduire un nouveau langage de test métier en instructions du langage intermédiaire. Ils peuvent de même prendre en compte un nouveau langage supporté par un nouveau banc de test.

Produire un DSL pour un domaine spécifique procure un certain nombre d'avantages [89] :

- Un DSL permet d'exprimer plus facilement des préoccupations métier. Dans notre cas les DSL métier permettent aux testeurs métier de formaliser rapidement leurs procédures de test.
- Un DSL cache la complexité de l'utilisation d'un GPL.
- Un DSL est utilisé par des experts métier n'ayant pas forcément de connaissance en programmation.
- Un DSL peut générer beaucoup de lignes de code GPL depuis peu de lignes en DSL. Par exemple pour une procédure de test décrite par 16 lignes de code DSL concernant

le métier de régulation de l'air dans un avion, nous obtenons plus de 300 lignes de code GPL Python et plus de 1000 lignes de code en State Chart XML.
- Un DSL améliore la communication dans l'équipe et consolide la connaissance commune d'un domaine.
- Un DSL est utilisé par les utilisateurs finaux au travers d'un IDE. Dans notre cas, les utilisateurs sont guidés par les éditeurs projectionnels.

Notre proposition concernant l'apport des DSL pour la validation de l'intégration des systèmes avioniques a été présentée à un *workshop* dédié au développement agile pour les systèmes critiques [16].

2.2 *Behavior Driven Development* (BDD)

Le Behavior Driven Development (BDD) [60] est une pratique agile inventée en 2003 pour répondre à la difficulté de mettre en place une approche *Test First* de bout en bout (*Test Driven Development* [8]). Le *Test Driven Development* est une bonne pratique agile venant d'*eXtreme Programming* [9]. Kent Beck propose de l'utiliser de façon intensive jusqu'à spécifier par des tests avant de coder.

Si l'idée parait séduisante au départ, elle est très difficile à mettre en œuvre. Nous introduisons le BDD comme une alternative au TDD, puis nous explicitons l'approche BDD et présentons les outils et *frameworks* associés.

2.2.1 Le BDD, une alternative au Test Driven Development

La principale difficulté lors de l'utilisation du TDD concerne le manque de règles claires sur ce qu'il faut tester ou non. Il est possible d'obtenir un code passant tous les tests dans une approche TDD et ne passant pas les tests d'acceptation pour la fonctionnalité à développer. Un autre difficulté concerne la granularité des tests unitaires. Faut-il faire des tests unitaires pour une méthode, un composant, une fonctionnalité ? Pour résoudre ces problèmes, le BDD propose de faire en sorte que les cas de test découlent des critères d'acceptation d'une histoire utilisateur (*user story*).

Un logiciel développé grâce aux méthodes Agile est découpé en *user stories*. Ces *user stories* décrivent les besoins fonctionnels du produit d'un point de vue utilisateur. Le BDD propose de faire correspondre à chaque *user story* un ensemble de scénarios illustrant les cas d'utilisation de la fonctionnalité qu'elle décrit. Ces scénarios sont ensuite transformés en tests exécutables. Il y a donc continuité entre les besoins et exigences d'un produit et les cas de test les validant. De ce fait, le BDD permet une traçabilité entre une exigence fonctionnelle et les cas de test qui la couvrent.

Dans l'approche BDD, un scénario de test utilise le vocabulaire de la description d'une *user story*. Les noms choisis pour les scénarios de test doivent s'apparenter à des phrases complètes. Selon Dan North [60], la phrase décrivant un scénario de test devrait commencer par le mot *should* (doit) pour aider son concepteur à se concentrer sur l'intention du

scénario. De plus, les interfaces de développement se focalisent uniquement sur les tests à exécuter et peuvent elles aussi s'adapter au domaine métier. Ainsi, la compréhension des tests et du code vérifiant les tests est simplifiée pour toutes les personnes impliquées dans le développement de l'application.

Le BDD est un sous-ensemble des pratiques ATDD qui sont aussi nommées *Story Test Driven Development* (SDD) ou spécification par l'exemple. Ces méthodes ont été popularisées à partir de 2004 grâce aux outils Fit et FitNesse [1], malgré les objections de Kent Beck en 2003 dans *Test Driven Development : By example* qui avait jugé ces méthodes inexploitables [8]. De fait, le TDD n'est qu'une des pratiques proposées par l'*eXtreme Programming*. La pratique du TDD est dédiée aux développeurs, tandis que la pratique du BDD implique toute l'équipe de développement, y compris le client.

2.2.2 L'approche BDD

Nous avons vu que la méthode BDD utilise les *user stories* comme point de départ lors de la spécification fonctionnelle d'un produit [21]. L'approche BDD promue par Dan North s'est inspirée du *Domain Driven Design* et de son concept de langage ubiquitaire [34] qui tend à unifier un langage commun entre experts métier et développeurs. Par son approche, le BDD accélère l'émergence d'un langage omniprésent. De plus, l'approche BDD se base sur un langage ubiquitaire dédié à l'analyse des besoins. Elle apporte deux patrons génériques pour définir une *user story* et les scénarios correspondants quel que soit le domaine métier. Le premier patron se focalise sur la fonctionnalité en elle-même représentée par une *user story*. Elle est supportée par la syntaxe suivante [61] :

```
Title: [one line describing the story]
       As a [role]
       I want [feature]
       So that [benefit]
```

Cette syntaxe identifie l'utilisateur principal de la fonctionnalité ainsi que les bénéfices attendus. Interconnecter l'utilisateur et les bénéfices d'une *user story* permet d'expliciter pour qui est faite la fonctionnalité et pourquoi l'utilisateur en a besoin.

Une des caractéristiques principales du BDD [75] est de se concentrer sur la description du comportement souhaité d'une fonctionnalité ou d'un système, par des scénarios de test qui automatisent la validation de chacune des *user stories* (critères d'acceptation exécutables) et permettent de déterminer la fin du développement. Pour cela, le BDD propose un deuxième patron qui divise un scénario de test rattaché à une *user story* en trois parties. Cette structuration est implantée par un langage appelé Gherkin. Elle est supportée par la syntaxe suivante :

```
Scenario : Title
```

1. `http://fitnesse.org/`

```
Given [context]
        And [some more context]...
When  [event]
Then  [outcome]
        And [another outcome]...
```

Chaque scénario est composé d'un nom (`Title`) décrivant une situation de test particulière. Le comportement est décrit par trois étapes suivant le patron Acteur-Action-Assertion (AAA) des test unitaires en *eXtreme Programming*. La première étape désigne le contexte de test correspondant à la mise en condition du système à tester (`Given`). L'étape suivante décrit le ou les évènements à traiter et décrit ainsi la fonctionnalité attendue (`When`). La troisième étape est composée d'assertions permettant de rendre le test exécutable, voire automatisable (`Then`). Cependant, Dan North considère que cette "unitarisation" des tests ne correspond pas à tout type de système à tester, en particulier pour les sites Web et pour les systèmes avioniques critiques et temps-réel, dans notre cas. Il évoque la possibilité d'enchaîner plusieurs scénarios dans un scénario, ce qui est exactement le cas pour l'ingénierie système avionique où chaque suite de tests suit un plan de vol.

Les scénarios produits grâce à la syntaxe proposée par le BDD utilisent le vocabulaire des experts du domaine et des utilisateurs. De plus, les tests étant écrits en langage naturel, ils donnent l'opportunité aux parties prenantes du projet de s'impliquer pleinement dans la réalisation des scénarios de test et par la même de spécifier le comportement attendu du système à développer.

Sans effort d'automatisation, l'approche BDD perd de sa pertinence ; les instructions produites grâce aux scénarios de haut niveau doivent être transformées en tests exécutables. Cette correspondance est réalisée grâce à des mécanismes d'expressions régulières. Le code nécessaire entre le scénario de test en langage naturel et le code exécutable s'appelle du code glu (*glue code*).

2.2.3 Outils et *framework* BDD

Le développement par l'approche BDD est supporté par des outils spécifiques. Il existe actuellement plus d'une trentaine d'outils de ce type développés pour des langages de programmation différents (PHP, Java, JavaScript, Ruby, Scala, C#, Delphi/Pascal, Groovy, etc.). Nous présentons succinctement les outils les plus connus comme JBehave, Cucumber, RSpec.

Le plus connu est JBehave car le co-administrateur de l'équipe de développement est le précurseur de l'approche BDD [48]. Écrire des tests avec JBehave revient à décrire les scénarios des comportements attendus pour chaque fonctionnalité. Un lien est établi avec le code à tester grâce à des expressions régulières. Les tests peuvent être exécutés par plusieurs séquenceurs, à la fin de chaque exécution un rapport de test sera produit.

Contrairement à JBehave uniquement centré sur le langage Java, Cucumber supporte une grande partie des langages de programmation classiques [24]. Les mécanismes proposés

sont identiques à ceux de JBehave. Des tests d'acceptation par les interfaces sont possibles par l'utilisation du *framework* Selenium [27].

RSpec, quant à lui, a été spécialement développé pour Ruby et son slogan reprend la philosophie initiale du BDD : rendre le TDD productif et amusant [65]. Le langage ubiquitaire utilisé pour les tests d'acceptation est un peu différent de celui proposé par le BDD, le patron *Given-When-Then* étant remplacé par *Describe-Context-It*.

2.3 Choix de l'outil pour la conception de langages

Nous listons dans un premier temps quelques uns des outils disponibles pour concevoir des DSL. Nous séparons les *frameworks* servant à concevoir des DSL textuels et les ateliers de conception de DSL projectionnels. Dans le cadre du projet FUI ACOVAS, nous avons expérimenté un exemple JBehave en utilisant le *framework* Xtext [47] et l'atelier Meta Programming System [49]. Nous avons choisi ces deux outils car ils sont *open source*. Nous détaillons cette expérimentation pour le deux outils. A l'issue de cette expérimentation, nous avons choisi l'atelier MPS et en donnons les raisons à la fin de cette section.

2.3.1 Outils disponibles

Dans cette section, nous étudions les outils permettant la création de DSL [31]. Ces outils sont classés en deux catégories. La première est composée des *frameworks* intégrant la génération d'analyseurs et des mécanismes de transformation du code pour produire des langages supportés par des éditeurs textuels libres. Les éditeurs textuels permettent d'écrire n'importe quel caractère à n'importe quel endroit : c'est pour cela qu'ils sont dits libres. Les outils implémentant des langages projectionnels composent la deuxième catégorie. L'édition des programmes conçus avec ces langages est réalisée grâce à des éditeurs projectionnels.

Nous dressons maintenant un tour d'horizon des outils pour la création de langages qui existent actuellement [31]. Les principaux sont Xtext, Spoofax, MetaEdit+, Intentionnal Software et Meta Programming System (MPS). Les *frameworks* les plus utilisés pour le développement de langages supportés par des éditeurs textuels libres sont :

- Xtext [47] est composé d'un ensemble de *plugins* qui sont utilisés pour créer des langages qui seront utilisables dans des IDE comme Eclipse ou IntelliJ. Ce *framework* exploite Xtend [12], un DSL interne dédié à la programmation de générateurs de code en Java. Il utilise ANTLR pour les analyseurs lexicaux et syntaxiques et exploite des fonctions avancées de génération d'éditeurs textuels pour les DSL.
- Spoofax [85] est une plateforme basée sur Eclipse pour développer des langages informatiques textuels spécifiques à un domaine. Elle a été financée en partie par des organismes de financement ou des entreprises comme Oracle Labs ou Philips Research. Cet outil se base sur SDF (*Syntax Definition Formalism*) pour utiliser des grammaires moins restrictives que des grammaires sous format EBNF. Il utilise Stratego/XT pour la partie sémantique i.e les règles de transformation de code dans un

langage GPL. La particularité de Stratego est que l'on peut générer du code GPL à partir de la syntaxe concrète du DSL.

- MetaEdit+ [81] est un *framework* indépendant pour la création de nouveaux langages graphiques développés en Smalltalk. Il est principalement utilisé pour concevoir des langages de modélisation spécifiques à un domaine. Ces langages sont manipulables *via* des éditeurs graphiques.

Les ateliers de développement de langages projectionnels les plus connus sont :

- Intentionnal Domain Workbench [71] est un atelier de conception de langages payant, développé à la base par Intentionnal Software et qui a été racheté par Microsoft en avril 2017. Son but premier est de fournir un environnement pour que la programmation soit moins complexe. Intentionnal Domain Workbench propose des éditeurs projectionnels pour manipuler des programmes en langages spécifiques et prône le fait que ce type d'éditeurs procure des interfaces naturelles et intuitives pour les non-informaticiens experts du domaine.
- Meta Programming System (MPS) [49] est un atelier de conception de langages *open source* développé par JetBrains. Son avantage principal est qu'il gère des syntaxes textuelles, symboliques, tabulaires ou encore graphiques *via* des éditeurs projectionnels. Il reste simple d'utilisation tout en proposant des fonctionnalités avancées de composition et d'extension des langages et des éditeurs.

2.3.2 Un exemple JBehave implémenté avec Xtext et MPS

Le choix de l'outil de développement utilisé pour la production des langages dédiés au contexte avionique nous a conduit à étudier deux outils : Xtext pour les *frameworks* et Meta Programming System pour les ateliers. Xtext a été choisi pour cette expérimentation car son utilisation est très répandue et il est *open source*. Les langages conçus avec ce *framework* peuvent être interfacés avec des *plugins* permettant l'ajout de fonctionnalités facilitant leur utilisation. MPS a été choisi car son utilisation commence à être répandue et qu'il est *open source*. Il apporte un panel de fonctionnalités, d'interfaces et de langages développés spécifiquement pour l'implémentation des langages projectionnels. Nous avons choisi d'implémenter le langage ubiquitaire instauré par le BDD et implémenté dans les outils comme JBehave, Cucumber etc. pour l'utiliser sur l'exemple simple de la calculatrice.

Ce langage a pour but de générer les cas de test de la multiplication d'une calculatrice à partir de scénarios de test en langage naturel. La multiplication est supportée par une classe Java `Multiply`. La classe `Multiply` est une classe simple contenant deux attributs x et y de type Entier et trois méthodes : un mutateur pour x, un mutateur pour y et une méthode réalisant la multiplication de x par y.

La figure 2.1 montre un exemple de la formalisation d'une *user story* et des scénarios de test grâce à l'outil JBehave. Dans l'approche BDD, la description d'une fonctionnalité se fait par une *user story*. Les cas de test se rapportant à la fonctionnalité de multiplication sont dans notre cas décrits selon les trois étapes préconisées par le langage Gherkin : `Given`,

FIGURE 2.1 – Exemple d'une *user story* et de scénarios

```
As a user
I want to calculate
So that I can get quickly results of complexe arithmetical operations

Scenario: 2 squared
Given a variable x with value 2
When I multiply x by 2
Then x should equal 4

Scenario: 4 squared
Given a variable x with value 4
When I multiply x by 4
Then x should equal 16

Scenario: 3 multiply by 5
Given a variable x with value 3
When I multiply x by 5
Then x should equal 15
```

`When` et `Then`. A la figure 2.2, la clause `Given` initialise la variable x à une valeur passée en paramètre dans le texte de la clause, la clause `When` initialise la valeur de y à une valeur passée en paramètre et effectuera la multiplication et la clause `Then` s'assure que la multiplication est conforme au résultat attendu. Le code produit lors de la transformation de ces scénarios peut-être exécuté par le moteur d'exécution de JUnit qui produira un rapport de test sur l'exécution de ces scénarios.

JBehave est plus générique qu'un DSL dédié au test de la multiplication et permet de produire des cas de test dédiés à n'importe quel domaine métier. Cependant, l'exécution des scénarios de test avec JBehave est possible uniquement si les phrases qui les composent suivent strictement les patrons prédéfinis dans le code glu. Nous avons implémenté le même comportement que JBehave mais avec un langage dédié au test de la multiplication. Il faudrait en effet développer un autre langage si l'on changeait de fonctionnalité à tester. Nous nous sommes concentrés sur l'implémentation du langage ubiquitaire, et n'avons pas implémenté les patrons de phrases tels quels. L'utilisateur est libre d'écrire ses phrases, l'important étant qu'elles se terminent par les valeurs de x, de y et du résultat attendu.

2.3.2.1 Validation de la multiplication avec Xtext

Nous avons choisi d'installer Xtext dans l'environnement de développement Eclipse. La création d'un projet Xtext permet d'avoir quatre *plugins* dédiés à la conception d'un DSL.

- un *plugin* dédié à la définition de la grammaire et des composants du langage (analyseur syntaxique, analyseur lexical, édition des liens, validation, générateur de code),
- un *plugin* dédié au paramétrage des éditeurs textuels,

FIGURE 2.2 – Code glu entre les instructions des scénarios et le code de test correspondant

```
public class ExampleSteps {
    Multiply myInstance = null;

    @Given("a variable x with value $value")
    public void givenXValue(@Named("value") int value) {
        myInstance = new Multiply();
        myInstance.setX(value);
    }

    @When("I multiply x by $value")
    public void whenImultiplyXBy(@Named("value") int value) {
        myInstance.setY(value);
        myInstance.multiply();
    }

    @Then("x should equal $value")
    public void thenXshouldBe(@Named("value") int value) {
        if (value != myInstance.evaluate())
            throw new RuntimeException("x is " + myInstance.evaluate()
                    + " but should be " + value);
    }
}
```

– un *plugin* dédié aux tests unitaires du langage,
– un *plugin* dédié à l'intégration des fonctionnalités avancées de l'IDE.

Tout comme pour l'expérimentation MPS, nous n'avons implémenté qu'un prototype de langage et n'avons produit que la syntaxe et la génération de code.

Xtext propose de commencer par définir la syntaxe concrète du langage. Il se base sur la syntaxe EBNF utilisée par l'outil ANTLR pour définir la grammaire des langages conçus. La figure 2.3 montre la syntaxe du langage que nous proposons pour produire les cas de test de multiplication. Xtext génère alors les fonctionnalités de base s'intégrant aux éditeurs de code textuels fournis par Eclipse. Ces fonctionnalités sont l'auto-complétion, la coloration syntaxique ... L'ajout de fonctionnalités plus avancées comme du *refactoring* ou du *debug* demandent de coupler le langage avec d'autres *plugins* de l'IDE Eclipse. Le nombre de *plugins* à assembler et à manipuler peut devenir important et il n'existe que très peu de documentation consacrée à ces *plugins*.

La génération du code avec Xtext est uniquement de modèle vers texte. Cette transformation est rendue possible par la méthode `doGenerate` donnant accès à l'AST du programme DSL à transformer. La méthode `doGenerate` est présentée à la figure 2.4. Le code produit à la génération est vu comme une chaîne de caractères passée en paramètre à la fonction `generateFile` avec le nom du fichier à générer. Cette chaîne de caractères est créée par la méthode `createTestsFromInput` de la figure 2.5 faisant appel à la méthode `createTest` présentée à la figure 2.6. Cette méthode est décomposée en trois parties : une

FIGURE 2.3 – Grammaire du langage

```
tests:
    (story+=Story)
    (Scenario+=Scenario)*;

Story:
    'Story:' name=myText
    'As a' roleStory+=myText
    'I want' feature+=myText
    'So that' benefitsStory+=myText;

Scenario:
    'Scenario:' name=myText
    given+=given when+=when then+=then;

given:
    'given' text+=myText m1+=INT;

when:
    'when' text+=myText m2+=INT;

then:
    'then' text+=myText result+=INT;

myText:
    ID+;
```

partie pour la clause `Given`, une partie pour la clause `When` et une partie pour la clause `Then`. Les programmes de transformation sont écrits grâce à Xtend et des méthodes de parcours de l'AST. Nous pouvons remarquer dans cet exemple que même si un scénario est composé d'un unique `Given`, `When` et `Then` et que chacune de ces clauses référence une unique valeur entière, nous sommes contraints d'utiliser la propriété `head` pour accéder à l'unique élément d'une liste retournée par défaut par les méthodes de parcours de l'AST.

FIGURE 2.4 – Génération Xtext grâce à la méthode `doGenerate`

```
class NewBehaveGenerator implements IGenerator {

    override doGenerate(Resource input, IFileSystemAccess fsa) {
        fsa.generateFile("generatedTest.java", new Generator(input).createTestsFromInput())
    }

}
```

La figure 2.6 présente les méthodes injectant les informations de l'AST dans du code GPL sous forme d'une chaîne de caractères à compléter. Cette transformation modèle vers texte ne garantit aucunement la validité syntaxique et sémantique du code produit. De plus, la transformation modèle vers texte prend la forme d'un programme impératif et non d'une suite de transformations de concepts à texte, ce qui contredit l'approche méta-modèle prônée par la mouvance *grammarware* [53].

2.3.2.2 Validation de la multiplication avec MPS

Un nouveau langage est composé à sa création de différents aspects permettant sa manipulation : structure, éditeur, contraintes, comportement, système de types et générateur comme le montre la figure 2.7.

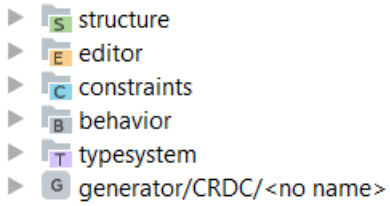
FIGURE 2.5 – Génération Xtext pour chaque scénario

```
def createTestsFromInput() {
    initTemplate();
    for (g : input.allContents.filter(typeof(Scenario)).toIterable) {
        createTest(g.given.head.m1.head,
             g.when.head.m2.head,
            g.then.head.result.head, g.name)
    }
    finalizeTemplate();
    return template;
}
```

FIGURE 2.6 – Injection des informations de l'AST dans le code Java produit

```
def createTest(Number m1, Number m2, Number res, String name) {
    givenPartOfTheTest(m1, name);
    whenPartOfTheTest(m2)
    thenPartOfTheTest(res)
    return template;
}

def givenPartOfTheTest(Number m1, String name) {
    var String nameWithoutSpace = name.replaceAll("\\s", "")
    template += "
@Test
public void test" + nameWithoutSpace + "(){
    myInstance = new Multiply();
    myInstance.setX(" + m1 + ");"
}

def whenPartOfTheTest(Number m2) {
    template += "
    myInstance.setY(" + m2 + ");
    myInstance.multiply();"
}

def thenPartOfTheTest(Number res) {
    template += "
    assertEquals(myInstance.evaluate(), " + res + ");\n }"
}
```

FIGURE 2.7 – Structuration MPS d'un projet de langage



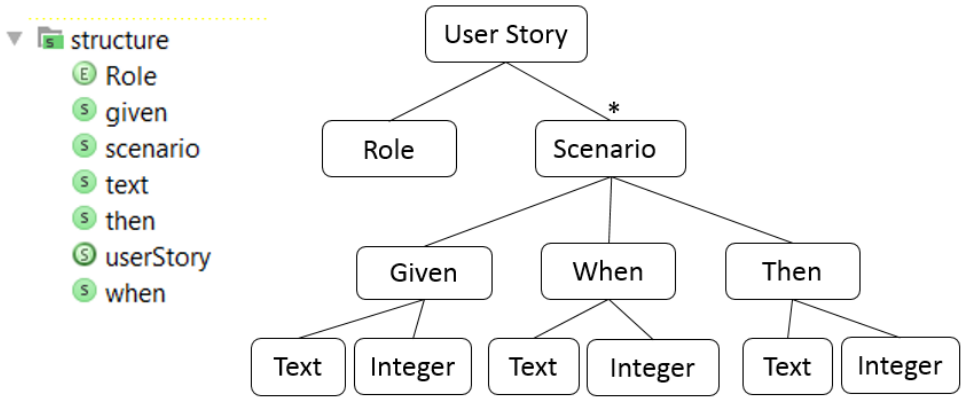
Le développement d'un langage avec MPS commence par la description de la structure (schéma ou grammaire abstraite) des concepts du langage. Les concepts ou éléments de la grammaire abstraite créés au travers de MPS sont composés pour former le méta-modèle du langage. Il existe deux types de liens possibles entre concepts d'un méta-modèle :

- la composition au sens classique du terme, par exemple une *user story* est composée d'une liste de scénarios,
- le lien de référence qui permet un adressage unique d'un concept particulier par plusieurs concepts du langage.

La figure 2.8 montre la liste des concepts MPS nécessaire à la définition du schéma pour le langage testant une multiplication. Dans ce modèle, une *user story* est spécifiée pour un type d'utilisateur (*Role*) et est composée d'une liste de scénarios. Un scénario est constitué des trois concepts *given*, *when* et *then*. Chacun de ces trois concepts est composé d'une chaîne de caractères (*text*) et d'une valeur numérique.

Il est possible d'y ajouter des contraintes sémantiques contrôlant que les modèles produits sont conformes aux règles métier ne pouvant pas être retranscrites par le méta-modèle. Par exemple, lors de la modélisation des liens de parenté entre des personnes, il faut s'assurer qu'une personne ne puisse pas être son propre ancêtre. Tout comme dans l'expérimentation Xtext, nous n'avons pas utilisé ces contraintes sémantiques.

FIGURE 2.8 – Liste des concepts du langage de test de la multiplication



Comme nous sommes dans un contexte d'éditeurs projectionnels, chaque concept doit définir son propre éditeur projectionnel encapsulant la grammaire concrète qui lui est associée. Il s'agit donc de créer les éditeurs qui manipuleront les concepts du langage. MPS propose le `Concept Editor` pour mettre en place l'édition projectionnelle de programmes du langage. La figure 2.9 est un exemple de code produit pour la création de l'éditeur du concept `scenario`.

Les concepts et mots-clés sont organisés dans des collections soit verticales ou horizontales. Dans le premier cas, la phrase est contenue sur plusieurs lignes ; par exemple, les trois clauses *given*, *when* et *then* sont formalisées sur trois lignes différentes. Dans le second, la phrase est contenue sur une seule ligne, comme pour le concept de `scenario` introduit par le mot-clé *scenario* suivi d'un nom sur une même ligne. Il est aussi possible d'ajouter plusieurs niveaux d'indentation. Un éditeur met en jeu les sous-concepts qui lui sont attribués *via* la hiérarchie des concepts proposée par la structure du langage. Dans notre exemple, le concept `scenario` est composé des sous-concepts `given`, `when` et `then`. Chaque sous-concept fait appel à son propre éditeur pour lui faire correspondre une syntaxe concrète.

FIGURE 2.9 – Conception des éditeurs projectionnels MPS

```
<default> editor for concept scenario
  node cell layout:
    [/
      [> scenario: { name } <]
      % given %
      % when %
      % then %
      <constant>
    /]
```

La troisième partie du développement comprend la mise en place du générateur de code. Tout comme dans Xtext, des fonctionnalités sont fournies pour parcourir simplement l'AST d'un programme. Le développement d'un générateur de code peut être réalisé selon deux méthodes : la transformation de modèle à texte (*Model2Text*) comme implémentée dans l'expérimentation Xtext ou la transformation de modèle à modèle (*Model2Model*) [26] comme implémentée dans l'expérimentation MPS, la grammaire du langage Java étant encodée sous MPS. Le *template* du code généré est compilable puisqu'il est produit *via* un éditeur projectionnel Java. Le *template* de génération présenté à la figure 2.10, produit un bloc de test JUnit pour chaque scénario de test d'un programme dans le DSL. Nous pouvons constater que le *template* du code produit est au centre des préoccupations de la transformation de modèle. Il ne reste plus qu'à remplacer les informations du *template* contenues entre `$[` et `]` par des informations contextuelles de l'AST du programme en DSL.

FIGURE 2.10 – Template de génération de code MPS

```
root template
input userStory

public class $[TestInput] extends TestCase {

  /*package*/ Multiply myInstance = new Multiply();

  $LOOP$ @Test()
         public void $[testScenario]() {
           myInstance.setX($[10]);
           myInstance.setY($[10]);
           myInstance.multiply();
           TestCase.assertEquals(myInstance.evaluate(), $[100]);
         }
```

Chaque valeur de l'AST trouve alors sa place dans le *template* représentant le code généré. Les figures 2.11, 2.12, 2.13 et 2.14 montrent les liens entre les différentes valeurs à substituer et les informations de l'AST.

FIGURE 2.11 – Lien entre $LOOP$ du template et les scénarios de l'AST

```
iteration sequence : (genContext, node, operationContext)->sequence<node<>> {
                       node.scenario;
                     }
```

FIGURE 2.12 – Lien entre le paramètre de la méthode `setX` et la valeur de la clause *Given*

```
value : (templateValue, genContext, node, operationContext)->int {
          node.given.givenValue.value;
        }
```

Le figure 2.15 montre le code généré correspondant aux scénarios formalisés pour la *story* JBehave présentée au début de l'expérimentation. Ce code est une classe Java composée de trois méthodes annotées avec la balise `@Test` de JUnit permettant à l'exécuteur JUnit de produire un rapport. Le code généré *via* Xtext, lors de cette expérimentation, est équivalent au code présenté ici.

2.3.3 Résultat de l'expérimentation

Nous abordons la comparaison entre Xtext et MPS selon deux points de vue : le point de vue concepteur de la génération de code et le point de vue de l'utilisateur final du DSL. D'un point de vue génération de code, l'approche MPS se focalise sur le résultat à produire,

FIGURE 2.13 – Lien entre le paramètre de la méthode `setY` et la valeur de la clause *When*

```
value : (templateValue, genContext, node, operationContext)->int {
          node.when.whenValue.value;
        }
```

FIGURE 2.14 – Lien entre le résultat attendu de la méthode `assertEquals` et la valeur de la clause *Then*

```
value : (templateValue, genContext, node, operationContext)->int {
          node.then.thenValue.value;
        }
```

FIGURE 2.15 – Code généré de la *user story* présentée à la figure 2.1

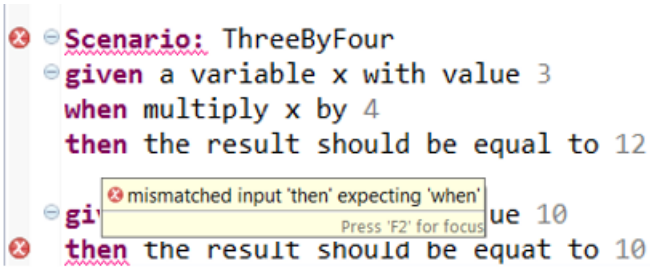
```
public class StoryMultiply extends TestCase {

  /*package*/ Multiply myInstance = new Multiply();

  @Test
  public void testTwoSquared() {
    myInstance.setX(2);
    myInstance.setY(2);
    myInstance.multiply();
    TestCase.assertEquals(myInstance.evaluate(), 4);
  }
  @Test
  public void testFourSquared() {
    myInstance.setX(4);
    myInstance.setY(4);
    myInstance.multiply();
    TestCase.assertEquals(myInstance.evaluate(), 16);
  }
  @Test
  public void testThreeByFive() {
    myInstance.setX(3);
    myInstance.setY(5);
    myInstance.multiply();
    TestCase.assertEquals(myInstance.evaluate(), 15);
  }
```

ce qui s'avère plus simple qu'avec l'approche impérative proposée par Xtext. De plus, en se basant sur un éditeur projectionnel, le *template* de génération ne peut générer du code contenant des erreurs de compilation comme cela pourrait être le cas en Xtext.

L'utilisateur final du DSL est concerné par un éditeur textuel produit par Xtext ou bien par un éditeur projectionnel conçu avec MPS. Cette différence a un impact direct sur l'utilisabilité des DSL par des non-informaticiens.

Un éditeur textuel reconnaît les mots-clés du langage pour les faire apparaître avec un style particulier afin qu'ils soient facilement reconnaissables par l'utilisateur. La notification des erreurs est faite à l'utilisateur en soulignant en rouge le code où l'incohérence est détectée. La figure 2.16 montre un scénario dans le DSL entaché d'erreurs dans un éditeur Xtext. L'éditeur notifie une erreur sur le premier scénario alors que celui-ci est correct ; elle concerne le deuxième cas de test qui ne commence pas par le mot-clé `Scénario`. Pour la deuxième erreur, l'éditeur informe l'utilisateur que le mot-clé `when` est attendu à la place du mot-clé `then`. Sur cet exemple, nous voulons montrer que l'utilisation d'éditeurs textuels demande aux utilisateurs finaux du langage de bien connaître la syntaxe de ce langage alors que l'utilisation d'une syntaxe avec des éditeurs projectionnels est guidée par l'assemblage des éditeurs selon le méta-modèle du langage.

FIGURE 2.16 – Scénario incorrect avec Xtext



Au lieu de souligner les erreurs syntaxiques, les éditeurs projectionnels ne donnent pas la possibilité de construire un programme dont la structure est erronée. Néanmoins, des erreurs sémantiques sont toujours possibles. Les figures 2.17 et 2.18 montrent les interfaces paramétrées dans MPS devant lesquelles se trouve un développeur lorsqu'il souhaite formaliser respectivement une *story* et un scénario. L'utilisateur final n'a plus qu'à remplir les informations manquantes. De fait, les éditeurs projectionnels répondent mieux aux problèmes d'ambiguïté des grammaires car l'utilisateur final du DSL n'a pas à gérer l'utilisation des mots-clés grâce à la manipulation directe des concepts du langage. Les formats de syntaxe proposés par les éditeurs projectionnels MPS sont multiples [87]. Ils peuvent être textuels mais aussi tabulaires, graphiques, symboliques ou un mélange des quatre. Les incohérences sémantiques sont vérifiées à la volée par des contraintes [11].

La figure 2.19 est un exemple d'ambiguïté supplémentaire qu'il faudra prendre en compte en utilisant des éditeurs textuels. Les deux erreurs soulignées en rouge montrent

FIGURE 2.17 – Instanciation d'une *Story* vide avec MPS

```
Story <no name>
As a  User
I want <no textProperty>
so that I can <no textProperty>
```

FIGURE 2.18 – Instanciation d'un scénario vide avec MPS

```
scenario: <no name>
given <no name> <int constant>
when <no name> <int constant>
then <no name> <int constant>
```

que l'éditeur reconnaît un mot-clé du langage (`I want`) dans la description d'un scénario alors que ces mots font partie de l'étape `when`. Cet exemple nous montre que l'expressivité des grammaires EBNF n'est pas suffisante pour un DSL métier. L'utilisateur final du DSL doit pouvoir écrire ce qu'il veut dans les clauses *given*, *when* et *then* indépendamment de la grammaire d'une *user story*. Étant contextualisés par le méta-modèle du langage, les éditeurs font abstraction de ce genre d'erreurs. Ceci facilite, selon nous, l'utilisation de langages munis d'éditeurs projectionnels pour des utilisateurs finaux n'ayant pas de compétences en informatique.

FIGURE 2.19 – Ambiguïtés syntaxiques

```
Story: Story Multiply
As a User
I want to multiply two numbers
So that get results of complex arithmetical operations


Scenario: twoSquared
required (...)+ loop did not match anything at input 'I want
when I want to multiply x by  2
then should be 4

Scenario: ThreeByFour
mismatched character 'm' expecting 'w'ue 3
when I multiply x by 4
then the result should be equal to 12
```

Xtext est un outil puissant mais son besoin d'être couplé à un IDE existant le rend complexe à utiliser. Notre choix s'est tourné vers les éditeurs projectionnels car ils sont,

a priori, plus accessibles à un public non-informaticien qui se verra guidé par la syntaxe concrète. MPS est un outil proposant une vision projectionnelle dont le panel de fonctionnalités semble le plus étoffé pour un outil *open source* [19] [31].

Les ateliers de conception de langages ne sont pas très matures actuellement mais leur évolution est croissante et constante. Nous avons utilisé quatre versions importantes de MPS durant ces trois dernières années. De nouveaux langage cibles de génération sont possibles tels XML, Ruby ou Python [50], ce qui nous aidera grandement dans la suite de nos expérimentations. MPS propose aussi des mécanismes de test et de *debug* des langages développés [20], et des fonctionnalités de composition des DSL.

2.4 Une approche BDD pour les tests d'intégration système

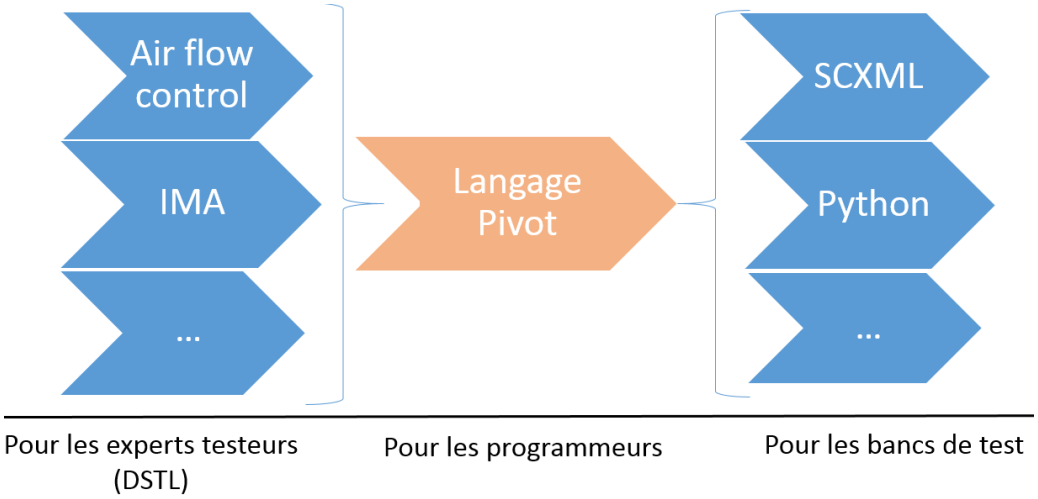
Dans un premier temps, nous avons proposé d'utiliser les DSL pour formaliser les procédures de test spécifiques à chaque ATA (ATA 21 *air flow control* et ATA 42 IMA). Puisque ces langages sont proches du niveau de langage des testeurs et proches des exigences, nous avons nommé ces langages DSTL (*Domain Specific Test Languages*). Pour assurer l'automatisation des tests sur différents bancs de test, nous avons exploité l'approche BDD et mis en œuvre plusieurs chaînes de transformations [15]. Les concepts de *user stories* et de scénarios sont transposés dans le monde avionique en objectifs de test et en plans de vol. Chaque plan de vol teste un objectif et est composé d'une suite de scénarios [92]. Le lien entre les langages de haut niveau et le code exécutable est réalisé par un langage intermédiaire qui permet l'ajout de code glu.

2.4.1 Schéma global du *framework*

Le schéma global du *framework* BDD pour l'automatisation des test d'intégration dans un contexte avionique est présenté à la figure 2.20. On y retrouve tout ou partie des travaux réalisées durant cette thèse. Les travaux concernant le domaine *air flow control* seront présentés au chapitre 3, et ceux concernant les procédures IMA le seront au chapitre 4. Ce *framework* est composé de trois niveaux de langages : les DSTL (à gauche), le langage pivot (au centre) et les langages exécutables que nous avons adressés (à droite). Ce *framework* se veut extensible et l'ajout d'un nouveau DSTL sera facilité par le langage pivot, de même pour les langages exécutables.

A l'instar du BDD, le niveau le plus élevé correspond à un langage proche du langage naturel où le vocabulaire du domaine métier est omniprésent. De plus, les DSTL sont simples d'utilisation car dédiés aux experts testeurs. Ils pourront formaliser plus facilement leurs procédures de test grâce à ces langages. La structuration des tests produits grâce aux DSTL est similaire à celle des procédures non-formalisées actuellement utilisées par l'un de nos partenaires du projet ACOVAS. L'initialisation des systèmes sous-test est exécutée automatiquement à chaque cas de test. La décomposition des procédures de test en cas de

FIGURE 2.20 – Orchestration de langages dans une approche BDD



test et en blocs de test suit un découpage proche de ce que proposent les *frameworks* de test dans le domaine logiciel.

Le fossé existant entre les descriptions des cas de test en DSTL et le code exécutable étant bien trop grand, nous proposons un langage intermédiaire spécifique aux systèmes avioniques que nous pensons suffisamment générique pour couvrir les besoins des différents ATA. L'utilisation du langage pivot simplifie la transformation d'un DSTL vers le code exécutable en proposant un premier niveau de transformation du DSTL vers le langage pivot qui concerne l'automatisation des instructions testables, et un deuxième niveau de transformation du langage pivot vers les plates-formes exécutables qui concerne la mise en œuvre de l'exécution des tests. Ce langage est dit pivot car il permet d'être transformé dans de multiples langages d'exécution, ce qui permet aux tests formalisés dans nos DSTL d'être exécutables sur plusieurs types de bancs.

Le langage pivot est donc un langage comprenant simplement les instructions nécessaires à la transformation des procédures de test en langage de haut niveau vers du code exécutable. Il comporte un nombre très réduit d'instructions, indépendantes les unes des autres permettant aux programmeurs testeurs de produire des procédures de test exécutables dans un langage impératif générique. Le nombre d'instructions est certainement amené à croître pour prendre en compte des besoins d'automatisation des différents domaines et métiers de l'avionique.

Les deux cibles de génération qui nous ont été demandées sont SCXML [7] et Python associé à des bibliothèques spécialisées. Elles seront détaillées au chapitre suivant de la thèse. La première cible de génération est une plate-forme de test conçus par l'un des partenaires utilisateur des bancs de test. Ce langage est accompagné d'un éditeur graphique où les procédures de test sont encodées par un assemblage d'automates temporisé. La

deuxième cible de génération est une plate-forme de test fournie avec les bancs de test vendus par l'un de nos partenaires du projet.

2.4.2 Bénéfices attendus

Le *framework* proposé contribue à formaliser les procédures d'intégration et automatiser leur exécution. Il propose de traduire des cas de test décrits *via* des langages, chacun spécifique à un domaine avionique, dans plusieurs codes exécutables sur un banc de test.

La traduction en code exécutable est simplifiée par la présence d'un langage pivot qui réduit le nombre de transformations à développer. En effet, prenons `n` le nombre de langages de la famille des DSTL et `m` le nombre de langages d'exécution. Grâce à l'utilisation du langage pivot, le nombre de transformations est de `n + m`, pour `n x m` projections d'un DSTL vers un langage exécutable du banc.

Ce *framework* vise à séparer les préoccupations de chaque acteur de la conception des procédures automatisées. Les experts en test système devront se concentrer sur les comportements qu'ils souhaitent tester et sur leur description par des scénarios de test dans des langages dédiés. Les programmeurs réaliseront l'automatisation de ces scénarios *via* l'utilisation du langage pivot.

Les langages de haut niveau ont pour objectif de maintenir une structure et une syntaxe proches des procédures existantes pouvant être formalisées en langage naturel. Ceci permet de réduire l'effort demandé lors de l'apprentissage de ces langages. Nous voulons que chaque scénario de test se concentre sur un seul objectif et donc un seul comportement à tester. Nous voulons produire des tests isolés les uns des autres. Nous voulons, de plus, que l'intention de chacun de ces tests soit facilement identifiable lors de la lecture des scénarios de test. Au delà d'être automatisées, ces procédures de test deviennent un vecteur de communication entre les testeurs d'un même domaine parlant le même dialecte, entre les testeurs et les programmeurs de test et surtout entre les testeurs et les concepteurs système. Le but du BDD étant de faciliter l'intégration d'un client ou d'un *product owner* au sein d'une équipe de développement, nous pensons que notre approche peut favoriser un rapprochement entre le bureau d'étude et le centre de test, voire même d'assurer une certaine forme de traçabilité entre les exigences et les tests automatisés.

Chapitre 3

Expérimentation du *framework* sur l'ATA 21

Pour cette expérimentation, nous nous sommes basés sur des procédures de test conçues pour la démonstration finale du projet ACOVAS. Ces procédures ont pour but de valider le comportement des systèmes régulant la température de l'air dans les différentes zones d'un avion (ATA 21). Actuellement, les procédures automatisées de l'ATA 21 ont une syntaxe tabulaire. Cette syntaxe se base sur une API adressable par une IHM fournie avec le banc de test. Cette syntaxe est uniquement composée d'ordres simples : mise à jour d'une valeur du SUT, récupération d'une valeur du SUT et attente d'un certain quantum de temps. Ces tableaux sont ensuite transformés en procédures exécutables *via* des programmes *ad hoc* écrits en Visual Basic. Après avoir expliqué nos attendus à l'équipe de programmeurs testeurs, nous avons élaboré conjointement une syntaxe textuelle permettant de rendre ces tests unitaires dans le sens agile du terme, c'est à dire isolés les uns des autres. La syntaxe proposée élimine les clauses *wait* suivies d'une lecture d'un ou plusieurs paramètres *via* l'IHM du banc de test en les remplaçant par des assertions permettant l'automatisation complète des tests. A ce niveau de maturité, le langage obtenu exhibe un niveau d'abstraction intermédiaire proche des considérations des programmeurs et proche des problématiques de transformation vers des langages exécutables. Nous nous sommes basés sur cette syntaxe textuelle pour proposer un langage informatique spécifique aux besoins d'automatisation des tests d'intégration pour tous les domaines avioniques. Nous l'avons appelé langage pivot dans notre *framework*.

Afin de valider notre approche globale, nous avons souhaité construire un DSTL répondant plus spécifiquement aux besoins de l'ATA 21 en embarquant plus de sémantique par des phrases proches du langage naturel et proches des préoccupations d'un testeur du domaine devant spécifier le comportement attendu par des cas de test. Ces besoins ont été identifiés par l'étude des procédures et par des discussions avec les experts testeurs du domaine. Le DSTL ATA 21 fournit des outils pour la formalisation des procédures de test

en s'attachant à conserver l'intention de chaque cas de test. La syntaxe du langage devra rester simple et la sémantique de chacune des instructions sera clairement définie et non ambiguë de manière à pouvoir être utilisée par un expert système du domaine, n'ayant pas forcément de compétences en programmation. Avec ce langage de tests de haut-niveau, nous souhaitons que les cas de test soient un moyen de communication entre le bureau d'études et les testeurs en phase d'intégration.

Dans ce chapitre, nous présentons la conception du langage pivot à partir des procédures fournies par le projet ACOVAS (section 3.1) et l'élaboration du DSTL répondant aux besoins de formalisation de l'ATA 21 (section 3.2). Nous montrons ensuite comment la chaîne de transformations du *framework* produit un code exécutable à partir de procédures écrites par des testeurs dans le DSTL ATA 21 (section 3.3).

3.1 Emergence du langage pivot

Le langage présenté dans cette section est le langage utilisé comme langage intermédiaire de transformation dans le *framework* BDD que nous proposons. Il se focalise sur l'automatisation des actions de test. Son objectif est de formaliser la manipulation des paramètres avioniques du système testé et de piloter les outils disponibles sur le banc de test. Ce langage se veut commun à tous les domaines avioniques et a pour but de réduire les efforts de traduction des procédures en DSTL. Le langage pivot est la cible de génération de tous les langages de haut niveau (DSTL). Il est composé d'un nombre réduit d'instructions pouvant être amené à grandir pour couvrir de nouveaux besoins.

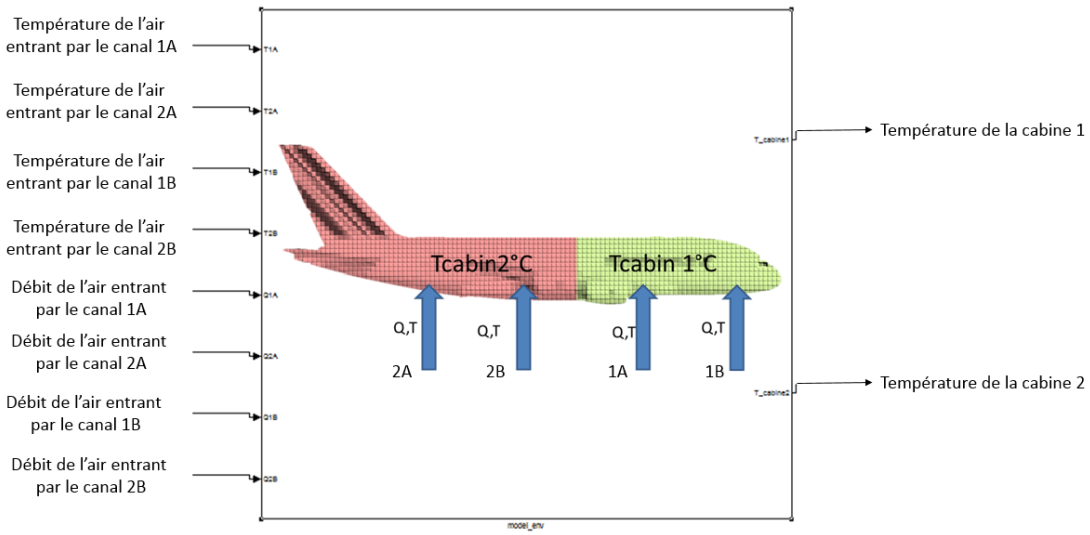
Il a été créé pour être utilisé par des testeurs développeurs et sa syntaxe se veut moins verbeuse que celle des DSTL. Son utilisation dans la chaîne de transformations est masquée aux testeurs utilisant les langages de tests de haut niveau. Il n'a pas été conçu pour être utilisé pour coder directement des tests automatisés. Cependant, il est accessible aux utilisateurs du *framework*. Ce niveau intermédiaire de transformation peut être utilisé pour vérifier le codage d'une chaîne de transformations, pour déboguer une procédure de test décrite à l'aide d'un DSTL et pour éventuellement coder directement une nouvelle suite d'actions correspondant à une nouvelle instruction de haut niveau avant qu'elle ne soit définie pour un DSTL.

3.1.1 Contexte

Le cas d'étude ACOVAS s'attache à valider le comportement du système de régulation de l'air des différentes zones de l'avion. La figure 3.1 est une représentation du modèle simulant les variations de température des zones contrôlées par le système sous test. Ce modèle de simulation propose de distinguer deux zones dans l'avion où chacune est alimentée par deux canaux d'arrivée d'air. Le système manipule un environnement composé de capteurs de température disposés dans chacune des zones et d'actionneurs contrôlant l'ouverture des

vannes pour ajuster le débit d'air injecté dans ces zones. Ce système sous test héberge les applications logicielles pour la régulation de l'air sur une plate-forme matérielle.

FIGURE 3.1 – Modèle de simulation couplé au système d'air conditionné



Tout l'art d'une procédure d'essai consiste à paramétrer les interactions du SUT avec un banc de test afin que le SUT puisse être considéré comme une boîte noire avec des entrées-sorties. Ceci permet d'imaginer des scénarios de test de façon plus intuitive.

Pour simplifier la mise en place des moyens d'essai, seuls les actionneurs contrôlant l'ouverture et la fermeture des vannes sont connectés à la baie de test ; les vannes ne sont donc pas réellement connectées aux canaux d'alimentation en air de chaque zone. Ces vannes sont pilotées par le calculateur hébergeant les fonctionnalités logicielles de régulation de l'air par l'intermédiaire du banc de test. Le banc sert aussi de passerelle de communication entre le système sous test et le modèle de simulation de l'environnement.

Les changements de température dans les zones sont simulés par le modèle d'environnement en fonction du taux d'ouverture de chaque vanne. Ce modèle d'environnement prend en compte de nombreux paramètres pour faire en sorte que son comportement soit le plus réaliste possible. Par exemple, il intègre le volume d'air de la zone à contrôler ou encore la température de l'air extérieur à la cabine. Il prend en entrée la valeur du débit (en fonction du niveau d'ouverture de la vanne) et de la température de l'air fourni par chaque canal. La valeur de la température simulée dans les zones est transmise en continu au système contrôlant la régulation de la température pour qu'il puisse ajuster la température des zones par rapport aux consignes demandées.

Nous souhaitons proposer un langage donnant la possibilité de modifier les consignes et valeurs d'entrée du système et de vérifier que les températures attendues soient bien atteintes. Lors de la défaillance d'un des canaux d'alimentation, le système doit agir sur

les autres canaux pour augmenter les débits d'air injecté dans les zones afin de compenser la défaillance.

3.1.2 Conception du langage pivot

Les procédures d'essai automatisées de l'ATA 21 fournies par l'un des partenaires du projet ACOVAS prennent la forme de tableaux. Ces tableaux listent les actions d'affectation sur les paramètres avioniques manipulés et les assertions du test. Les actions d'affectation stimulent le SUT alors que les assertions vérifient les réponses qu'il produit. Ces tableaux sont transformés en code exécutable par des programmes spécifiquement développés pour ces besoins. La figure 3.2 est un extrait d'une procédure tabulaire. La première colonne (*Task description*) décrit l'intention de la liste des affectations ou d'assertions contenues dans le bloc de test. La deuxième colonne (*Time*) décrit le temps écoulé depuis le début de l'exécution de la procédure. Le nom de l'outil ou de l'interface utilisateur permettant de réaliser les actions pour ce bloc de test est précisé à la troisième colonne (*Control desk screen*). Les paramètres formels manipulés sont spécifiés par la quatrième colonne (*Item*) ; chaque paramètre fait référence à un paramètre d'un outil, de l'interface utilisateur ou bien peut correspondre à un paramètre réel du SUT. La colonne qui suit (*Unit*) indique l'unité du paramètre utilisé pour cette action. Enfin la dernière colonne (*Value to be applied*) est la valeur à appliquer au paramètre ou la valeur attendue.

FIGURE 3.2 – Reproduction d'une procédure tabulaire

Task description	Time	Control desk screen	Item	Unit	Value to be applied
Increase both TLA from 0 to 25:	5s	ENGINES	**ENG_1** *TLA*	°	25
			ENG_2 *TLA*	°	25
Check on the synoptic that as soon as the IP pressure is higher than the threshold the LH HPV is commanded to close.	185s	ARINCTOOL_BMC1_CFDIU	*LH_HPV_POS_LABEL_067_ BMC1_BUS_A429_BMC1_LS*	N/A	***not fully closed***
			LH_HPV_FAILED_OPEN_ LABEL_067_BMC1_BUS_A429_BMC1_LS	N/A	***Failed open***
Check also that the LH HRV remains open			*LH_BLEED_FAULT _LABEL_067_BMC1_BUS_A429_BMC1_LS*	N/A	***Fault***
			LH_PRV_POS_LABEL_067_ BMC1_BUS_A429_BMC1_LS	N/A	***fully closed***
After 180s check that the LH HRV failed open message is send and that the LH PRV is commanded to close and the LH bleed fault light is lightened.		EBas valve	*LH PRV – measured current*	mA	***0***
			RH AFD– measured current	mA	***< 400***
			LH AFD– – measured current	mA	***400***
			RH FAAFD–V– measured current	mA	***> 0***
			LH HPV – state	-	***MANU***
			LH HPRRV – manual state	-	***OPEN***
			RH HSPV – measured state	-	***CLOSE***

La lecture et la compréhension des procédures tabulaires sont rendues complexes de par la prolifération des paramètres avioniques et des paramètres pour manipuler les outils et interfaces de contrôle du test. Ces paramètres sont utilisés pour décrire la liste des actions à réaliser lors d'un test. Des phrases sont apposées en regard de chaque bloc regroupant une liste d'affectations et d'assertions. Elles décrivent succinctement l'intention des affectations et assertions réalisées par le bloc. Beaucoup de sigles spécifiques au contexte sont utilisés dans ces phrases, rendant la compréhension de la procédure complexe.

Nous avons reconsidéré ces procédures tabulaires en collaboration avec les experts de l'ATA 21 pour produire des procédures de test avec une syntaxe textuelle. Les procédures textuelles informelles produites avec les experts testeurs ont une syntaxe très épurée : très peu de mots-clés sont utilisés et les différents paramètres avioniques ne sont pas masqués dans la procédure. Elles ont été écrites pour la démonstration finale du projet ACOVAS. Le niveau d'abstraction utilisé pour ces procédures est assez fin pour permettre de les transformer aisément en code exécutable : les paramètres avioniques réels sont visibles dans la syntaxe du langage. Nous avons donc choisi de concevoir un premier langage à partir de cette syntaxe textuelle et de considérer ce langage comme niveau intermédiaire de traduction des procédures formalisées par les DSTL de notre *framework*.

Ces procédures textuelles proposent un premier bloc (*Initial conditions*) permettant d'initialiser l'état du modèle d'environnement (*Environment parameters*) et les paramètres de test (*Test parameters*). La figure 3.3 montre un exemple d'un bloc d'initialisation pour ces procédures textuelles.

FIGURE 3.3 – Bloc d'initialisation

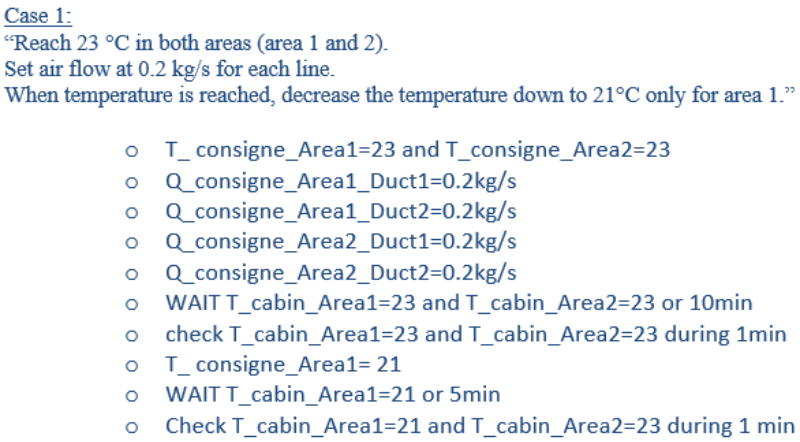
Function 1: air flow influence with test parameters set to 1
Initial conditions:

- Environment parameters:
  - T_cabin_Area1=18 and T_cabin_Area2=18
  - T_outside: 25°C
  - Q_consigne_Area1_Duct1=0kg/s
  - Q_consigne_Area1_Duct2=0kg/s
  - Q_consigne_Area2_Duct1=0kg/s
  - Q_consigne_Area2_Duct2=0kg/s
- Test parameters:
  - Kp1=1
  - Ki1=1
  - Kd1=1

Ce bloc d'initialisation rejoué entre chacun des tests permet de remettre le SUT dans l'état de départ adéquat. Il est suivi de plusieurs blocs de test, correspondant chacun à un comportement différent à tester. Les différences entre les cas de test d'une même procédure peuvent être minimes. Un bloc de test est aussi caractérisé par un commentaire explicitant son intention. La figure 3.4 est un exemple de cas de test. Ce bloc de test vérifie que le

système répond aux consignes de température pour les deux zones dans le temps imparti et que cette température est constante pendant un certain laps de temps. Il s'assure ensuite que le changement de la consigne de température pour une des deux zones est bien pris en compte par le système et que ce changement n'a pas d'impact sur la température de l'autre zone. Les deux zones ayant des températures différentes, le système doit maintenant être capable de maintenir ces températures stables pendant une minute. Pour ce test, les vannes d'alimentation ont un débit d'air de 0,2 Kg/s pour les deux zones de l'avion.

FIGURE 3.4 – Bloc de test



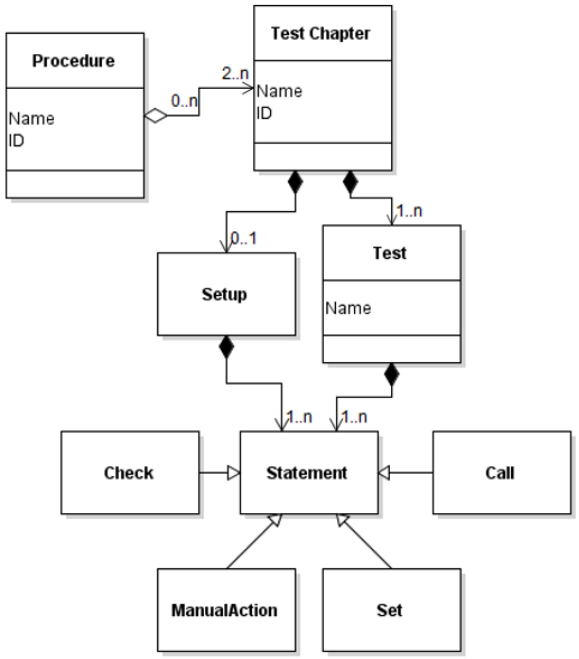
La structure du langage pivot reprend la structure des exemples de procédures proposés par les figures 3.3 et 3.4 : un bloc d'initialisation est lié à plusieurs blocs de test. Les blocs de test sont composés de trois types d'instructions : l'affectation, le *wait* et le *check*. Nous avons formalisé l'affectation par l'instruction *set* et n'autorisons que l'affectation d'un seul paramètre à la fois, pour faciliter la lisibilité et le débogage des tests avioniques. Plusieurs affectations peuvent être demandées en même temps dans les procédures exécutables afin de rendre les tests plus rapides à l'exécution. L'instruction *check during* vérifie l'état d'un ou plusieurs paramètres avioniques durant un certain temps. Les besoins couverts par la clause *WAIT* sur les exemples sont formalisés dans le langage par l'instruction *check until* associée à une contrainte de temps.

3.1.3 Grammaire des instructions du langage

Nous nous sommes adaptés au vocabulaire courant usité dans les tests d'intégration de l'ingénierie système. Notre partenaire du projet ACOVAS a aussi défini les termes "procédure de test", "chapitre de test", "bloc de test" et "bloc d'initialisation". Nous établissons dans ce qui suit le lien entre ces notions. Une procédure de test en langage pivot référence plusieurs chapitres de test. Un chapitre de test est composé d'un bloc d'initialisation optionnel et d'un ensemble de blocs de test. Il référence aussi les outils concourants à la

réalisation du test. Le bloc d'initialisation est optionnel car la validation de certains objectifs demande de vérifier le comportement du SUT dès lors qu'il est mis sous tension. Les instructions possibles dans ces blocs sont `set`, `check`, `manualAction` et `call`. Les procédures, les chapitres et les blocs de test sont désignés par un nom. La figure 3.5 présente le méta-modèle du langage pivot.

FIGURE 3.5 – Méta-modèle de la structure et des instructions du langage pivot



Le langage permet de manipuler les paramètres avioniques grâce aux instructions `set` et `check`. Les paramètres avioniques sont formalisés *via* un langage développé séparément pour des besoins de réutilisabilité. Les instructions `set` et `check` manipulent des références à ces paramètres.

L'instruction `set` est composée d'une référence à un paramètre avionique et de la valeur à assigner. L'instruction se termine par l'unité de mesure du paramètre manipulé. Cette instruction permet de gérer des paramètres entiers, booléens ou réels. Le type et l'unité d'un paramètre sont définis dans un fichier ICD décrit ultérieurement. Le listing 3.1 montre la grammaire EBNF de l'instruction `set`. Les concepts `valueFloat`, `valueInteger` et `valueBoolean` sont supportés par des types de données proposés par MPS. Les unités proposées par le langage sont amenées à évoluer en fonction des besoins. Le concept `IDparameter` est une référence au paramètre avionique identifié par le triplet nom de l'application (`applicationRef`), nom du bus (`busRef`) et nom du signal (`signalRef`). Les paramètres utilisés dans les instructions du langage pivot sont formalisés *via* le langage

`ICD` présenté à la section 3.1.5.

Listing 3.1 – Grammaire EBNF de l'instruction `set`

```
set := IDParameter, setEgal, parameterUnit
setEgal := setBoolean|setFloat|setInteger
setFloat := IDparameter, '=', valueFloat
setInteger := IDparameter, '=', valueInteger
setBoolean := IDparameter, '=', valueBoolean
IDparameter := applicationRef, busRef, signalRef
parameterUnit := 'kg/s'|'°C'|' '
```

L'instruction `check during` permet de vérifier que l'état (ou la valeur) d'un ou plusieurs paramètres avioniques reste constant. Cette instruction est composée d'une liste de paramètres et de valeurs attendues. Le listing 3.2 présente la grammaire EBNF de l'instruction `check during`. Le concept `during` est optionnel ; dans le cas où il n'est pas instancié, la vérification est ponctuelle, sans période de temps. On se retrouve alors dans le cas classique d'une assertion dans le contexte d'un test logiciel.

Listing 3.2 – Grammaire EBNF de l'instruction `check during`

```
check := 'Check that', *compareEgal, ?during
compareEgal := compareBoolean|compareFloat|compareInteger
compareFloat := IDparameter, '==', valueFloat, parameterUnit
compareInteger := IDparameter, '==', valueInteger, parameterUnit
compareBoolean := IDparameter, '==', valueBoolean, parameterUnit
parameterUnit := 'kg/s'|'°C'|' '
IDparameter := applicationRef, busRef, signalRef
during := 'during', valueInteger, timeUnit
timeUnit := 'sec'|'ms'|'min'
```

L'instruction `check until` du langage pivot donne la possibilité de s'assurer qu'un paramètre peut atteindre un certain état (ou une certaine valeur) avant un certain laps de temps ; c'était l'objectif de l'instruction *WAIT* présentée précédemment. Le listing 3.3 présente la grammaire EBNF de l'instruction `check until`. Le concept `until` définit le temps durant lequel le test attendra que le système atteigne les conditions expectées.

Listing 3.3 – Grammaire EBNF de l'instruction `check until`

```
check := 'Check that', *compareEgal, until
compareEgal := compareBoolean|compareFloat|compareInteger
compareFloat := IDparameter, '==', valueFloat, parameterUnit
compareInteger := IDparameter, '==', valueInteger, parameterUnit
compareBoolean := IDparameter, '==', valueBoolean, parameterUnit
parameterUnit := 'kg/s'|'°C'|' '
IDparameter := applicationRef, busRef, signalRef
until := 'until', valueInteger, timeUnit
timeUnit := 'sec'|'ms'|'min'
```

Lors de l'utilisation des instructions `set`, `check during` et `check until`, le testeur choisit le paramètre qu'il souhaite référencer. En fonction du type du paramètre (entier, flottant ou booléen), l'éditeur sélectionne automatiquement la bonne version du concept à instancier : `SetEgal` pour l'instruction `set` ou `CompareEgal` pour les instructions `check during` et `check until`.

Les figures 3.6 et 3.7 sont des exemples d'utilisation des instructions `set` et `check during`. La première montre deux exemples d'instruction `set` en langage pivot pour les consignes de la zone 1 et la zone 2. Nous avons choisi d'utiliser le symbole d'affectation classique plutôt que le mot-clé *set* pour la syntaxe concrète de cette instruction. La deuxième figure montre une instruction `check during` se rapportant aux deux zones. De même que pour l'instruction `set`, les symboles `==` remplacent dans la syntaxe concrète les appels aux différentes instructions de comparaison. Nous mettons ainsi en œuvre un polymorphisme pour les opérateurs d'affectation et de comparaison.

FIGURE 3.6 – Exemples d'instruction `set`

```
T consigne area1 = 18.0 °C
T consigne area2 = 18.0 °C
```

FIGURE 3.7 – Exemple d'instruction `check`

```
Check that T cabin area1 == 20. °C during 60 sec
            and
           T cabin area2 == 20. °C
```

L'instruction `call` fait référence à une fonctionnalité d'un outil théoriquement présent sur le banc de test pour aider à la mise en œuvre des procédures. Dans le cas où l'oracle de test n'est pas encore implémenté ou non implémentable pour le moment, ce type d'instruction permet de faire appel à un outil extérieur dans le but de récupérer une structure de données ou de tracer et loguer l'état du SUT pour une vérification manuelle ultérieure. La liste des outils qu'il est possible de référencer est définie grâce au langage `Tool`. Ce langage permet de formaliser les outils et de leur faire correspondre une liste de fonctions. Le méta-modèle de ce langage sera présenté à la section 3.1.5. Le choix d'un outil, lors de l'utilisation de l'instruction `call`, restreint les fonctions possibles à celles qu'il propose. Ce choix s'établit par le biais d'une liste des outils disponibles sur le banc.

Listing 3.4 – Grammaire EBNF des instructions `call` et `manualAction`

```
call := 'Call', toolRef, toolFunctionRef 'function'
manualAction := 'Manual Action: ', message
```

Le langage pivot supporte aussi certains types de tests semi-automatiques grâce à l'instruction `manualAction` qui permet de stopper l'exécution d'un test jusqu'à ce que le testeur

atteste que l'action a bien été réalisée. L'exécution de cette instruction fait apparaître une fenêtre *pop-up* contenant un message donné par le testeur lors de l'instanciation d'une action manuelle. L'utilisation de ce type d'action donne la possibilité d'intégrer, dans des procédures formalisées, des actions qui ne sont pas encore réalisables automatiquement. Par exemple, l'action de déconnecter un câble de données d'un calculateur n'est pas automatisable dans le cas des tests d'intégration IMA. De même, le pilotage de certains outils n'est pas automatisable car beaucoup d'entre eux ne proposent pas d'API permettant d'activer leurs fonctionnalités *via* un script. Le listing 3.4 présente la syntaxe EBNF des instructions `call` et `manualAction`.

3.1.4 Syntaxe concrète et éditeurs projectionnels

La conception des éditeurs du langage pivot permet d'agencer les concepts du langage au sein des interfaces d'édition de programmes. Les éditeurs projectionnels permettent de s'abstraire de la grammaire concrète du langage.

Le concept `TestChapter` est composé d'un nom géré par l'interface `INamedConcept`, d'une liste de références aux outils utilisés dans la procédure, d'un `setup` optionnel et d'une liste de blocs de test. Les sous-concepts sont rattachés à un concept grâce à la définition de sa structure. La figure 3.8 fournit un exemple de structure. Elle présente les propriétés (*properties*), les sous-concepts (*children*) et les références (*references*) du concept `TestChapter`. Les cardinalités des concepts doivent obligatoirement être renseignées pour les sections des sous-concepts et les références.

FIGURE 3.8 – Structure du concept `TestChapter`

```
concept TestChapter extends    BaseConcept
                    implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  setup    : Setup[0..1]
  testCase : Test[1..n]
  tools    : ToolRef[0..n]

  references:
  << ... >>
```

La figure 3.9 montre la formalisation de l'éditeur projectionnel MPS liée au concept `TestChapter`. L'éditeur MPS dédié à un concept établit le lien entre les mots-clés du lan-

gage et ses sous-concepts (entre `%` ou `{ }` pour l'attribut `name`). On y retrouve les concepts `%tools%` correspondant à la liste des outils référencés dans le chapitre (`ToolRef`), `%setup%` étant optionnel et `%testCase%` représentant la liste des blocs de test (`Test`) conformément à la structure d'un chapitre. Dès lors que la cardinalité du sous-concept est autre que 0..1 ou 1, il prendra la forme d'une liste dans l'éditeur. Les sous-concepts des éditeurs entre '`(/`' et '`/)`' correspondent à la création d'une liste dans l'éditeur. Un emplacement vide est alors prévu dans chacune de ces listes pour permettre l'ajout d'un nouveau concept instancié dans la liste. Cet espace vide est formalisé par la cellule `/empty cell:` suivie d'une cellule `<default>`. Il est possible de créer des espaces vides dans les éditeurs en ajoutant une cellule ne contenant aucun concept. Les cellules vides sont représentées par le texte `<constant>`. Ces cellules sont importantes pour obtenir une présentation des concepts aérée et donc plus agréable à utiliser dans l'éditeur de l'utilisateur.

FIGURE 3.9 – Editeur du concept `TestChapter`

```
node cell layout:
[/
    [> Chapter: { name } <]
    <constant>
    Tools:
    (/ % tools % /)
        /empty cell: <default>
    <constant>
    % setup %
    <constant>
    (/ % testCase % /)
        /empty cell: <default>
/]
```

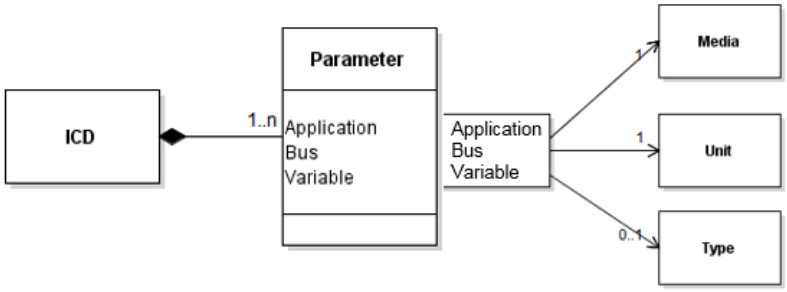
Les éditeurs définissent la syntaxe concrète des concepts d'un langage. Les mots-clés du langage sont présentés en gras dans les éditeurs. Ces éditeurs fixent la disposition des concepts dans les éditeurs de programmes. Cette disposition ne peut pas être modifiée par les utilisateurs du langage ; seuls les éditeurs projectionnels associés au langage peuvent mettre à jour la disposition des concepts dans les interfaces d'édition des programmes.

3.1.5 Formalisation d'un ICD et des outils externes

Les besoins en formalisation de tests avioniques font apparaître certains concepts communs à une partie ou à l'ensemble des métiers impliqués dans la création d'un avion. Ces concepts sont la liste des paramètres avioniques des ICD appartenant au système que l'on souhaite tester et les outils externes nécessaires à la récupération de résultats validant un test. Ces outils sont pilotables grâce à la procédure de test *via* des scripts et des API. Un fichier ICD pourra aussi contenir des paramètres permettant d'accéder à des fonctionnalités du banc de test pour gérer son démarrage, son redémarrage, sa mise en pause ou son arrêt

par exemple.

FIGURE 3.10 – Meta-modèle du langage `ICD`



Un fichier ICD est composé des paramètres d'entrée et de sortie d'un système. Un nom unique, composé de trois parties, identifie chacun de ces paramètres. Changer la valeur d'un paramètre en entrée ou récupérer et vérifier la valeur d'un paramètre en sortie, nécessite de référencer l'identifiant du paramètre. Manipuler des paramètres avioniques dans le contexte de l'utilisation d'un langage informatique demande donc préalablement de formaliser ces paramètres. Pour ce faire, nous avons créé un concept `ICD` comme le nœud racine de l'AST du langage ICD. La définition de ce concept et ses sous-concepts rattachés est indépendante des autres concepts du langage. Les informations qui composent un paramètre avionique donneront lieu à autant de sous-concepts du paramètre. L'organisation hiérarchique du concept `ICD`, des paramètres avioniques et des informations qui les caractérisent, forment le méta-modèle du langage ICD. Ce méta-modèle est présenté à la figure 3.10.

FIGURE 3.11 – Exemple de paramètres ICD formalisés

```
ICD:
Application Name: T Bus Name: cabin Variable name: area1 type: integer unit: °C Media: MODEL
Application Name: T Bus Name: cabin Variable name: area2 type: integer unit: °C Media: MODEL
Application Name: T Bus Name: setpoint Variable name: area1 type: integer unit: °C Media: MODEL
Application Name: T Bus Name: setpoint Variable name: area2 type: integer unit: °C Media: MODEL
Application Name: Q Bus Name: setpoint_Area1 Variable name: duct1 type: Float unit: kg/s Media: MODEL
Application Name: Q Bus Name: setpoint_Area1 Variable name: duct2 type: Float unit: kg/s Media: MODEL
Application Name: Q Bus Name: setpoint_Area2 Variable name: duct1 type: Float unit: kg/s Media: MODEL
Application Name: Q Bus Name: setpoint_Area2 Variable name: duct2 type: Float unit: kg/s Media: MODEL
Application Name: Model Bus Name: environment Variable name: Kp1 type: integer unit:   Media: MODEL
Application Name: Model Bus Name: environment Variable name: ki1 type: integer unit:   Media: MODEL
Application Name: Model Bus Name: environment Variable name: Kd1 type: integer unit:   Media: MODEL
Application Name: Default Bus Name: flow Variable name: Duct1 type: Boolean unit:   Media: MODEL
```

La figure 3.11 propose un extrait d'une liste des paramètres composant un ICD dans l'outil MPS. Chaque ligne correspond à un paramètre différent. Un paramètre est composé d'un triplet application-bus-variable qui l'identifie de manière unique, d'un type (booléen, entier, flottant ...), d'une unité optionnelle et d'un média de communication. Le type,

l'unité et le média de communication d'un paramètre sont spécifiés par l'utilisateur en sélectionnant le choix adéquat dans la liste des choix possibles. Les choix possibles sont définis par une énumération MPS. Le langage ICD définit les interfaces des systèmes avioniques indépendamment des langages de test.

FIGURE 3.12 – Meta-modèle du langage `Tools`

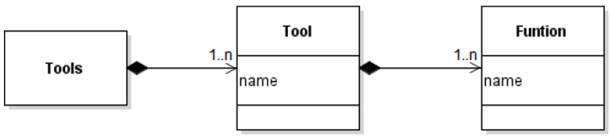


FIGURE 3.13 – Exemple d'une liste d'outils formalisée avec le langage `Tools`

```
Tools:
Tool: DLCS
 function: Start dataloading
 function: Abort dataloading
Tool: PIT
 function: Start trace
 function: Stop trace
Tool: Wireshark
 function: Start trace
 function: Stop trace
Tool: CMS
 function: No FMC raised
Tool: FSA-NG
 function: Stop SSP emission
Tool: SIS reader
 function: Stop trace
 function: Start trace
```

Les outils externes au test sont prépondérants dans certains domaines métier avioniques. Pour les domaines métier où les tests demeurent manuels, les outils utilisés ne sont pas tous pilotables par des scripts informatiques. Il faudra alors prévoir de les remplacer par des outils pilotables automatiquement, proposant les mêmes fonctionnalités, ou de les faire évoluer pour qu'ils le deviennent.

Pour permettre le pilotage de ces outils, la formalisation se base sur une liste d'outils caractérisés par leur nom. A chaque outil est rattachée la liste des fonctionnalités qu'il propose. Les fonctionnalités sont elles aussi identifiées par un nom unique. La formalisation des outils nécessite la conception d'un langage dédié, nommé le langage `Tools`, de la même manière que les paramètres de l'ICD. Le méta-modèle de la figure 3.12 montre la structure des concepts du langage `Tools`. La figure 3.13 illustre une liste d'outils et des fonctions associées formalisées avec le langage `Tools`.

3.1.6 Sémantique des instructions du langage

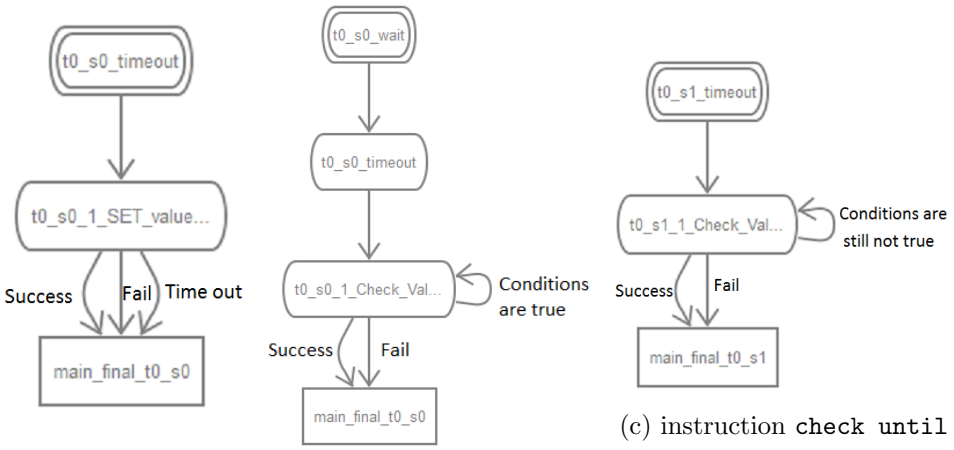
Nous proposons des patrons de transformation génériques de compilation des différentes instructions du langage pivot en code exécutable. Ces patrons ont été élaborés en collaboration avec l'ingénieur R&D d'Airbus responsable de la mise en place des tests automatisés en langage SCXML (State Chart XML). Ils montrent la sémantique opérationnelle des instructions du langage pivot sur les bancs de test et sont décrits par des automates.

La figure 3.14a correspond à l'automate de l'instruction `set`. L'automate est composé de trois états : le premier état initialise un *time out*, l'état suivant réalise l'action d'affectation et l'état final établit le lien avec l'instruction suivante du test. L'action d'affectation demande de récupérer une référence du paramètre avionique. C'est grâce à cette référence que l'affectation de la valeur attendue est réalisée. Les paramètres doivent être préalablement formalisés grâce au langage `ICD` que nous proposons. Il existe trois transitions de sortie de l'état principal correspondant à trois cas d'exécution de l'instruction. Le premier cas intervient lorsque l'affectation a été exécutée avec succès, le deuxième quand une erreur survient lors de l'exécution et le troisième cas fait intervenir un *time out* lorsque l'action d'affectation ne répond pas dans le délai imparti. Le *time out* est nécessaire car un test pourrait être bloqué dans cet état dans le cas où l'action d'affectation ne fournirait pas de retour. Le *timer* mettant en place le *time out* est démarré dans l'état initial. Un *log* gardant une trace du résultat de l'exécution d'une instruction est produit lors de l'activation d'une transition dans l'automate (*Success*, *Fail* ou *Time out*).

L'instruction `check during` décrit à la figure 3.14b garantit qu'un état est constant pendant une période donnée. Le premier état produit une attente de 20 ms pour s'assurer que le système a pu réagir aux précédentes actions. L'état suivant réalise la récupération des valeurs des paramètres à vérifier et les compare aux valeurs attendues. Cette vérification doit montrer que les conditions sont respectées durant toute la durée du `check`. Deux transitions pour sortir de cet état sont possibles. Si les conditions ne sont pas respectées, la transition *Fail* sera activée. Si les conditions sont toujours respectées à l'issue du délai spécifié par l'instruction, alors la transition *Success* sera activée.

L'instruction `check until` s'assure qu'un ou plusieurs paramètres peuvent atteindre une valeur attendue qui leur sera rattachée. Cette instruction est présentée à la figure 3.14c. Tout comme l'instruction `set`, elle est composée de trois états. Le premier état met en place un *time out*. Le second récupère la valeur des paramètres à vérifier et les compare aux valeurs attendues, tout comme l'instruction `check during`. Tant que les conditions ne sont pas respectées, cette vérification intervient toutes les 200 ms pendant la durée du `check`. Deux transitions sont issues de cet état. La première est activée lorsque l'évènement *time out* est levé, c'est-à-dire quand les conditions n'ont pas été atteintes pendant la durée déterminée par l'instruction `check` (*Fail*). La deuxième est activée lorsque les conditions du `check` sont atteintes (*Success*).

FIGURE 3.14 – Automates d'exécution des instructions du langage pivot



(a) instruction `set`

(b) instruction `check during`

(c) instruction `check until`

3.2 Le DSTL ATA 21

Nous avons conscience que l'utilisation d'un langage informatique pour les experts en intégration de systèmes avioniques n'est pas triviale. Les langages composant la famille des DSTL proposée par le *framework* BDD s'attachent à être simples d'utilisation. Ces DSTL sont dits de haut niveau car leur syntaxe est très proche du langage naturel permettant aux tests d'être compréhensibles par toutes les parties prenantes à la réalisation d'un avion.

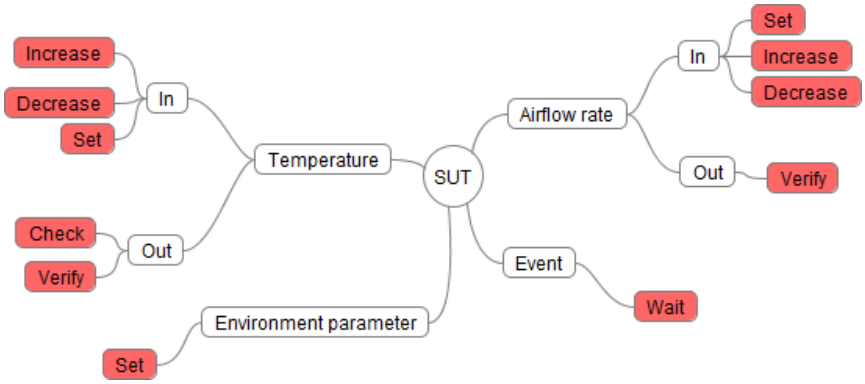
Lors de cette expérimentation, nous avons conçu un langage de haut niveau pour formaliser les tests d'intégration des systèmes du conditionnement de l'air dans un avion (ATA 21). Ce langage est nommé DSTL ATA 21 ; il a été présenté lors de la démonstration finale du projet ACOVAS, montrant la formalisation et l'exécution de deux procédures. La conception de ce langage s'est faite à partir des mêmes exemples de procédures textuelles que le langage pivot.

Le but du DSTL ATA 21 est de proposer un jeu d'instructions répondant aux problématiques du domaine tout en gardant une syntaxe proche du langage naturel. Les cas de test produits valideront le comportement du système lorsque les fonctions de régulation de la température seront sollicitées. Chaque instruction est composée de mots-clés formant une phrase qui décrit un but unique. Le code produit grâce à ce langage décrira simplement l'intention de chaque cas de test. Les instructions doivent permettre d'augmenter ou de diminuer les valeurs de débit d'air des canaux ou la consigne de température. Elles doivent aussi vérifier si la température réelle d'une zone est atteinte ou si cette température est constante pendant un certain temps.

3.2.1 Conception du langage ATA 21

Le but du langage ATA 21 est de produire des cas de test simples et compréhensibles, où l'intention de chaque test est clairement décrite. L'utilisation des identifiants réels des paramètres ICD rend complexe la lisibilité et la compréhension des procédures de test de haut niveau. Nous avons estimé qu'il est préférable de nommer ces variables différemment pour masquer la complexité du nommage des triplets (application, bus, variable) correspondant aux paramètres de l'ICD. Pour ce faire, nous avons implémenté des mécanismes d'*alias* des paramètres réels pour simplifier le nommage des paramètres avioniques et par la même occasion, la compréhension des procédures de test.

FIGURE 3.15 – Carte conceptuelle du DSTL ATA 21



La conception des instructions du langage s'est faite grâce à la carte conceptuelle de la figure 4.13, nous aidant à communiquer avec les experts sur les besoins de formalisation de l'ATA 21. Les instructions visent à réaliser des cas de test portant sur la validation des fonctionnalités logicielles pour le conditionnement de l'air et des calculateurs les hébergeant dans l'avion. L'installation de fonctions logicielles sur un calculateur forme un système qui, lors du test, agit sur le modèle de simulation de l'environnement. Ce modèle simule la variation de la température induite par le système sous test dans les zones de l'avion. Cette carte classifie les instructions proposées en quatre catégories : *Temperature, Airflow rate, Environnement parameter* et*Event*.

Dans l'objectif de mettre en exergue l'intention des tests, toute instruction du DSTL ATA 21 doit proposer une syntaxe décrivant réellement son objectif de test. La combinaison de ces instructions pour former les cas de test s'apparente alors à des phrases en langage naturel. Le texte sera formé d'une séquence d'actions agissant sur le SUT ou sur le modèle d'environnement. Ces actions permettent aussi de vérifier l'état et le comportement du SUT durant le test.

La carte conceptuelle montre que nous souhaitons proposer plusieurs instructions d'affectation d'une valeur à un paramètre avionique ou de simulation. Ces instructions donneront la possibilité aux utilisateurs de changer la valeur d'un paramètre d'entrée (*In*) du

système et de notifier si l'intention du test est d'augmenter (`increase`) ou de diminuer (`decrease`) la valeur du paramètre en question. Nous proposons aussi des instructions différentes pour notifier si cette augmentation ou diminution impacte des températures ou des débits d'air fournis par les canaux. De plus, pour augmenter l'expressivité des instructions, nous remplaçons le symbole égal de l'affectation par un groupe de mots-clés plus proches du langage naturel. Pour une hausse de la valeur, les mots-clés `up to` seront présents avant la valeur ; de même pour une diminution avec `down to`. Lorsque la syntaxe concrète du langage ressemble à une phrase en langage naturel, le test est plus compréhensible et lisible par toutes les parties prenantes et non simplement par les programmeurs. Voici un exemple d'instruction qui augmente la température de la zone `B02` afin qu'elle atteigne 21,2 °C :

```
Increase the temperature of Area_B02 up to 21.2 °C
```

L'instruction `Set` de la catégorie *Airflow rate* permet l'initialisation des débits d'air en fonction de l'ouverture réelle des vannes d'alimentation en air. Cette instruction n'est utile que lors de la première affectation du débit d'air pour une vanne. Par la suite, les changements seront notifiés par les instructions `Increase` et `Decrease`. Il en est de même pour l'instruction `Set` de la catégorie *Temperature*.

Il est aussi possible de modifier directement la température simulée d'une zone de l'avion en influant sur le modèle de simulation de l'environnement grâce à une instruction `Set`. L'instruction `Set` de la catégorie *Environment parameter* permet de modifier l'état de n'importe quel paramètre du modèle de simulation. Cette instruction a pour but d'initialiser les conditions d'un test, sans avoir besoin d'attendre la réponse du système testé. Un paramètre simulé ne doit pas être modifié par une instruction une fois initialisé, car il s'agit de vérifier que ce paramètre est correctement régulé par le système testé.

Dans les cas où le système entre en défaillance, il doit pouvoir émettre un signal. L'instruction `Wait` de la catégorie *Event* rend possible la formalisation de l'attente d'un tel signal.

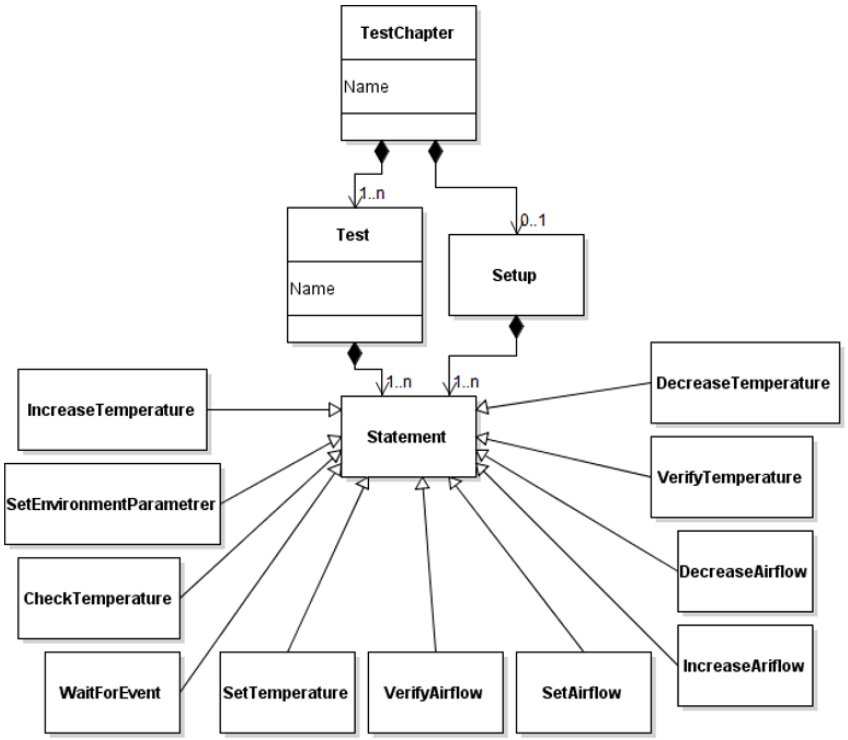
Il est possible de vérifier l'état des paramètres de sortie grâce aux instructions `Check` et `Verify` associées aux catégories *Temperature* et *Airflow rate*. Ces instructions proposent deux types d'assertions : les assertions `Check` s'assurent qu'un état du système peut être constant et les assertions `Verify` font de même vis-à-vis des conditions pouvant être atteintes par le système. Il n'a pas été jugé pertinent de proposer un `Check` pour la catégorie *Airflow rate* car les valeurs du débit d'air injecté dans une zone sont ajustées en permanence par le système.

Chacun de ces cas possibles est incarné par une instruction spécifique du langage afin de limiter la complexité combinatoire induite par des instructions paramétrisables couvrant un nombre important de cas possibles.

3.2.2 Grammaire du DSTL ATA 21

La figure 3.16 met en exergue les principaux concepts du DSTL dédié à l'ATA 21. Ce méta-modèle a pour concept racine un chapitre de test (`TestChapter`). Un chapitre de test est composé d'un bloc d'initialisation optionnel (`Setup`) et d'au moins un bloc de test (`Test`). Les blocs de test et d'initialisation sont peuplés par des instructions (`Statement`).

FIGURE 3.16 – Méta-modèle de la structure et des instructions du langage ATA 21



Le méta-modèle montre aussi les onze instructions proposées. L'instruction `WaitForEvent` permet d'attendre qu'un signal booléen change d'état (de 0 à 1 ou de 1 à 0). La température d'entrée est mise à jour par les actions `IncreaseTemperature`, `DecreaseTemperature` et `SetTemperature`. L'instruction `Set` permet d'initialiser la consigne de température d'une zone ou de forcer l'initialisation de la température d'un paramètre de sortie lors de l'utilisation d'un modèle comportemental du système. Les instructions `IncreaseAirflow` et `DecreaseAirflow` font varier le débit d'air dans les canaux d'alimentation et l'instruction `SetAirflow` l'initialise. Modifier la température est mis en œuvre par l'instruction `SetTemperature`.

Le langage ne propose pas de règles sémantiques pour assurer la cohérence des cas de test. Ces règles peuvent par exemple vérifier qu'une instruction dont l'intention est de

diminuer (respectivement d'augmenter) une température ou un débit d'air affecte bien une valeur inférieure (respectivement supérieure) à la valeur actuelle du paramètre. Ces règles auraient pour but de valider sémantiquement les tests formalisés. Dans le but de proposer une version plus complète du DSTL ATA 21, des règles de vérification de ce type pourront faire l'objet de travaux futurs.

Les paramètres de simulation de l'environnement peuvent également être modifiés par l'instruction `SetEnvironmentParametrer`.

La température effective des zones peut être vérifiée par les deux instructions `CheckTemperature` et `VerifyTemperature`. L'instruction `VerifyTemperature` s'assure qu'une valeur est atteinte dans une zone avant un certain laps de temps, alors que l'instruction `CheckTemperature` garantit que la valeur de la température d'une zone est constante (*is equal to*) sur une période donnée (*while*).

Listing 3.5 – Grammaire des instructions du DSTL ATA 21

```
increaseTemperature := 'Increase the temperature of', variable,
        'up to', value, '°c';
decreaseTemperature := 'Decrease the temperature of', variable,
        'down to', value, '°c';
setTemperature := 'Set temperature to', value, 'for' variable;
checkTemperature := 'Check that temperature of',
        *(variable, 'is equal to', value,'°c'), ?while;
verifyTemperature := 'Verify that temperature of',
        *(variable, 'has reached', value, '°c'), ?until;
decreaseAirflow := 'Decrease the air flow rate of', variable,
        'down to', value, 'kg/s';
increaseAriflow := 'Increase the air flow rate of', variable,
        'up to', value, 'kg/s';
setAirflow := 'Set air flow rate to', value, 'kg/s for', variable;
verifyAirflow := 'Verify that air flow rate of',
        *(variable, 'has reached', value, 'kg/s'), ?until;
setEnvironmentParametrer := 'Set', variable, 'to', value, 'for', variable;
waitForEvent := 'Wait for', variable, ?until
value :=  *digit, '.' *digit
digit := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
until := 'until', *digit, unit
while := 'while', *digit, unit
unit := 'ms'|'s'|'min'
```

L'instruction `VerifyAirflow` vérifie que le débit d'air est bien celui attendu. Les instructions de vérification `VerifyTemperature` et `VerifyAirflow` valident qu'un état donné du système est atteint (*has reached*) avant un certain laps de temps (*until*). Le concept `until` est optionnel. Toutes ces instructions sont décrites par le méta-modèle de la figure 3.16.

Le listing 3.5 résume l'ensemble des instructions du DSTL ATA 21.

La table 3.1 montre la correspondance entre les concepts du DSTL ATA 21 et ceux du langage pivot. Un chapitre de test en DSTL est traduit en langage pivot par le concept

TABLE 3.1 – Table de correspondance entres les concepts du DSTL et ceux du pivot

Concepts du DSTL ATA 21	**Concepts du langage pivot**
TestChapter	TestChapter
Setup	Setup
Test	Test
DecreaseTemperature	set*
IncreaseTemperature	set*
SetTemperature	set*
DecreaseAirflow	set*
IncreaseAirflow	set*
SetAirflow	set*
SetEnvironmentParameter	set*
VerifyAirflow	check until
CheckTemperature	check during
VerifyTemperature	check until
WaitForEvent	check until

`TestChapter`. Un ensemble d'instructions `Set` du langage pivot traduisent les instructions d'affectation (`Increase`, `Decrease` et `Set`). Les assertions correspondent soit à l'instruction `check until`, soit à l'instruction `check during`.

3.2.3 Mécanisme d'*alias* des paramètres avioniques

Les paramètres avioniques de même type ayant le même comportement pour un test donné sont mis en correspondance avec un seul paramètre de haut niveau. Ce mécanisme est utilisé pour masquer la complexité de nommage des paramètres avioniques et de les rendre plus compréhensibles lors de leur utilisation dans les instructions de test. Son but est aussi de fournir la possibilité de traiter les paramètres avioniques par lot.

FIGURE 3.17 – Correspondance des paramètres ICD

```
Mapping:
Name: Expected_Temperature_Of_Area2 Params: T setpoint area2
Name: Expected_Temperature_Of_Area1 Params: T setpoint area1
Name: Ducts_Of_Area1 Params: Q setpoint_Area1 duct1 and Q setpoint_Area1 duct2
Name: Temperature_Area1 Params: T cabin area1
Name: Temperature_Area2 Params: T cabin area2
Name: Ducts_Of_Area2 Params: Q setpoint_Area2 duct1 and Q setpoint_Area2 duct2
```

La figure 3.17 montre comment le mécanisme d'*alias* des paramètres avioniques réels est réalisé. Ce mécanisme fait correspondre à un paramètre de haut niveau un ou plusieurs paramètres avioniques réels. Chaque ligne de ce fichier déclare un paramètre de haut

niveau référençable par les instructions du DSTL ATA 21. Ce paramètre est défini par un nom, renseigné après le mot-clé `Name`, et par la liste des paramètres avioniques auquel il correspond, après le mot-clé `Params`. Les paramètres avioniques réels de cette liste sont formalisés par le code présenté à la figure 3.11 et sont séparés par le mot-clé `and`. Par exemple, les canaux d'alimentation en air de la zone 1 sont rassemblés sous la même variable `Ducts_Of_Area1`, de même pour la zone 2 avec `Ducts_Of_Area2`. Les consignes de température pour la zone 1 et 2 sont elles renommées en `Expected_Temperature_Of_Area1` et `Expected_Temperature_Of_Area2` et masquent la complexité de nommage des paramètres réels, respectivement `T setpoint area1` et `T setpoint area2`.

3.2.4 Exemple de cas de test

Cet exemple correspond au test exécuté lors de la démonstration finale du projet ACOVAS. Il utilise la correspondance entre les variables de test et les paramètres avioniques réels spécifiés à la figure 3.17 de la section 3.2.3. La figure 3.18 exhibe des cas de test en DSTL ATA 21 visant à vérifier qu'une température peut être atteinte en un certain laps de temps, lorsque le débit d'air envoyé par les alimentations en air est fixé à une valeur prédéfinie. Dans le premier cas de test, le débit est fixé à 0,2 kg/s et il est de 0,3 kg/s pour le deuxième.

Le bloc `setup` vise à initialiser la valeur demandée dans les deux zones étudiées à 18 °C et à initialiser le débit d'air dans les canaux d'alimentation de ces zones à 0,0 kg/s. Le premier cas de test demande une augmentation de température à 20 °C pour les deux zones, puis ouvre les canaux d'alimentation à 0,2 kg/s. Le test doit ensuite s'assurer que la consigne est bien atteinte dans les deux zones avant 10 minutes et que cette température est constante durant une minute. Par la suite, la consigne demandée est de 21 °C pour la zone 2 uniquement. Le test exige alors que la température dans la zone 2 soit bien atteinte avant 5 minutes, tout en garantissant une température de 20 °C pour la zone 1. Le comportement du système est validé lorsque ces valeurs sont constantes dans les deux zones pendant une minute. Ce premier cas de test permet de vérifier que le système réagit bien aux consignes de température demandées et que la variation de la température d'une zone n'a pas d'impact sur la température de l'autre.

Le deuxième cas de test répète le même schéma en changeant la valeur du débit d'air fourni aux deux zones de 0,2 kg/s à 0,3 kg/s. Entre l'exécution de chaque cas de test, le bloc `setup` est rejoué pour remettre le système dans les conditions initiales. On peut remarquer que le délai accordé au système pour s'adapter aux nouvelles consignes de température est moins important dans le deuxième cas de test vu que l'ouverture des vannes est plus conséquente.

FIGURE 3.18 – Cas de test formalisé en DSTL ATA 21

```
Procedure: Airflow influence with different airflow rates

setup:
Set temperature to 18.0 °c for  Temperature_Area1
Set temperature to 18.0 °c for  Temperature_Area2
Set air flow rate to 0.0 kg/s for Ducts_Of_Area1
Set air flow rate to 0.0 kg/s for Ducts_Of_Area2
Set TestParam to 1

Test: with_ducts_open_to_2
Increase the temperature of  Expected_Temperature_Of_Area2 up to  20. °c
Increase the temperature of  Expected_Temperature_Of_Area1 up to  20. °c
Increase the air flow rate of  Ducts_Of_Area1 up to 0.2 kg/s
Increase the air flow rate of  Ducts_Of_Area2 up to 0.2 kg/s
Verify that temperature of Temperature_Area1 has reached 20. °c  until 600 sec
                            and
                           Temperature_Area2 has reached 20. °c
Check that temperature of Temperature_Area1 is equal to 20. °c  while 60 sec
                            and
                           Temperature_Area2 is equal to 20. °c
Decrease the temperature of Expected_Temperature_Of_Area2 down to 21.0 °c
Verify that temperature of Temperature_Area1 has reached 20. °c  until 300 sec
                            and
                           Temperature_Area2 has reached 21. °c
Check that temperature of Temperature_Area1 is equal to 20. °c  while 60 sec
                            and
                           Temperature_Area2 is equal to 21. °c

Test: with_ducts_open_to_3
Increase the temperature of  Expected_Temperature_Of_Area2 up to  20. °c
Increase the temperature of  Expected_Temperature_Of_Area1 up to  20. °c
Increase the air flow rate of  Ducts_Of_Area1 up to 0.3 kg/s
Increase the air flow rate of  Ducts_Of_Area2 up to 0.3 kg/s
Verify that temperature of Temperature_Area1 has reached 20. °c  until 500 sec
                            and
                           Temperature_Area2 has reached 20. °c
Check that temperature of Temperature_Area1 is equal to 20. °c  while 60 sec
                            and
                           Temperature_Area2 is equal to 20. °c
Increase the temperature of  Expected_Temperature_Of_Area2 up to  21.0 °c
Verify that temperature of Temperature_Area1 has reached 20. °c  until 180 sec
                            and
                           Temperature_Area2 has reached 21. °c
Check that temperature of Temperature_Area1 is equal to 20. °c  while 60 sec
                            and
                           Temperature_Area2 is equal to 21. °c
```

3.3 Chaîne de transformations du *framework*

Nous présentons dans cette section les différentes transformations proposées par le *framework* BDD que nous avons développé. Ces transformations sont de deux types et chaque type de transformation intervient à un moment différent dans la chaîne de transformations. La première transformation a pour but de générer du code en langage pivot à partir d'un code en DSTL. La deuxième transformation génère du code exécutable à partir du code en langage pivot.

Le langage pivot propose deux cibles de génération qui sont SCXML et Python. Les patrons présentés à la section 3.1.6 nous ont guidés dans l'optique de fournir le même comportement d'exécution quel que soit le langage d'exécution cible.

Les langages DSTL ont tous pour cible le langage pivot. Un exemple de transformation est donné après avoir expliqué les règles de transformation d'un code DSTL ATA 21.

3.3.1 Du langage DSTL ATA 21 vers le langage pivot

La génération de code depuis un langage de haut niveau vers le langage pivot est une génération de modèle à modèle. La transformation de modèle à modèle traduit un modèle source, c'est-à-dire l'AST d'un programme en DSTL dans notre cas, en un modèle cible correspondant à un AST en langage pivot. L'élaboration du moteur de transformation se fait directement sur le méta-modèle. La transformation se base sur un ensemble de règles décrivant comment transformer un ou plusieurs concepts appartenant au méta-modèle du langage source vers un ou plusieurs concepts du méta-modèle du langage cible.

Cette transformation est la première étape de la chaîne de transformations proposée par notre *framework*. Elle a pour but de faire correspondre aux concepts du langage de haut niveau un ou plusieurs concepts du langage pivot. Avec l'outil MPS, les règles de transformation de modèle à modèle sont décrites par des *templates* présentés ci-dessous. L'outil MPS permet d'obtenir directement l'AST d'un programme en DSTL ATA 21 à traduire et génère l'AST correspondant en langage pivot grâce aux règles de transformation associées aux concepts du méta-modèle.

Un chapitre de test décrit en DSTL ATA 21 est composé d'un bloc `setup` optionnel et d'une liste de blocs de test. La figure 3.19 est un chapitre de test en langage pivot servant de *template* à la génération du concept `TestChapter` du DSTL. Le concept `TestChapter` du DSTL étant un concept racine du méta-modèle, le *template* correspondant à ce concept est un *root template*. Il sert de règle de base pour la génération du code pour cette transformation. Ce *template* change le nom du chapitre (dans l'exemple `procedure`) par le nom réel du chapitre en DSTL en cours de génération. Dans les blocs `Setup` et `Test`, le *template* parcourt les instructions du chapitre et génère la ou les instructions correspondantes en langage pivot.

La génération des instructions en fonction du type d'instruction DSTL est réalisée par le *switch* `InstructionsSwitch`. La majeure partie du code implémentant ce *switch* est

FIGURE 3.19 – *Template* général de transformation d'un chapitre de test DSTL en langage pivot

```
root template
input Procedure

Chapter: $[procedure]

Tools:
<< ... >>

$IF$[Setup:
  $LOOP$[$SWITCH$ InstructionsSwitch[% Instructions]]]

$LOOP$[Test: $[Test]
  $LOOP$[$SWITCH$ InstructionsSwitch[% Instructions]]]
```

présentée à la figure 3.20 ; il est composé des onze types d'instructions composant le langage DSTL et fait correspondre à chacun d'eux le *template* de génération correspondant. Ces règles de transformation se basent sur la table de correspondance 3.1.

La figure 3.21 correspond au *template* transformant l'instruction `DecreaseTemperature` en une liste d'instructions `Set` en langage pivot. Il fait correspondre le nom du *template* au le nom du type d'instruction du DSTL ATA 21 auquel il se rattache. Le lien entre le *template* et l'instruction du DSTL concernée est réalisé par l'attribut `input`. Le lien avec le code généré est décrit par l'attribut `content node`. Ce *template* produit une instruction `Set` en langage pivot pour chaque paramètre désigné par la variable de haut niveau utilisée dans l'instruction en DSTL. Un paramètre est identifié dans le code en langage pivot par des références aux noms de l'application, du bus et du signal qui le caractérisent. Le code entre les symboles `->$[` et `]` remplace le nom du paramètre par la référence au nom réel du paramètre manipulé par le code généré.

Le *template* de l'instruction `CheckTemperature` correspond à une instruction `check during` en langage pivot ; son *template* de transformation est donné à la figure 3.22. L'instruction `check` manipule tous les paramètres regroupés grâce à la variable de test de haut niveau. Les instructions de type `Verify` du DSTL sont traduites en instructions `check until`, comme le montre le deuxième *template* de la figure 3.22. Nous manipulons dans ces `templates` les concepts du langage pivot à travers ses éditeurs projectionnels.

La notion d'attente en langage pivot est supportée par l'instruction `check until`. L'instruction `WaitForEvent` attend qu'un signal soit émis par le système. Le troisième *template* de la figure 3.22 donne la traduction de l'instruction `WaitForEvent` en langage pivot.

FIGURE 3.20 – Génération des instructions du langage DSTL

```
cases:

    [concept    VerifyAirflow]      --> verifyAirflow(node.value)
    [inheritors true]
    [condition  <always>]

    [concept    DecreaseTemperature] --> DecreaseTemperature(node.value)
    [inheritors true]
    [condition  <always>]

    [concept    DecreaseAirflow]    --> DecreaseAirflow(node.value)
    [inheritors true]
    [condition  <always>]

    [concept    IncreaseAirflow]    --> increaseAirflow(node.value)
    [inheritors true]
    [condition  <always>]

    [concept    IncreaseTemperature] --> increaseTemperature(node.value)
    [inheritors true]
    [condition  <always>]

    [concept    SetEnvironmentParam] --> setEnvParam(node.value + "")
    [inheritors true]
    [condition  <always>]

    [concept    VerifyTemperature]  --> VerifyTemperature(  (genContext, node, operationContext)->list<string> {  )
    [inheritors true]                                         list<string> values = new arraylist<string>;
    [condition  <always>]                                     foreach vals in node.hasReached {
                                                                values.add(vals.value);
                                                              }
                                                              return values;
                                                            }

    [concept    CheckTemperature]   --> CheckTemperature(  (genContext, node, operationContext)->list<string> {  )
    [inheritors true]                                       list<string> values = new arraylist<string>;
    [condition  <always>]                                   foreach vals in node.isEqualTo {
                                                              values.add(vals.value);
                                                            }
                                                            return values;
                                                          }
```

FIGURE 3.21 – *Template* de transformation de l'instruction `DecreaseTemperature`

```
template DecreaseTemperature
input    DecreaseTemperature

parameters
value : string

content node:
    [$LOOP$[->$[model_control_I] ->$[consigne_1] ->$[consigne_1] = $[30.9] Celsius ]]
```

FIGURE 3.22 – *Templates* de transformation des instructions `CheckTemperature`, `VerifyTemperature` et `WaitForEvent`

```
template CheckTemperature
input    CheckTemperature

parameters
values : list<string>

content node:
    [ Check that $LOOP$[$LOOP$[->$[model_control_I] ->$[Kd1] ->$[Kd1] == $[0.2] ]] $IF$[during $[200] $[sec] ]]


template VerifyTemperature
input    VerifyTemperature

parameters
values : list<string>

content node:
    [ Check that $LOOP$[$LOOP$[->$[model_control_I] ->$[Kd1] ->$[Kd1] == $[2.] ]] $IF$[until $[200] $[sec] ]]

template WaitForEvent
input    waitForEvent

parameters
<< ... >>

content node:
    [ Check that $LOOP$[->$[model_control_I] ->$[in1] ->$[in1] == false  ] $IF$[until $[200] $[sec] ]]
```

3.3.2 Exemple de transformation en langage pivot

A partir des cas de test de la figure 3.18, le *framework* génère du code en langage pivot. La correspondance entre les paramètres avioniques réels et les variables de test utilisées dans le code en DSTL est présentée à la figure 3.17. Grâce à cette correspondance, les variables de test de haut niveau utilisées dans le code DSTL se traduisent en paramètres avioniques réels dans le langage pivot. Les instructions DSTL utilisant des variables de test regroupant plusieurs paramètres avioniques réels s'expriment en langage pivot par un `Set` pour chaque paramètre réel.

Les instructions `Check` et `Verify` du DSTL sont transformées en instructions `check` du langage pivot accompagnées respectivement des clauses `until` et `during`. Les figures 3.23 et 3.24 expriment le code en langage pivot généré des deux cas de test de la figure 3.18.

3.3.3 Du langage pivot vers SCXML

Cette section présente la transformation des concepts du langage pivot en code SCXML (State Chart XML) [7]. La cible SCXML décrit un test sous la forme d'un ensemble d'états imbriqués et de transitions. Le code SCXML fait appel à des fonctions XML-RPC [1] déve-

1. http ://xmlrpc.scripting.com/

FIGURE 3.23 – Génération du code en langage pivot pour les cas de test de la figure 3.18 (1/2)

```
Chapter: Airflow influence with different airflow

Tools:
<< ... >>

Setup:
 T cabin area1 = 18.0 °C
 T cabin area2 = 18.0 °C
 Q setpoint_Area1 duct1 = 0.0 kg/s
 Q setpoint_Area1 duct2 = 0.0 kg/s
 Q setpoint_Area2 duct1 = 0.0 kg/s
 Q setpoint_Area2 duct2 = 0.0 kg/s
 Model environment Kp1 = 1
 Model environment ki1 = 1
 Model environment Kd1 = 1

Test: with_ducts_open_to_2
 T setpoint area2 = 20. °C
 T setpoint area1 = 20. °C
 Q setpoint_Area1 duct1 = 0.2 kg/s
 Q setpoint_Area1 duct2 = 0.2 kg/s
 Q setpoint_Area2 duct1 = 0.2 kg/s
 Q setpoint_Area2 duct2 = 0.2 kg/s
 Check that T cabin area1 == 20. °C until 600 sec
              and
            T cabin area2 == 20. °C
 Check that T cabin area1 == 20. °C during 60 sec
              and
            T cabin area2 == 20. °C
 T setpoint area2 = 21.0 °C
 Check that T cabin area1 == 20. °C until 300 sec
              and
            T cabin area2 == 21. °C
 Check that T cabin area1 == 20. °C during 60 sec
              and
            T cabin area2 == 21. °C
```

FIGURE 3.24 – Génération du code en langage pivot pour les cas de test de la figure 3.18 (2/2)

```
Test: with_ducts_open_to_3
 T setpoint area2 = 20. °C
 T setpoint area1 = 20. °C
 Q setpoint_Area1 duct1 = 0.3 kg/s
 Q setpoint_Area1 duct2 = 0.3 kg/s
 Q setpoint_Area2 duct1 = 0.3 kg/s
 Q setpoint_Area2 duct2 = 0.3 kg/s
 Check that T cabin area1 == 20. °C until 500 sec
             and
            T cabin area2 == 20. °C
 Check that T cabin area1 == 20. °C during 60 sec
             and
            T cabin area2 == 20. °C
 T setpoint area2 = 21.0 °C
 Check that T cabin area1 == 20. °C until 180 sec
             and
            T cabin area2 == 21. °C
 Check that T cabin area1 == 20. °C during 60 sec
             and
            T cabin area2 == 21. °C
```

loppées spécifiquement pour répondre aux besoins d'automatisation des tests de systèmes avioniques. Ces fonctions permettent, par exemple, d'affecter une valeur à un paramètre d'entrée ou de vérifier l'état d'un paramètre de sortie du système. Il est aussi possible de communiquer avec un serveur pilotant l'exécution des outils externes du banc et gérant le téléchargement des différentes configurations du système.

La figure 3.25 détaille le *template* du code SCXML produit par le générateur de modèle à modèle MPS. Ce *template* est lié au nœud `TestChapter` du langage pivot : il correspond au modèle du code SCXML généré pour chaque chapitre de test formalisé en langage pivot.

SCXML demande de produire un bloc contenant l'initialisation des variables utilisées dans le code. Ces variables sont créées dans un bloc nommé *data model*. La création du *data model* pour la génération du code SCXML a été factorisée dans un *template* séparé. L'appel à ce *template* se fait par la méthode $CALL$ en passant en paramètre le nom du *template* (dans notre cas le *template* `DataModel`), comme cela est indiqué à la deuxième ligne de la figure 3.25.

L'état `Temporary_unsubscribe` décrit à la ligne 9 réalise le désabonnement de tous les paramètres avioniques qui auraient pu être manipulés par des tests antérieurs. Cet état a pour but de remettre le système dans les conditions initiales du test. Le *template* général fait ensuite appel, à la ligne 25, au *template* `subscribe` gérant la souscription de tous les paramètres avioniques utilisés par le test. L'état SCXML suivant à la ligne 28 incarne le *setup* d'un chapitre de test. Il est entouré de la clause IF car il ne doit être

FIGURE 3.25 – *Template* de génération du code SCXML pour un chapitre de test

```
<scxml initial="Temporary_unsubscribe">
  $CALL$ DataModel[<datamodel>
                     <data expr="{'Name':'Application_NAME','BUSNAME':{'Media':'VMAC,AFDX,ARINC,TYPED'}}"
                           id="Application_NAME"/>
                     <data expr="{'ParameterType':'Float'}" id="ADAPT_entsorsys_VMAC.1.PARAMETERNAME"/>
                     <data expr="{'ParameterTpe':'Float'}" id="ADAPT_entsorsys_VMAC.1._115VU_MODE_ETC"/>
                     <data id="main_final_done"/>
                   </datamodel>]
  <state id="Temporary_unsubscribe">
    <invoke type="xml-rpc" id="temporary_unsubscribe_invoke">
      <content>
        <xmlrpc tool_id="CGIB" tool_type="GFIB">
          <function name="UnSubscribe">
            <returnvalue name="Result_unsubscribe"/>
          </function>
        </xmlrpc>
      </content>
      <finalize>
        <assign expr="_event.data.Result_unsubscribe" location="Result_unsubscribe"/>
      </finalize>
    </invoke>
    <transition status="success" cond="Result_unsubscribe.Status == 'OK'" target="subscribe" event="done.invoke._all
    <transition status="fail" cond="Result_unsubscribe.Status =! 'OK'" target="FinalSCXML" event="done.invoke._all"/
  </state>
  $CALL$ subscribe[<SUBSCRIBE></SUBSCRIBE>]
  <state id="Init">
    <transition status="success" target="$[setup]"/>
  </state>
  $IF$[<state id="setup" initial="Setup_Procedure">
         <state id="Setup_Procedure">
           $LOOP$[$SWITCH$ switch[<instructionSetup></instructionSetup>]]
           <final id="FinalTest_"/>
           <transition target="final_setup"/>
         </state>
         <transition status="success" cond="$[NbtestPassed == 'null']" target="test0">
           <assign expr="'0'" location="NbtestPassed"/>
         </transition>
         $LOOP$[$IF$[<transition status="success" cond="$[NbtestPassed == 1]" target="$[test0]">
                       <assign expr="$['0']" location="NbtestPassed"/>
                     </transition>]]
         <final id="final_setup"/>
       </state>]
  $LOOP$[<state id="$[test0]" initial="$[Temporary_unsubscribe]">
           <state id="$[Test_Procedure]">
             $LOOP$[$SWITCH$ switch[<instructionTest></instructionTest>]]
             <transition target="$[Final0]"/>
             <final id="$[FinalTest]"/>
           </state>
           <final id="$[final]"/>
           <log expr="$['']" label="info"/>
           <transition target="$[FinalSCXML]"/>
         </state>]
  <final id="FinalSCXML"/>
  <state id="unsubscribe">
    <invoke type="xml-rpc" id="unsubscribe_invoke">
      <content>
        <xmlrpc tool_id="CGIB" tool_type="GFIB">
          <function name="UnSubscribe">
            <returnvalue name="Result_unsubscribe"/>
          </function>
        </xmlrpc>
      </content>
      <finalize>
        <assign expr="_event.data.Result_unsubscribe" location="Result_unsubscribe"/>
      </finalize>
    </invoke>
    <transition status="fail" cond="Result_unsubscribe.Status != 'OK'" target="FinalSCXML" event="done.invoke._all"/>
    <transition status="success" cond="Result_unsubscribe.Status == 'OK'" target="FinalSCXML" event="done.invoke._all"
        />
    <log expr="'NbtestPassed'" label="info"/>
  </state>
</scxml>
```

généré que lorsqu'un bloc *setup* est instancié dans le chapitre. Les instructions de ce bloc d'initialisation sont traitées une à une grâce au code de la ligne 31 :

```
$LOOP$ $SWITCH$ switch<instructionSetup></instructionSetup>
```

Ce code réalise l'appel au `switch` pour chacune des instructions du bloc d'initialisation. La figure 3.26 présente le code MPS du `switch` prenant en compte les différents types d'instructions du langage pivot. Chaque type d'instruction est lié à un *template* correspondant au code à générer.

FIGURE 3.26 – Génération en fonction du type d'instruction

```
cases:

  [concept    Set      ] -->Set(stateNumber, testNumber)
  [inheritors true     ]
  [condition  <always>  ]

  [concept    Check    ] -->CheckTemplate(stateNumber, testNumber)
  [inheritors true     ]
  [condition  <always>  ]

  [concept    Check                                                                  ] -->CheckUntil(stateNumber,
  [inheritors true                                                                   ]                 testNumber)
  [condition  (genContext, node, operationContext)->boolean {                        ]
  [             if (node.abstactTimeConstraint.isInstanceOf(until)) { return true; } ]
  [             return false;                                                        ]
  [           }                                                                      ]

  [concept    Call     ] -->Call(stateNumber, testNumber)
  [inheritors true     ]
  [condition  <always>  ]

  [concept    ManualAction] -->ManualAction(stateNumber, testNumber)
  [inheritors true        ]
  [condition  <always>    ]
```

Le bloc contenu dans la macro $LOOOP$ présent juste après le bloc d'initialisation à la ligne 43 de la figure 3.25 correspond à l'état généré pour chaque cas de test décrit par l'AST d'un programme. De façon analogue au bloc d'initialisation, le *switch* est ici utilisé pour générer le code des instructions dans un bloc de test en fonction de leur type.

Le *template* de l'instruction `Set` est présenté à la figure 3.27. Toutes les portions de code des *templates* entre `$[` et `]` sont remplacées par une chaîne de caractères indiquant des informations stockées dans l'AST. Le code de l'instruction `Set` pour un paramètre avionique commence par la mise en place de l'émission d'un évènement au bout de trois secondes si l'invocation de la fonction XML-RPC ne produit aucune réponse. La réception d'un évènement de ce type permettra de ne pas rester bloqué indéfiniment dans cet état. Le *template* `invokeSetParam` à la ligne 14 fait l'appel à la fonction XML-RPC `SetParam` prenant en paramètre une variable indiquant le paramètre avionique et la valeur à lui faire correspondre.

Le *template* de l'instruction `Set` propose ensuite les trois transitions amenant à l'état final de l'instruction comme montré à la figure 3.14a. La première transition à la ligne 15

correspond au succès, lorsque le retour de la fonction `SetParam` est reçu. La transition de la ligne 19 traduit un échec avec un statut du retour différent de OK. La dernière transition, ligne 24 est exécutée uniquement lorsqu'un *time out* est déclenché après trois secondes sans aucune réponse.

FIGURE 3.27 – *Template* de génération de l'instruction `Set` en SCXML

```
content node:
1  <state id="$[t0_s0]" initial="$[s0_timeout]">
2    <state id="$[s0_timeout]">
3      <onentry>
4        <send event="$[s0_TimeOut_Event]" delay="3000ms"/>
5      </onentry>
6      <transition target="$[s0_1_SET_value_of_PARAM_FUNCTION_STATUS_NO_TO_VALUE]"/>
7    </state>
8    <state id="$[s0_1_SET_value_of_PARAM_FUNCTION_STATUS_NO_TO_VALUE]">
9      <onentry>
10       <assign expr="
11               $[{'Parameter': 'PRIM1A.param','FunctionType':'Float','Status':'NO', 'FunctionValue' : {'Value':'1'}}]
12               location="$[MultiSetList_s0[0] ]"/>
13     </onentry>
14     $CALL$ invokeSetParam[<invoke></invoke> ]
15     <transition status="success" target="$[main_final]" event="$[done.invoke.i0]">
16       <log expr="$['Set value of PRIM1A.PARAM (Function Status:NO) to 1 :;:OK:;:True:;:True:;:::;:Success:;:0']"
17            label="info"/>
18     </transition>
19     <transition status="fail" target="$[main_final]" cond="$[Return_MultiSetList_s0.Status != OK]"
20                 event="$[done.invoke.i0]">
21       <log expr="$['Set value of PRIM1A.PARAM (Function Status:NO) to 1 :;:OK:;:True:;:True:;:::;:Success:;:0']"
22            label="Error"/>
23     </transition>
24     <transition status="fail" target="$[main_final]" event="$[s0_TimeOut_Event]">
25       <assign expr="'TimeOut'" location="$[s0_Status]"/>
26       <log expr="$['Set value of PRIM1A.PARAM (Function Status:NO) to 1 :;:OK:;:True:;:TimeOut:;:::;:TimeOut:;:0']"
27            label="Error"/>
28     </transition>
29     <onexit>
30       <assign expr="[]" location="$[MultiSetList_s0]"/>
31       <log expr="
32            $['Set value of PRIM1A.PARAM (Function Status:NO) to 1 :;:NA:;:PRIM1A.param:;:'+PRIM1A.param+':;:::;:None:
33            label="info"/>
34     </onexit>
35   </state>
36   <final id="$[main_final_s]">
37     $IF$[<onentry>
38            $IF$[<assign expr="$[Val]" location="NbtestPassed"/> ]
39            <assign expr="'Executed'" location="main_final_done"/>
40          </onentry>]
41   </final>
42   $IF$[<transition target="setup"/> ]
43   <transition target="$[Next]"/>
44 </state>
```

L'exécution de l'affectation étant terminée, il ne reste plus qu'à réaliser la transition vers la prochaine instruction ou le prochain bloc de test à exécuter.

3.3.4 Du langage pivot vers Python

Pour la deuxième cible de génération, nous avons utilisé la transformation de modèle à texte proposée par MPS. Cette transformation fait correspondre aux concepts du lan-

gage pivot du code en Python. Le code Python utilise la bibliothèque de fonctions `alysa` développée lors du projet ACOVAS intégrant les actions élémentaires pour manipuler un système au travers du banc. Les actions possibles consistent à affecter une valeur à un paramètre avionique en entrée du système ou à vérifier l'état d'un paramètre en sortie du système.

FIGURE 3.28 – Code du générateur Python pour le concept `TestChapter`

```
1  text gen component for concept TestChapter {
2  file name : <Node.name>
3  extension : (node)->string {
4    return "py";
5  }
6  encoding : utf-8
7  text layout : <no layout>
8  context objects : << ... >>
9
10   (context, buffer, node)->void {
11     append {from Interface_IRIT_ALYSA import *} \n \n;
12     append {alysa=Interface_IRIT_ALYSA("} ${node.name} {")} \n;
13     if (node.setup.isNotNull) {
14       append ${node.setup} \n;
15     }
16     append test node;
17     append {def main():} \n;
18     with indent {
19       foreach test in node.testCase {
20         if (node.setup.isNotNull) {
21           append {setup()} \n;
22         }
23         append {Test_} ${test.name} {()} \n;
24       }
25     }
26     append {main()};
27   }
```

La figure 3.28 présente le générateur de code Python correspondant au concept `Test-Chapter` du langage pivot. Ce concept est un nœud racine des AST des programmes en langage pivot. Un fichier de code Python est produit pour chaque chapitre de test. Ce fichier généré propose une méthode `setup()` dans le cas où un chapitre de test en langage pivot contient un bloc d'initialisation. L'appel au nœud `${node.setup}` à la ligne 14 établit le lien avec le code de la figure 3.29.

Le *template* génère aussi une méthode de test pour chaque bloc de test d'un chapitre de test décrit en langage pivot. Les blocs de test d'un chapitre sont générés grâce à l'opération `test` de la ligne 16. Le code de l'opération `test` est détaillé à la figure 3.30 ; il fait appel au *template* de génération du concept `Test` pour chaque cas de test d'un chapitre. Le *template* de génération du concept `Test` est présenté à la figure 3.31. L'orchestration de l'exécution

FIGURE 3.29 – Code généré pour le concept `Setup`

```
text gen component for concept Setup {
  (context, buffer, node)->void {
    append {def setup():} \n;
    with indent {
      append {global alysa} \n;
      append {alysa.logger.Notice("TEST INITIALISATION")} \n;
      foreach instruction in node.Statements {
        append instruction instruction \n;
      }
    }
  }
```

de la méthode `setup` entre chaque bloc de test est réalisée par la méthode `main()` de la ligne 26.

FIGURE 3.30 – Génération de tous les cas de test d'un chapitre

```
base text gen component test extends <no baseTextGen> {
  operation test(node<TestChapter> testChapter) {
    foreach test in testChapter.testCase {
      append ${test};
    }
  }
```

FIGURE 3.31 – Code généré pour le concept `Test`

```
text gen component for concept Test {
  (context, buffer, node)->void {
    append {def Test_} ${node.name} {():} \n;
    with indent {
      append {global alysa} \n;
      append {alysa.logger.Notice("====== Start of } ${node.name} { ======")} \n;
      foreach instruction in node.instructions {
        append instruction instruction \n;
      }
      append {alysa.logger.Notice("====== Test } ${node.name} { End ======")} \n;
    }
  }
```

Chaque bloc de test ou de `setup` est ensuite peuplé des instructions qui le composent. La génération de chaque type d'instruction est gérée par l'opération `instruction` qui, en fonction du type d'instruction, invoque le générateur de code correspondant.

La figure 3.32 retranscrit le code généré d'une instruction `Set`. Ce code initialise trois variables Python correspondant respectivement au nom de l'application, au nom du bus et au nom du signal paramètre avionique manipulé par l'instruction. La référence à ce

FIGURE 3.32 – Code généré pour l'instruction `Set`

```
text gen component for concept Set {
  (context, buffer, node)->void {
    append {application="} ${node.appliRef.applicationRef.name} {"} \n;
    append {bus="} ${node.busRef.busRef.name} {"} \n;
    append {signal="} ${node.signalRef.signalRef.name} {"} \n;

    append {sig=alysa.formatSignalALYSA(application,bus,signal)} \n;
    append {alysa.logger.Notice("SET } ${node.signalRef.signalRef.name} { to } ${node.getValue()} {")}
    append {alysa.signalSetReal(sig,} ${node.getValue()} {)} \n;
  }
```

paramètre est récupérée grâce à la fonction `formatSignalALYSA`. Puis, l'affectation de la valeur à ce paramètre est réalisée par la fonction `signalSetReal`.

3.4 Conclusion

Lors de cette expérimentation, nous avons conçu le langage pivot pour répondre aux besoins d'automatisation des tests des systèmes avioniques étudiés. Ce langage a pour but de réduire le fossé entre le code exécutable et la description des cas de test formalisés. Nous l'utilisons dans le *framework* BDD comme langage intermédiaire lors de la génération du code exécutable.

Nous proposons de formaliser les tests avec une syntaxe proche du langage naturel *via* des DSTL. Le DSTL ATA 21 permet de décrire les cas de test validant le comportement des systèmes de régulation de l'air d'un avion. Chacune des instructions de ce langage répond à un besoin précis correspondant aux problématiques du domaine ciblé. La syntaxe des instructions, proche du langage naturel, rend leur compréhension possible par les testeurs du domaine.

Cette première expérimentation nous a permis de démontrer la faisabilité de notre solution. Le *framework* utilise désormais le langage pivot, dont nous proposons une première version, et un premier DSTL dédié à l'ATA 21 à partir duquel nous proposons de générer du code en langage pivot. Le *framework* donne la possibilité de générer du code exécutable dans deux cibles différentes. Nous avons aussi pu valider l'utilisation du langage pivot dans la chaîne de transformations.

Le développement des langages composant le *framework* est grandement simplifié par l'utilisation de l'atelier MPS. Cet atelier simplifie aussi la manipulation de différents concepts d'un DSL car ils sont munis d'éditeurs projectionnels qui guident les utilisateurs à produire un code valide. Dans ces éditeurs, un programme ne correspond pas à une chaîne de caractères, dans laquelle le respect de la syntaxe concrète du langage est obligatoire, mais à un assemblage de concepts du langage formant l'AST du programme. Ce type d'éditeur rend la programmation accessible à des personnes n'ayant de compétences en informatique et est donc adapté à des experts en intégration système. C'est grâce à ces langages munis

d'éditeurs projectionnels que nous validons notre approche DSTL.

Les tests décrits en DSTL ATA 21 ont été exécutés sur une nouvelle génération de bancs conçue lors du projet ACOVAS. Nous avons pu tester la transformation complète, au travers de notre *framework*, d'une procédure ATA 21 pour générer du code Python. La chaîne de transformations et l'exécution du code Python ont été validées lors de la démonstration finale du projet ACOVAS. Les procédures ont un comportement similaire aux tests non-formalisés présentés à la section 3.1. L'exécution du code SCXML généré à partir du langage pivot a elle aussi été testée sur un banc Airbus intégrant un séquenceur SCXML.

Nous souhaitons maintenant proposer un DSTL dédié à un autre ATA afin d'expérimenter à nouveau notre approche DSTL et pour augmenter les possibilités de notre *framework*.

Chapitre 4

Expérimentation de la formalisation d'un DSTL pour l'ATA 42

Un deuxième cas d'étude, beaucoup plus complexe, nous a été proposé par les partenaires du projet ACOVAS. L'intégration de systèmes IMA (*Integrated Modular Avionic*) se concentre essentiellement sur la validation des téléchargements des différentes configurations des calculateurs IMA et de leur comportement une fois la configuration téléchargée. Les procédures de ce corpus sont décrites en langage naturel et sont actuellement utilisées pour les tests d'intégration. Elles sont stockées dans des fichiers *Word* d'une trentaine de pages décrites en anglais. En moyenne, une procédure est composée de 360 instructions et chaque test contient une douzaine d'instructions. Autant au chapitre précédent les travaux sur l'ATA 21 ont été utilisés comme preuve de concept de notre approche, autant les travaux sur l'ATA 42 s'inscrivent dans une perspective de passage à l'échelle.

Dans l'ingénierie avionique, les tests d'intégration sont cruciaux : ils permettent de s'assurer du bon comportement d'un avion avant son premier vol, ils sont nécessaires au processus de certification et permettent des tests de non-régression à chaque nouvelle version d'un système, d'un logiciel ou d'un matériel. La conception d'un test d'intégration coûte cher car elle mêle la réalisation de la procédure, le paramétrage de nombreux outils couplés au banc ainsi que l'adressage des interfaces du système testé. Avec des procédures de test non formalisées, l'interprétation des instructions d'un test lors de son rejeu manuel peut provoquer des erreurs coûteuses à corriger, en raison notamment des actions précises à entreprendre lors de l'exécution d'une instruction de test.

Dans notre approche BDD dédiée aux tests d'intégration système, nous proposons une formalisation de ces procédures à l'aide d'un langage proche des préoccupations des testeurs. Ces tests formalisés permettent d'augmenter la confiance dans les consignes des procédures et augmentent la valeur des procédures de test. La possibilité d'une automati-

sation même partielle de ces tests permet aux équipes de testeurs, déchargés de certaines tâches laborieuses et fastidieuses, de se concentrer sur la réalisation de nouveaux cas de test nécessaires à la mise au point de tels systèmes embarqués et, de façon plus générale, à la fiabilité des systèmes avioniques. Les experts en charge des test disposeront ainsi de plus de temps pour rejouer un *bug* ou pour en découvrir les causes afin que celui-ci puisse être corrigé au plus vite.

Ce chapitre commence par une présentation du corpus des procédures mises à notre disposition par Airbus (section 4.1). Ces procédures se basent sur une structure générique des documents de test que nous avons implémentée dans un langage *core* (section 4.2). Notre idée est de décomposer tout DSTL en deux parties distinctes, une structure générique sur laquelle vient se greffer un jeu d'instructions propre à chaque domaine étudié. Dans le cadre de l'ATA 42, l'émergence de patrons de phrases, représentant la syntaxe concrète du jeu d'instructions, n'a pas été chose aisée. Nous présentons une expérimentation combinant des techniques de traitement du langage naturel (section 4.3) et des cartes conceptuelles (section 4.4) pour découvrir le jeu d'instructions dédié à l'ATA 42. Une première validation consistant à l'écriture de procédures de test existantes grâce au DSTL ATA 42 terminera ce chapitre (section 4.5).

4.1 Présentation du corpus des procédures

Le test des calculateurs IMA reste un domaine où l'automatisation des tests n'est pas systématique, ce qui implique que toutes les instructions d'un test se doivent d'être compréhensibles par les testeurs. L'exécution d'une procédure leur demande de manipuler, en même temps, plusieurs outils et interfaces graphiques composant le banc de test. L'utilisation de ces outils et interfaces requiert une expérience spécifique pour obtenir des résultats de test exploitables. Ce manque d'automatisation oblige les testeurs à renseigner eux-mêmes les fichiers de résultats (nommés *Test Result Analysis*) avec les données produites par le système lors du test.

Le corpus de procédures présenté dans cette section a été utilisé comme point de départ de l'étude du domaine ATA 42. Ces procédures proviennent de deux équipes d'intégration testant chacune un module IMA. La présentation du corpus commence par décrire la structure de ces procédures. Cette structure, commune à plusieurs équipes intégrant des systèmes de différents domaines avioniques, a été mise au point par une des équipes d'intégration de l'ATA 46 (systèmes d'information).

La structure commune des procédures ainsi que la classification des instructions dénotent un effort de standardisation qui tend à se propager, pour unifier les procédures d'un domaine et même plus largement, de différents domaines. Même si ces procédures sont toutes exécutées manuellement, la formalisation est une des préoccupations des équipes de testeurs, car elle permet d'améliorer la communication au sein du processus d'intégration en se servant de la procédure comme un média. Néanmoins, les phrases en langage naturel

sont porteuses d'ambiguïtés pouvant causer des erreurs d'interprétation lors de l'exécution.

4.1.1 Structuration des procédures existantes

Chaque fichier de ce corpus est une procédure composée d'un ensemble de chapitres de test. Un chapitre de test comporte un ou plusieurs tests qui sont définis par une séquence d'actions.

Pour définir un chapitre, un premier bloc (*Objective*) lui associe un nom, grâce à la balise [*TITLE*], et détaille son intention grâce à la balise [*DESCRIPTION*]. Le deuxième bloc (*Covered Objectives*) établit le lien entre un ou plusieurs objectifs de test. Chaque objectif couvert par le chapitre est identifié par une balise [*OBJECTIVE*]. Le troisième bloc (*Test configuration and test means*) correspond aux moyens de test nécessaires pour exécuter le test. Ce bloc liste le ou les bancs capables de l'exécuter et les outils utilisés pour le chapitre. Un exemple d'en-tête de chapitre de test contenant les trois premiers blocs est présenté à la figure 4.1.

FIGURE 4.1 – Exemple d'en-tête d'un chapitre de test d'intégration ATA 42

Objective:
[TITLE] Empty CRDC behavior when SPP not acquired [/TITLE]
[DESCRIPTION] Check that CRDC is able to start-up and accepts dataloading when SPP messages are not sent. [/DESCRIPTION]

Covered Objectives:
[OBJECTIVE] OBJ-CRDC-SPP-001 [/OBJECTIVE]

Test configuration and test means:
This test is performed on Resources&Infra and Network-IS test benches with the configuration stated in §Test Means.

Tools used for this procedure:
- DLCS
- FSA-NG
- PIT
- Wireshark

Le dernier bloc (*Test sequences*) décrit à la figure 4.1 spécifie les actions à exécuter pour le chapitre. Ces actions sont encadrées par des balises [*TEST*]. Dans le cas où un test est décomposé en plusieurs sous-tests, la balise [*SUBTEST*] est utilisée pour les séparer. Chaque sous-test porte un nom unique et, dans certains cas, le premier des sous-tests factorise les instructions initiales communes à de multiples sous-tests. L'appel du sous-test d'initialisation dans les sous-tests concernés est décrit par une phrase impérative commençant par le verbe *Execute*, suivi du nom du sous-test.

4.1.2 Classification des instructions de test

Les procédures utilisent une première classification des instructions composant les blocs de test. La plupart des procédures textuelles sont composées d'instructions typées par les

cinq balises suivantes : *[STEP]*, *[CHECK]*, *[LOG]*, *[TRACE]* et *[REMINDER]*.

FIGURE 4.2 – Exemple de séquence d'instructions pour un chapitre de test

Test sequences:

[TEST]

-
- [STEP] Perform "ARINC 615A dataloading & configuration report: normal operation" test procedure. [/STEP] [STEP]Power off the CRDC. [/STEP]
- [STEP] Disconnect the CRDC from the network and connect EDAT directly to this last. [/STEP]
- [TRACE] Start a Wireshark trace of whole AFDX traffic.[/TRACE]
- [STEP] Power on the CRDC. [/STEP]
- [STEP]Wait for a SID, in order to check the CRDC configuration. [/STEP]
- [STEP]With EDAT, perform an Information Mode (and wait for CRDC answer). [/STEP]
- [STEP] Stop the wireshark trace. [/STEP]
- [STEP]Power off the CRDC. [/STEP]
- [STEP]Connect SIS reader. [/STEP]
- [TRACE]Take a photo of the screen of the SIS reader [/TRACE]
- [CHECK] Check with both wireshark captures (from this test, and from the "ARINC 615A dataloading & configuration report: normal operation" test), SIS screenshot and module identification plate, that information match (P/N, S/N for module and P/N for CSW, CT and RBCT). [/CHECK]
- [TRACE]Compute the wireshark trace into EDAT Spy and save relative report.[/TRACE]

[CHECK]Check with EDAT Spy that no check KO is reported in order to assess compliance with ARINC 615A-2. [/CHECK][/TEST]

Les balises *[STEP]* sont peuplées de phrases décrivant des actions à réaliser sur le SUT ou sur le banc. Les balises *[CHECK]* sont des instructions pour vérifier l'état du SUT à un instant précis. Ces vérifications sont réalisées grâce aux outils pilotés par le banc. Les balises *[LOG*] contiennent des phrases pour spécifier les informations produites par le SUT à enregistrer dans un fichier de résultats durant le test. Les balises *[TRACE]* réalisent les lancements et les arrêts des outils d'enregistrements des données produites durant le test.

Les balises *[REMINDER]* apportent des commentaires insérés dans la description des cas de test. Dans certains cas, les instructions étiquetées par ces balises sont prépondérantes à la compréhension des instructions qui lui sont juxtaposées.

Le texte contenu entre ces balises forme une ou plusieurs phrases en langage naturel comme illustré à la figure 4.2. Il correspond à la séquence d'instructions d'un chapitre contenant un seul test composé de neuf *Step*, trois *Trace* et deux *Check*. La séquence d'actions considérée est formalisée à la figure 4.25 de la section 4.5.

Bien que les procédures soient écrites en langage naturel, la formalisation proposée grâce à cette classification des instructions sert actuellement à générer des documents pré-structurés pour accueillir les résultats d'un test. Ces documents permettent d'acquitter du bon déroulement des instructions *Step* et *Trace*, d'enregistrer les données de test notifiées par les instructions *Log* et enfin de valider les comportements et les états du système décrits par les instructions *Check*. Les champs correspondant aux instructions *Check* ne sont généralement pas renseignés durant l'exécution d'un test car ces actions demandent une analyse, réalisée *a posteriori*, des résultats et des données collectées.

Les procédures exécutées manuellement sont susceptibles d'être entachées d'erreurs hu-

maines et les nombreuses manipulations qu'elles nécessitent augmentent le risque d'erreurs rendant les données enregistrées inutilisables.

4.1.3 Métriques du corpus

Le corpus fourni est composé de dix-huit procédures. Nous avons uniquement considéré les dix procédures contenant des balises pour typer les phrases en langage naturel. Ces dix procédures totalisent 108 chapitres de test et 252 tests ou sous-tests comportant au total 3708 instructions. La procédure la plus grande est stockée dans un fichier de plus de 100 pages ; elle contient 16 chapitres, 104 tests et décrit plus de 1500 instructions à elle seule. La procédure la plus réduite comporte 3 chapitres et 6 cas de test contenant seulement 64 instructions.

TABLE 4.1 – Nombre d'instructions par type

	Step	Check	Log	Trace	Reminder
Nombre	1274	1047	594	520	269
%	34,4	28,3	16	14	7,2

Le nombre d'instructions ainsi que leur volume par type en pourcentage est détaillé à la table 4.1. Les instructions les plus présentes sont les *Step* pour plus d'un tiers et les *Check* pour 28%. Ces deux types sont les plus représentés car les *Step* correspondent aux actions et les *Check* correspondent aux vérifications (assertions) du test. Viennent ensuite les *Log* et les *Traces*, respectivement 16% et 14%, servant à enregistrer les informations produites par le SUT durant le test.

Les phrases contenues à l'intérieur de ces balises sont décrites en anglais et presque aucune n'est identique. Un effort est entrepris pour unifier la tournure de certaines phrases récurrentes. Cet effort est principalement visible pour les *Check*, *Log* et *Trace* et a pour but de diminuer les erreurs de compréhension. Le contenu des *Check* et *Log* comporte beaucoup de détails pour spécifier respectivement, les comportements du SUT attendus et les données produites lors du test à enregistrer. Les *Step* décrivent les actions à réaliser pour obtenir le comportement que l'on souhaite vérifier. Ces actions sont spécifiques au SUT et aux objectifs étudiés. On remarque, dans toutes ces phrases, l'utilisation omniprésente de sigles et d'un vocabulaire spécifique aux tests d'intégration de l'ATA 42.

4.2 Structure générique d'un DSTL

Nous souhaitons reprendre cette nomenclature ainsi que la structure des procédures du corpus afin de les formaliser dans un langage dédié. Ce langage sera facilement réutilisable par d'autres ATA. Il est nommé langage *core*. Dans cette nomenclature, une procédure de

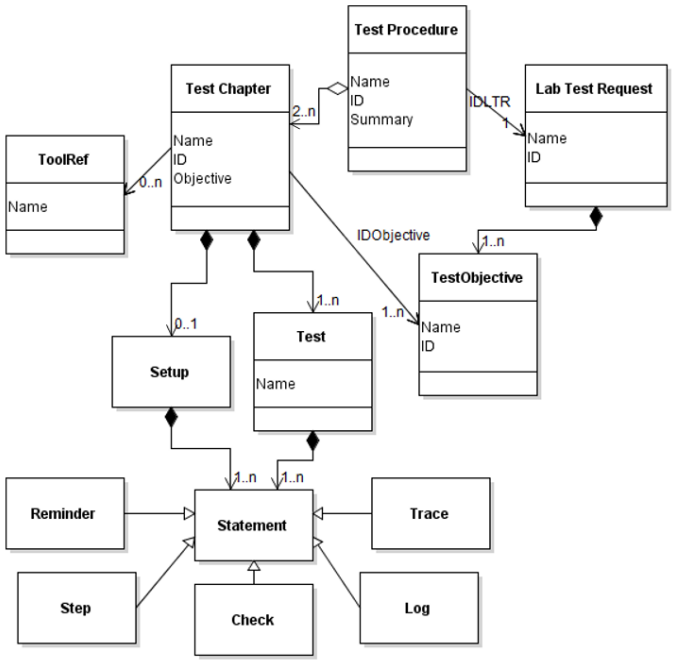
test est composée de plusieurs chapitres de test ; un chapitre de test est généralement composé d'un ensemble de tests. Chaque test valide le même comportement dans des conditions différentes. Le langage *core* formalise la structure des procédures d'intégration et aide à la conception de tests isolés les uns des autres.

Les procédures actuelles sont décrites dans des fichiers *Word* qui ne s'interfacent pas avec les outils spécifiques de traçabilité des exigences. Nous proposons de formaliser le document listant les objectifs de test grâce au langage *core*. Le lien entre les tests et les exigences donne la possibilité de générer automatiquement la matrice de traçabilité permettant de s'assurer que tous les objectifs sont bien atteints.

4.2.1 Concepts du langage *core*

Un chapitre utilise fréquemment le premier test de la liste pour factoriser les étapes d'initialisation des différents tests. Nous proposons de formaliser ce concept par un bloc d'initialisation optionnel (`setup`) au début des chapitres de test. Ce bloc d'initialisation est exécuté entre deux cas de test pour remettre le système dans les conditions initiales du test à l'instar des exécuteurs de tests logiciels de type xUnit.

FIGURE 4.3 – Méta-modèle du langage *core*



La structuration des procédures de test a été déclinée en concepts MPS selon le méta-modèle de la figure 4.3. Puisqu'une `TestProcedure` est vue comme un ensemble de `Test-`

`Chapter`, elle doit en contenir au moins deux. Elle référence un `Lab Test Request` (LTR). Elle est formalisée par un nom, un identifiant et une liste d'objectifs (`TestObjective`), eux-mêmes formalisés par un nom et un identifiant.

Un `TestChapter` est composé d'un `Setup` optionnel utilisé dans certains cas pour factoriser les instructions d'initialisation communes à plusieurs tests et d'une liste de `Test`. Le concept `Setup` est optionnel car certains tests sont engagés dès lors que le banc alimente électriquement le SUT. Un `TestChapter` est identifié par un nom et un identifiant unique, son intention est décrite par le concept `Objective` et il référence aussi un ou plusieurs objectifs de test provenant du `LTR` référencé dans la procédure. Les concepts `Setup` et `Test` contiennent les instructions de test. Ce méta-modèle propose aussi une première classification décomposant les instructions en cinq types : `Reminder`, `Step`, `Check`, `Log` et `Trace`.

Un chapitre de test référence la liste des outils qu'il utilise (`ToolRef`). L'utilisateur choisit les outils à référencer parmi les outils disponibles sur le banc, qui auront été formalisés préalablement grâce au langage `Tools` présenté au chapitre précédent. Seuls les outils référencés dans un chapitre de test peuvent être utilisés dans les instructions qui composent les concepts `Setup` et `Test`.

FIGURE 4.4 – Procédure contenant trois chapitres de test

```
Test procedure: CRDC Test procedures of SPP integration
ID: V42D13007870
Summary:
Test chapters will be performed on the two CRDC positions which allow SPP: A04 and A06.

Responding to: IMA LTR SPP emissions

Test chapters:
 EMPTY CRDC BEHAVIOR WHEN SPP NOT ACQUIRED
 INTEGRATED CRDC BEHAVIOR WHEN SPP NOT ACQUIRED
 CRDC BITE integration
```

La figure 4.4 présente une procédure décrite dans le langage *core* sous MPS. Cette procédure référence trois chapitres de test. Son nom est *CRDC Test procedures of SPP integration*. Derrière le mot-clé `Summary`, suit un commentaire en format libre explicitant le but de la procédure. On retrouve ensuite la référence au `LTR` *IMA LTR SPP emissions* auquel la procédure répond après les mots-clés `Responding to`. Le `Lab Test Request` *LTR IMA LTR SPP emissions* est présenté à la section 4.2.3. Les chapitres de test référencés par la procédure sont listés à la fin de la procédure par les mots-clés `Test chapters`. Il est possible de référencer autant de chapitres qu'on le souhaite mais au moins deux sont nécessaires pour qu'une procédure soit valide.

La figure 4.5 montre la formalisation de l'en-tête du premier chapitre de test de la procédure de la figure 4.4. Ce chapitre de test se nomme *EMPTY CRDC BEHAVIOR WHEN SPP NOT ACQUIRED* ; il est associé à un identifiant et met en avant son intention à l'aide d'un court commentaire. Il répond à deux objectifs provenant du `LTR` *LTR IMA*

FIGURE 4.5 – En-tête d'un chapitre de test

```
Test chapter: EMPTY CRDC BEHAVIOR WHEN SPP NOT ACQUIRED
ID: 3500018
Objective: Check that CRDC is able to start-up and
           accepts dataloading when SPP messages are not sent.

Test will be perform on: Resources&Infra

Covered Objectives:
SPP-OS3
SPP-OS4

Tool:
PIT
FSA-NG
Wireshark
DLCS
```

LTR SPP emissions qui sont référencés après les mots-clés `Covered Objectives`. L'en-tête mentionne également la liste des outils utiles à l'exécution de la procédure. Cette liste est formalisée après le mot-clé `Tool`.

4.2.2 Contraintes sémantiques

Toutes les règles métier ne pouvant pas être exprimées par le méta-modèle prennent la forme de règles sémantiques. Certaines de ces règles, nommées *Concept Contraints* dans MPS, ont pour but de guider l'utilisateur à produire un code sémantiquement correct. Ces contraintes sont implémentées grâce une méthode MPS où les nœuds et les éléments de l'AST d'un programme MPS sont gérés comme des ensembles. Cette méthode retourne la liste des chapitres qu'il est encore possible de référencer pour une procédure. La figure 4.6 présente le code produit pour s'assurer qu'un chapitre de test ne peut pas être référencé deux fois dans une même procédure. Deux autres contraintes sont encore nécessaires au bon assemblage des concepts référencés par une procédure. La première s'assure de l'unicité d'un outil lors de son référencement dans un chapitre de test. La deuxième s'assure qu'un objectif de test référencé par un chapitre appartient bien au LTR référencé par la procédure.

Les *Concept Contraints* MPS permettent aux utilisateurs finaux de ne pas faire d'erreurs en proposant des listes déroulantes de choix valides. Ces listes déroulantes empêchent l'utilisateur final de commettre un impair détecté plus tard dans le cycle de vie d'une procédure.

L'utilisation de *Concept Contraints* n'exclut des erreurs sémantiques possibles et, de même que pour les environnements de développement de langages par une approche dirigée par la syntaxe concrète, des règles sémantiques évaluées *a posteriori* sont nécessaires. Ces règles sont nommées *Checking Rules*. Si ces règles ne sont pas respectées, le code concerné sera souligné en rouge dans l'éditeur et un message d'erreur sera affiché à l'utilisateur. Nous

FIGURE 4.6 – Contrainte sur les références d'un chapitre d'une procédure

```
link {TestChapterRef}
  referent set handler <none>
  scope (referenceNode, contextNode, containmentLink, position, linkTarget, operationContext)->Scope {
    final nlist<TestChapter> allChapter = contextNode.model.nodes(TestChapter);
    final nlist<TestProcedure> allProc = contextNode.model.nodes(TestProcedure);
    final node<TestChapterRef> me = referenceNode.TestChapterRef.isNotNull ? contextNode as TestChapterRef : null;
    final sequence<node<TestChapter>> alreadyRefButMe =
        allProc.ChapterRef.where({~it => it :ne: me; }).select({~ittest => ittest.TestChapterRef; });
    sequence<node<TestChapter>> autoCompletList =
        allChapter.where({~procRef => alreadyRefButMe.all({~idOk => idOk :ne: procRef; }); });
    return new ListScope(autoCompletList) {
      public string getName(node<> child) {
        child : TestChapter.name;
      }
    };
  }
```

avons identifié et implémenté 13 règles de type *Checking Rules* pour le langage *core* :

- Les noms d'une `TestProcedure`, d'un `TestChapter`, d'un `Lab Test Request` et de ses objectifs doivent être uniques.
- L'identifiant d'une `TestProcedure`, d'un `TestChapter` et d'un `Lab Test Request` et de ses objectifs doivent être uniques.
- Une `TestProcedure` est composée d'au moins deux `TestChapter`.
- Un `Lab Test Request` contient au moins deux objectifs de test.
- Un `Test` et un `Setup` contiennent au moins une instruction `Step`.
- Un `Test` contient au moins une instruction `Check` ou une instruction `Trace`.

FIGURE 4.7 – Contrainte d'unicité pour les noms de `LTR`

```
checking rule LTRNameUniqueness {
  applicable for concept = LabTestRequest as labTestRequest
  overrides false

  do {
    sequence<string> nameObjectiveLTR = labTestRequest.model.roots(LabTestRequest).select({~it => it.name; });
    list<string> allNames = new arraylist<string>;
    allNames.addAll(nameObjectiveLTR);
    if (allNames.where({~it => it != null && it.equals(labTestRequest.name); }).size > 1) {
      error "The name of a LTR has to be unique." -> labTestRequest;
    }
  }
```

La figure 4.7 présente la *Checking Rule* définissant la contrainte d'unicité des noms des `LTR`. Cette règle informe l'utilisateur lorsque deux `LTR` ont le même nom. Ces contraintes permettent de tracer les objectifs de test en évitant les ambiguïtés lors de la création des matrices de traçabilité.

Les noms des fichiers (procédures, chapitres et LTR) doivent être uniques au sein d'une solution MPS formalisant un ensemble de procédures d'une même campagne de tests. Il est admis que ces fichiers puissent avoir le même nom dans deux campagnes différentes. Ces campagnes différentes sont formalisées dans des solutions MPS séparées et, dans ce cas, ce

sont les identifiants de ces fichiers qui serviront de clés uniques pour les distinguer lors de plusieurs campagnes de tests.

4.2.3 Traçabilité des objectifs de test

Les matrices de traçabilité découlant des procédures non formalisées nécessitent d'être produites manuellement lors de leur conception. La rédaction manuelle des matrices de traçabilité demande un effort pour les valider et pour les mettre à jour, lorsque les procédures sont amenées à évoluer. La figure 4.8 est un exemple de matrice de traçabilité produite manuellement. Cette matrice est élaborée en faisant correspondre à chaque chapitre de la procédure SPP, l'objectif qu'il couvre.

FIGURE 4.8 – Exemple de matrice de traçabilité produite manuellement

SPP Procedure	LTR Test Objective ID
PROC – EMPTY CRDC BEHAVIOR WHEN SPP NOT ACQUIRED	OBJ-CRDC-SPP-01
PROC – INTEGRATED CRDC BEHAVIOR WHEN SPP NOT ACQUIRED	OBJ-CRDC-SPP-02
PROC – CRDC BEHAVIOR DURING INTERRUPTION OF SPP BROADCAST	OBJ-CRDC-SPP-03
PROC – ACCESS OF SPP VALUES VIA THE BITE INTERACTIVE MODE	OBJ-CRDC-SPP-04
PROC – FUNCTIONAL TEST ON FILTERING FAILURES FROM NOT INSTALLED EQUIPMENT	OBJ-CRDC-SPP-05
PROC – FUNCTIONAL TEST ON FILTERING FAILURES FROM INSTALLED EQUIPMENT	OBJ-CRDC-SPP-06
PROC – EMPTY SPP MESSAGE ACQUISITION	OBJ-CRDC-SPP-07
PROC – SPP MESSAGE IMPACT AFTER FLIGHT PHASE 1	OBJ-CRDC-SPP-09

FIGURE 4.9 – Formalisation du `LTR` *IMA LTR SPP emissions*

```
IMA LTR SPP emissions
ID: A460SPP
System under test:
CRDC A06/A04

Objectives:
ID: SPP-0S1 Name: Interactive mode exit
ID: SPP-0S2 Name: Maintenance messages for latchable failures
ID: SPP-0S3 Name: Reaction to HOLD command
ID: SPP-0S4 Name: Consistency between Maintenance Messages and Effects
```

Un `Lab Test Request` est décrit par un nom, un identifiant et répertorie les systèmes auxquels les objectifs s'appliquent. La figure 4.9 présente le `LTR` *IMA LTR SPP emissions* référencé par la procédure *CRDC Test procedures of SPP integration* pour laquelle nous illustrons la génération automatique de la matrice de traçabilité. Ce `LTR` a pour identifiant *A460SPP*. Il est dédié aux systèmes CRDC A06 et A04. Ses objectifs sont définis par un identifiant et un nom. Une procédure peut couvrir, en partie ou totalement, les objectifs du `LTR` auquel elle est rattachée. Il n'est pas possible de lier des objectifs appartenant à un `LTR` qui n'est pas référencé par la procédure associée au chapitre concerné.

Grâce à la formalisation du `LTR`, la matrice de traçabilité des objectifs de test pour une procédure donnée est automatiquement générée. C'est un cas d'application du langage *core*. La matrice générée met en relation chaque chapitre contenu dans une procédure et les objectifs qu'il couvre. La matrice inverse permettrait de s'assurer que tous les objectifs ont bien été couverts par au moins un chapitre de test d'une procédure. Cette matrice aide à l'identification des cas de test devant être rejoués lorsqu'un certain nombre d'objectifs de test demande une nouvelle validation.

FIGURE 4.10 – Code HTML de la traçabilité des objectifs

```
<Table>
  <TR>
    <TD>EMPTY CRDC BEHAVIOR WHEN SPP NOT ACQUIRED</TD>
    <TD>SPP-0S3</TD>
    <TD>SPP-0S4</TD>
  </TR>
  <TR>
    <TD>INTEGRATED CRDC BEHAVIOR WHEN SPP NOT ACQUIRED</TD>
    <TD>SPP-0S1</TD>
    <TD>SPP-0S4</TD>
  </TR>
  <TR>
    <TD>CRDC BITE integration</TD>
    <TD>SPP-0S1</TD>
    <TD>SPP-0S2</TD>
  </TR>
</Table>
```

La matrice fournie par le langage *core* est structurée soit en XML, soit en HTML. Un exemple de code HTML correspondant à la matrice de traçabilité de la procédure de la figure 4.4 est présenté à la figure 4.10. La table HTML est composée d'une ligne par chapitre de test et sur cette ligne, la première colonne correspond au nom de ce chapitre et les colonnes suivantes listent les objectifs qu'il couvre.

4.2.4 Composition des langages dans le *framework*

Il existe quatre techniques pour composer des langages [86]. Ces techniques sont : réutiliser, embarquer, référencer et étendre. Les deux premières techniques, réutiliser et embarquer [46], n'introduisent pas de dépendance entre les langages composés mais un troisième langage, nommé langage adaptateur (*adapter language*), est nécessaire à ce type de com-

position et dépend des deux langages composés. La réutilisation conserve des fragments de code homogènes pour les deux langages composés et s'appuie sur un concept du langage adaptateur qui étend un des concepts d'un premier langage, le langage source, et qui référence un concept d'un deuxième langage, le langage réutilisé. C'est le cas de Java Native Interface (JNI) qui permet à du code Java d'appeler ou d'être appelé par des applications natives ou par des bibliothèques d'autres langages (C, C++ assembleur ...). Embarquer un langage introduit une dépendance entre les fragments de code des deux langages : les programmes décrits en langage adaptateur utilisent des concepts des deux langages composés. Pour mettre en place ce type de composition, un concept du langage adaptateur doit étendre un des concepts du langage hôte et doit aussi être composé d'un concept du langage embarqué. Par conséquent, les programmes décrits en langage adaptateur forment un ensemble de fragments de code hétérogènes. Du code SQL peut être embarqué dans de nombreux programmes écrits, par exemple en C, C++ ou COBOL. Pour les bases de données de type PostgreSQL, le code SQL embarqué dans du code C est pré-compilé, par un pré-compilateur nommé ECPG transformant le code SQL en code C.

Dans notre *framework*, nous avons utilisé le référencement et l'extension de langages car nous ne souhaitions pas introduire de nouveau langage jouant le rôle de langage adaptateur. Le DSTL ATA 21, le langage pivot et le langage *core* référencent les langages `Tools` et `ICD`. Nous avons conçu ces deux langages indépendamment car ils répondent à des problématiques communes à une majorité de domaines avioniques.

Une dépendance se crée lorsque les concepts du langage référencé se composent aux concepts du langage à concevoir. On dit alors que le langage fait référence aux concepts du langage de base. C'est le cas pour les langages `Tools` et `ICD` lorsque les DSTL et le langage pivot font référence aux outils du banc et aux paramètres avioniques de l'ICD.

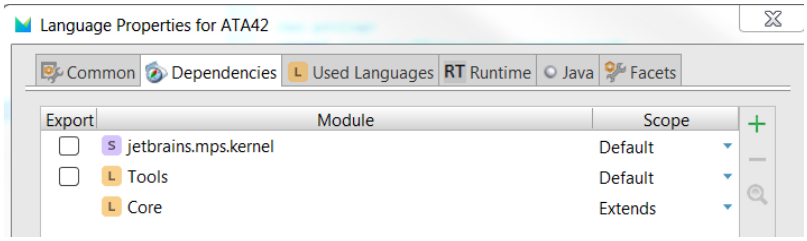
Le langage `Tools` formalise la gestion des outils externes du banc et leurs fonctionnalités. Un chapitre de test référence des outils définis par le langage `Tools` et seuls les outils référencés et leurs fonctionnalités sont utilisables dans les instructions de tests du chapitre. Le langage `ICD` formalise les paramètres d'interface d'un système. Ces paramètres sont référencés dans les instructions du langage pivot ou encore dans les fichiers de déclaration des paramètres de haut niveau du DSTL ATA 21.

La deuxième technique utilisée repose sur l'extension de langages. Le langage *core* est étendu par les langages implémentant les jeux d'instructions de chaque domaine avionique. Une instruction sera utilisable dans les blocs d'un chapitre uniquement si elle étend le concept `Statement` ou les concepts `Step, Check, Trace, Log` ou `Reminder` du langage *core*. Dans le cas où un concept correspondant à une instruction n'étend pas un des concepts cités précédemment, il ne sera pas possible de l'instancier dans un concept `Setup` ou `Test` d'un chapitre de test. Un chapitre de test formalisé grâce au langage *core* devra être composé d'instructions provenant d'un langage spécifique à un domaine avionique. La conception d'un jeu d'instructions répondant aux besoins de l'ATA 42 est présentée à la section suivante.

La conception d'un nouveau DSTL basé sur la structure proposée par le langage *core*

est grandement simplifiée car seules les instructions du DSTL sont à concevoir. Il est alors possible de concevoir autant de langages dédiés qu'il y a de domaines à couvrir.

FIGURE 4.11 – Interface MPS de gestion des dépendances du DSTL ATA 42



La structuration du langage *core* peut être utilisée pour définir simplement le DSTL ATA 21 grâce au mécanisme d'héritage de MPS. En effet, les instructions d'affectation du DSTL ATA 21 (commençant par `Set`, `Increase` ou `Decrease`) seraient typées par `Step` et les instructions d'assertion (commençant par `Check` ou `Verify`) seraient typées par `Check`.

La figure 4.11 traduit l'interface de gestion des dépendances du DSTL ATA 42. On peut y voir le lien avec le langage `Tools` qui permet au DSTL de référencer les concepts d'outils et de fonctions au sein des instructions. Cette figure explicite aussi le lien de dépendance entre le langage *core* et le jeu d'instructions dédié à l'ATA 42. Ce lien est caractérisé par *Extends* ce qui permet aux instructions de l'ATA 42 d'étendre les concepts `Step, Check, Trace, Log` ou `Reminder`. Les concepts implémentant ces instructions étendront le type qui leur aura été attribué *via* les mécanismes d'héritage de MPS. A l'utilisation, le langage *core* et le langage implémentant un jeu d'instructions doivent être importés dans la même solution MPS. Une solution est le nom d'un projet MPS dans lequel les programmes DSL peuvent être conçus.

4.3 Outils NLP pour l'identification des instructions ATA 42

L'approche BDD ne propose pas de méthodologie pour découvrir et concevoir les langages de haut niveau décrivant les cas de test. Les experts et les testeurs se réunissent pour se mettre d'accord sur les scénarios de test qu'ils ont besoin de formaliser. Les scénarios sont décrits sous la forme de patrons en langage naturel et les données variables sont extraites des phrases grâce à des expressions régulières.

Dans notre cas, les procédures sont déjà rédigées en langage naturel en dehors des considérations précédentes. Au vu de la complexité des procédures existantes, une approche exclusivement basée sur des techniques de traitement du langage naturel (*Natural Language Processing*) a été tentée sans succès, si ce n'est pour amorcer un processus empirique de découverte des patrons d'instructions. L'utilisabilité de ces outils est discutée dans la section

suivante qui sera poursuivie par une première étude statistique portant sur les quatre types d'instructions (`Step, Check, Trace, Log`).

4.3.1 Utilisabilité des outils NLP

L'utilisation d'outils NLP (*Natural Language Processing*) pour la conception de langages de test en BDD a été proposée par Soeken et al. [74]. Ces travaux visent à découvrir la structure des procédures de test à partir d'un ensemble de scénarios d'acceptation décrits en langage naturel, en utilisant des outils de traitement du langage naturel, notamment des algorithmes d'apprentissage semi-automatique de patrons de phrases. Les patrons de phrases appris par l'algorithme doivent être validés par le testeur.

Dans un premier temps, nous avons (naïvement) essayé de mettre en place des outils d'apprentissage semi-automatique de patrons de phrases du corpus. Puisque les procédures de test étaient déjà existantes et d'une complexité beaucoup plus grande que des scénarios d'acceptation, nous avons utilisé une autre suite d'outils Standford-Core-NLP [58] pour apprendre et retrouver les patrons de phrases d'un corpus, à la manière des travaux autour de l'outil WHISK [73]. Nous pensions utiliser ces algorithmes pour faire apparaître, directement à partir des procédures, la syntaxe concrète des patrons des instructions du langage. La génération de ces règles nous aurait permis de dissocier la partie fixe d'une instruction, c'est-à-dire les mots-clés formant sa syntaxe concrète, de sa partie variable représentant les concepts qu'elle manipule.

Ces algorithmes d'apprentissage sont basés sur une analyse grammaticale des phrases correctes. Malheureusement, les phrases composant ces procédures sont très hétérogènes en raison de la liberté offerte par le langage naturel. Les phrases sont écrites en anglais par des testeurs français, ce qui suscite des tournures maladroites et donc difficilement décomposables. Qui plus est, les analyseurs grammaticaux ont des difficultés à analyser des phrases impératives qui sont le lot commun des procédures de test. Pour finir, la complexité et la diversité des patrons de phrases utilisés ne permettent pas l'unification de patrons redondants.

Nous avons donc pragmatiquement, dans un deuxième temps, étudié les phrases type par type, pour identifier les instructions les plus couramment utilisées. Pour ce faire, nous avons répertorié, pour chaque fichier du corpus, les instructions en fonction de leur type et les avons stockées dans des fichiers séparés. Nous avons ensuite généré les arbres grammaticaux de chacune des phrases de ces fichiers pour les importer dans l'outil *Tregex*[1] de la suite d'outils Standford-Core-NLP [58]. Nous avons identifié manuellement des patrons ou des structures de phrases récurrentes et avons produit des statistiques sur les verbes de chaque type.

Ces travaux ont servi de point de départ pour mettre en place un processus itératif pour la conception du langage de test spécifique aux besoins d'intégration des systèmes IMA. La

1. https ://nlp.stanford.edu/software/tregex.shtml

conception du jeu d'instructions s'est poursuivie par l'élaboration d'une carte conceptuelle lors de réunions de travail avec les ingénieurs des tests d'intégration de l'ATA 42.

4.3.2 Instructions *Step*

Les instructions *Step* sont majoritaires dans le corpus des dix procédures étudiées. Elles décrivent, en langage naturel, les actions manuelles des étapes d'un test. Dans les procédures ATA 42, ces actions portent sur le calculateur IMA (le SUT) et sur les outils pilotés par le banc. Ces outils sont par exemple utilisés pour générer des paquets réseaux envoyés au SUT par l'interface du banc pour simuler des pannes. Ils pourront servir à télécharger les applications logicielles ainsi que la configuration du calculateur. Ces calculateurs possèdent différents modes de fonctionnement et leur comportement varie en fonction de l'état global de l'avion (taxi, décollage, en vol, etc.).

Listing 4.1 – Patrons des phrases *Step*

```
– Ensure to be in [Phase].
– Ensure [component] sends only good health message.
– Start up configured [module] on a simulator.
– Generate a failure with [Tools] as specifies in [File].
– Before [event] disconnect ethernet links between [Component1] and [component2].
– Restart [message] emission.
```

Le Listing 4.1 présente quelques exemples de patrons de phrases des instructions *Step* où les éléments entre crochets sont variables. Ces phrases demandent par exemple aux testeurs de valider la phase de vol du SUT. Les différentes phases de vol sont désignées par un chiffre (allant de 0 à 7 dans le cas général). Les phrases étudiées peuvent aussi décrire le démarrage du SUT (*Start up*), la génération d'une panne grâce à un outil (*Generate failure*) et bien d'autres cas.

L'utilisation de ces patrons a pour but de proposer une syntaxe proche des phrases en langage naturel contenues dans les procédures actuelles. C'est à partir de ces patrons que la syntaxe des instructions jugées nécessaires a été élaborée.

La table 4.2 montre que les verbes décrivant les instructions *Step* sont nombreux. Elle met en avant que plus de 50% des actions sont caractérisées par les verbes *Ensure*, *Start*, *Generate*, *Perform* et *Wait*. Ces verbes couvrent donc une grande partie des besoins des phrases typées `Step`. On remarque aussi que certains verbes peu usités sont synonymes comme *Create*, *Generate* et *Make* ou *Remove* et *Eliminate*. Les verbes non répertoriés couvrent environ 6% des usages.

Les phrases avec le verbe *Ensure* pourraient être vues comme ayant une intention proche des phrases *Check*. Cependant, elles ne servent pas à la validation du comportement du SUT durant un cas de test. Ces phrases s'assurent que certaines conditions sont bien respectées ou qu'une action est réalisable.

Dans les phrases étudiées, les verbes les plus courants ne sont généralement pas utilisés dans un seul contexte. Ces verbes sont polysémiques, par conséquent un même verbe inter-

TABLE 4.2 – Étude des verbes d'action des phrases typées `Step`

Verbe	Nombre d'occurrences	Nombre d'occurrences par instruction en %
Ensure	151	11,85
Start	139	10,91
Generate	136	10,67
Perform	131	10,28
Wait	113	8,86
Set	76	5,96
Execute	56	4,39
Launch	48	3,76
Through	35	2,74
Capture	29	2,27
Make	26	2,04
Create	24	1,88
Disconnect	23	1,80
Send	22	1,72
Connect	22	1,72
Select	21	1,64
Open	19	1,49
Reconnect	19	1,49
Shall	17	1,33
Remove	16	1,25
Switch	14	1,09
Use	14	1,09
Move	13	1,02
Eliminate	13	1,02
Verify	10	0,78
Access	6	0,47
Identify	4	0,31
Process	4	0,31
Total	**1201**	**94,27**

vient pour décrire des actions différentes. Par exemple, le verbe *Perform* peut être utilisé aussi bien pour des besoins de téléchargement d'une configuration (*Perform dataloading*) que pour récupérer des données provenant du SUT (*Perform an Information Mode*). Le listing 4.2 montre des exemples de patrons de phrases contenant le verbe *Perform*.

Listing 4.2 – Patrons de phrases *Step* contenant le verbe *Perform*

```
– Perform a [Phase] to [Phase] transition.
– Perform a long power cut.
– Perform an Information Mode.
```

Il en est de même pour le verbe *Generate* utilisé pour répondre à des besoins similaires au verbe *Perform*. Ce verbe est aussi employé pour la génération de pannes par exemple. Le listing 4.3 montre des exemples de patrons de phrases commençant par le verbe *Generate*.

Listing 4.3 – Patrons de phrases *Step* contenant le verbe *Generate*

```
– Generate a failure with [Tools] as specifies in [File].
– Generate a [Phase] to [Phase] transition.
– Generate a power cut.
```

De par la nature des éditeurs projectionnels, nous pouvons nous permettre de ne pas contraindre la grammaire par des considérations d'ambiguïté d'un terme. Ces éditeurs permettent à un même terme d'avoir plusieurs significations en fonction du concept manipulé par le testeur. En effet, les ambiguïtés syntaxiques n'ont aucune incidence sur l'utilisabilité d'un langage muni d'éditeurs projectionnels car l'utilisateur du langage manipule directement les concepts du langage sans considérer leur syntaxe concrète. Nous souhaitons donc proposer, pour les DSTL de notre *framework*, des instructions utilisant des verbes identiques pour formaliser des besoins différents. Cependant, nous souhaitons qu'un besoin soit formalisé par une seule instruction afin que le langage ATA 42 ne comporte pas trop d'instructions et soit facilement assimilable par un nouveau testeur. Cette formalisation a permis aux testeurs de se mettre d'accord sur un ensemble d'instructions minimal et bien formé.

4.3.3 Instructions *Trace*

Les phrases de type *Trace* pilotent les outils externes gérant l'enregistrement des données produites au cours du test. La table 4.3 présente des statistiques sur les verbes utilisés pour décrire ces phrases. Un tiers des actions concerne le démarrage des outils de trace (*Start*) et un autre tiers se rapporte à leur arrêt (*Stop*). Ensuite, on trouve les verbes pour la réalisation des captures d'écran (*Make/Take*). On remarque que *Perform, Compute, Collect* et *Save* représentent chacun moins de 10% des verbes utilisés dans les phrases étudiées. Ces verbes décrivent généralement des traitements à réaliser sur les données une fois enregistrées. Les phrases contenant le verbe d'action *Log* ne sont *a priori* pas correctement typées et devraient l'être par `Log`.

TABLE 4.3 – Étude des verbes d'action des phrases typées `Trace`

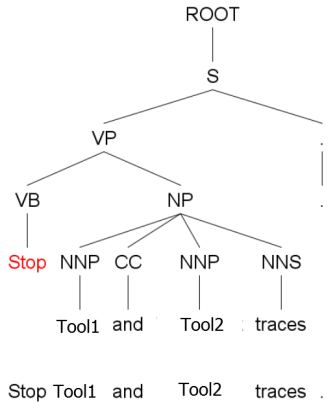
Verbe	Nombre d'occurrences	Nombre d'occurrences par instruction en %
Start	193	37,11
Stop	171	32,88
Make/Take (screenshot)	64	12,30
Perform	32	6,15
Compute	22	4,23
Collect	12	2,30
Save	12	2,30
Log	11	2,11
Total	**517**	**99,42**

On peut constater que le verbe *Perform* est présent dans les instructions *Step* dans 10% des cas (table 4.2), mais aussi dans les *Trace* à hauteur de 6% (table 4.3). Dans les instructions *Trace*, ce verbe sert à décrire des actions réalisées sur les données de test. Dans le cas des *Step*, il décrit des actions à réaliser sur le SUT. Le listing 4.4 présente une phrase `Trace` utilisant le verbe *Perform*. Des exemples de patrons de phrases *Step* basés sur le verbe *Perform* sont présentés au listing 4.2. Comme nous l'avons vu précédemment, utiliser un même verbe pour formaliser des actions différentes ne pose pas de problème grâce à l'utilisation des éditeurs projectionnels.

Listing 4.4 – Exemple d'instruction `Trace` utilisant le verbe *Perform*

```
– Perform a Non–volatile memory dump on each position.
```

FIGURE 4.12 – Arbre grammatical d'une phrase `Trace`



Les outils de traitement du langage naturel nous ont servi à identifier des arbres syntaxiques pour en déduire, par une approche empirique, les meilleurs patrons de phrases à utiliser. Ces arbres sont intégrables à l'outil Tregex [2] permettant de classer des phrases en fonction de certaines caractéristiques, pouvant être lexicales ou grammaticales. Nous avons exploité cet outil pour différencier ce qui décrit l'action de la phrase (la proposition verbale) et ce sur quoi l'action porte (le groupe nominal). Les groupes nominaux représentent les parties variables de l'instruction qui devront être renseignées par l'utilisateur du langage.

La figure 4.12 décrit l'arbre grammatical de la phrase *Stop Tool1 and Tool2 traces*. Cette phrase comporte un groupe verbal (*Verb Phrase* - VP), lui-même composé d'un verbe (VB) et d'un groupe nominal (*Noum Phrase* - NP). Ce groupe nominal est composé de deux noms propres (*Proper noun, singular* - NNP) séparés par une conjonction de coordination (*Coordinating conjunction* - CC) et il se termine par un nom commun au pluriel (*Noun, plural* - NNS).

Les outils de la suite Stranford-Core-NLP ne sont pas opérationnels dans 100% des cas ; certaines phrases impératives ne sont pas identifiées. Les erreurs les plus fréquentes surviennent lors de l'utilisation du verbe *Start* en début d'une phrase impérative où ce verbe est presque toujours identifié comme un nom.

4.3.4 Instructions *Check* et *Log*

Les phrases de type *Log* utilisent *Log* comme verbe d'action dans plus de 95% des cas. Ces phrases explicitent quelles sont les informations produites lors du test qu'il est nécessaire d'enregistrer dans le fichier de résultats. Ces informations sont extrêmement dépendantes des objectifs du test auxquels l'instruction *Log* se rattache. Elles sont fondamentales car elles sont utilisées pour la certification des systèmes d'un avion. Si toutes les informations de certification ne sont pas enregistrées correctement, le test doit être rejoué.

L'aspect manuel de ces tâches, mêlant l'utilisation des outils du banc pour récolter les informations spécifiées et l'inscription de ces informations dans le fichier de résultats est source d'erreurs. Tout ceci demande une dextérité et une organisation aux testeurs car ces actions sont à réaliser au cours du test et certaines en un temps concis.

Le listing 4.5 liste des exemples de phrases *Log* décrivant des actions manuelles pour stocker les informations servant à vérifier les comportements souhaités. Ces phrases commencent toutes par *Log in TRA*. Les phrases *Log* détaillent des actions où des informations très diverses sont à enregistrer, ces informations provenant d'interfaces d'outils pas forcément adaptées aux besoins.

Listing 4.5 – Exemples d'instructions `Log`

```
- Log in TRA the dataloading duration.
- Log in TRA the operation acceptance status code
(in the '.LUS') sent by the target, and the message displayed by the DLCS.
```

2. `https://nlp.stanford.edu/software/tregex.shtml`

```
– Log in TRA the PN of the loads seen in the SID and in the Information Mode.
– Log in TRA the message displayed by the Dataloader.
```

La même tendance se retrouve pour les phrases de type *Check*, où le verbe d'action *Check* est utilisé dans plus de 95% des cas. Ces phrases désignent le comportement attendu par le système à valider lors du test. Dans ce type de phrase, beaucoup de cas différents doivent être gérés car les instructions décrites sont spécifiques aux objectifs d'un test. Dans les 3708 instructions étudiées, un peu plus de 1000 sont des *Check* et presque aucune n'est identique.

Listing 4.6 – Exemples d'instructions `Check`

```
– Check with PIT that all logbooks of all partitions have been well created.
– Check that no aborted records are detected.
– Check with dump that NVM dumps can be successfully decoded
by Dump Bite Tool (only system partitions part). If not perform
again previous step.
– Check that Module starts correctly i.e. Module in OPS mode
and all partitions in NORMAL_START.
```

Le listing 4.6 répertorie quatre occurrences de phrases *Check* du corpus de procédures. Dans ces phrases, beaucoup de précisions sont données pour permettre aux testeurs de réaliser les vérifications souhaitées. Ces phrases donnent aussi des informations supplémentaires décrites entre parenthèses ou après l'abréviation i.e (*id est*). Cependant, toutes ces phrases suivent à peu près le même patron. Elles commencent toutes par le verbe *Check*. Parfois, ce verbe est complété par la locution *with* suivie du nom de l'outil de vérification. Le mot *that* indique les conditions que le test doit satisfaire.

4.4 Carte conceptuelle des instructions ATA 42

La conception des instructions du DSTL ATA 42 a été réalisée en collaboration avec des testeurs du domaine. Les statistiques produites précédemment ont été utilisées pour guider efficacement les discussions avec les testeurs. Les besoins de formalisation élicités lors de ces discussions sont exprimés *via* la création d'une carte conceptuelle. Son objectif est de classer les instructions en fonction des différents types proposés par les procédures du corpus. Elle établit le lien entre les besoins identifiés avec les experts testeurs d'un domaine et les mots de la syntaxe qui peupleront les instructions d'un langage dédié à ce domaine.

FIGURE 4.13 – Principaux concepts de la carte conceptuelle

La figure 4.13 résume la hiérarchie des principaux concepts de la carte conceptuelle dont la racine correspond aux instructions (*Statements*). Ces instructions sont divisées en deux : celles dédiées au SUT et celles spécifiques au test. Le dernier niveau présenté par cette figure est le type d'instructions. Seuls les *Step* sont spécifiques au SUT. Les *Log*, *Trace* et *Check* sont dépendants du test réalisé. Nous détaillons par la suite chaque type d'instructions et présentons les extraits de la carte conceptuelle qui leur sont associés. Lors de la présentation des instructions et de leur syntaxe, les bouts de phrases délimités par le caractère $ sont les parties variables et les portions entre [et] sont optionnelles.

4.4.1 Processus de création de la carte conceptuelle

La carte conceptuelle présentée dans cette section a été réalisée par itérations successives avec les experts en intégration de l'ATA 42, nous permettant d'affiner les instructions déjà découvertes et d'ajouter des instructions couvrant de nouveaux besoins au fil des itérations. Quatre itérations, chacune marquée par une réunion avec les experts testeurs, ont été nécessaires pour atteindre les résultats présentés. Outre l'approbation des propositions de formalisation explicitées lors de la précédente réunion, toute réunion, de l'ordre de trois heures, avait pour objectif d'identifier de nouvelles instructions couvrant les nouveaux besoins élicités. Un travail de synthèse, prenant la forme d'une mise à jour de la carte conceptuelle, était ensuite entrepris afin de s'assurer que tous les nouveaux besoins soient correctement pris en compte. Une nouvelle itération pouvait ainsi prendre place après envoi de la nouvelle carte conceptuelle aux experts testeurs.

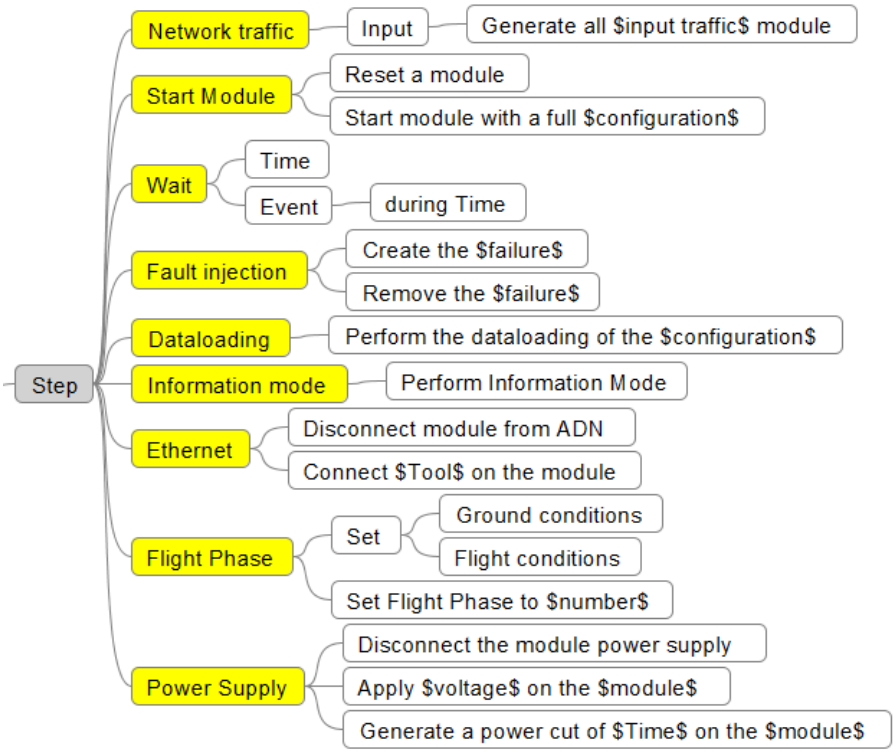
Pour identifier un besoin à formaliser par la carte conceptuelle, les discussions lors des réunions se sont basées sur les statistiques présentées à la section précédente. A partir des verbes les plus utilisés pour un type de phrase, nous présentions aux experts un ensemble d'exemples de phrases ou de patrons de phrases identifiés lors de nos études préliminaires. Ces exemples ont permis d'alimenter les discussions afin de valider en séance s'ils correspondaient à un besoin de formalisation primordial ou non.

Les phrases composant les procédures textuelles actuelles sont très hétérogènes. Dans ces procédures, des phrases exprimant la même notion ont très souvent une syntaxe différente car elles sont décrites en langage naturel par des testeurs différents. Ainsi, plusieurs patrons proposés peuvent formaliser le même besoin. Les discussions avec les testeurs visaient à s'accorder sur la syntaxe d'une instruction afin qu'elle soit proche de leurs habitudes. Côté sémantique, il s'agissait aussi de s'assurer que l'intention de l'instruction soit correctement interprétée par les utilisateurs du DSTL.

4.4.2 Instructions `Step`

La figure 4.14 correspond à la carte conceptuelle des instructions `Step` découpées en neuf catégories.

FIGURE 4.14 – Carte conceptuelle des instructions `Step`



Chacune des neuf catégories classant les instructions `Step` traite d'un aspect particulier, d'un comportement ou d'une fonction du système testé qu'il est nécessaire de formaliser. Dans ces catégories, les instructions expriment les différentes actions nécessaires à la description des cas de test d'intégration du domaine. Les neuf catégories proposées à la figure 4.14 sont le trafic réseau (*Network traffic*), le démarrage du module (*Start Module*), l'attente (*Wait*), l'injection de pannes (*Fault injection*), le téléchargement des configurations (*Dataloading*), le mode information (*Information mode*), la connexion au réseau (*Ethernet*), la gestion des phases de vol (*Flight Phase*) et l'alimentation électrique (*Power Supply*).

La catégorie trafic réseau (*Network traffic*) gère les paquets transportant des messages *via* différents média sur le réseau connectant le banc au calculateur IMA. Ces paquets seront générés par un outil spécifique avant d'être transmis au calculateur. L'instruction `GenerateInputTraffic` formalise les besoins de cette catégorie. Pour cette instruction, le média de communication des paquets générés est variable. Le testeur pourra décider si les paquets seront transmis par ethernet (AFDX), ou par les bus CAN, ARINC par exemple.

Pour la catégorie démarrage du module (*Start Module*), deux instructions existent : `ResetModule` et `StartModule`. Ces instructions gèrent respectivement le redémarrage et le démarrage du module. Le redémarrage du module est formalisé par la syntaxe `Perform a module reset`. Le démarrage est formalisé par la syntaxe `Start the module` et est complété d'une partie optionnelle associant le module testé à une configuration. Cette configuration est choisie parmi un ensemble de configurations possibles. Le choix de la configuration est traduit dans le langage par `with the configuration $Configuration$`.

La catégorie attente (*Wait*) gère deux cas de figure : le premier permet de mettre en pause le test pendant une durée définie et le deuxième formalise l'attente d'un évènement où il est aussi possible de notifier un temps d'attente. Pour cette catégorie, nous proposons donc une instruction `Wait` prenant deux formes : la syntaxe `Wait for $time$` formalise le premier cas et la syntaxe `Wait for $event$ during $time$` formalise le deuxième.

La catégorie injection de pannes (*Fault injection*) donne la possibilité d'introduire des pannes lors de l'exécution d'un test afin de vérifier que le SUT sait les identifier pour y répondre convenablement. Ces pannes sont provoquées par la transmission au SUT de paquets contenant des trames erronées. Lors de la réception de ces trames, le système est censé lever un signal informant les autres systèmes, dont celui centralisant la gestion des pannes, qu'un comportement inattendu a été détecté. L'injection d'une panne se fait par l'instruction `CreateFailure` dont la syntaxe est `Create $failure$`. La suppression d'une panne se fait *via* l'instruction `RemoveFailure` qui propose la syntaxe `Remove $Failure$`. Le type de panne souhaité doit être formalisé à la place du concept `$failure$`. La liste des pannes utilisées pour un cas de test est à produire préalablement par le testeur.

La catégorie téléchargement des configurations (*Dataloading*) est composée d'une instruction réalisant le téléchargement de la configuration souhaitée. Cette instruction se nomme `PerformDataloading` ; elle utilise la syntaxe `Perform dataloading of the configuration $Configuration$` à la manière de l'instruction `StartModule`, mais pour cette

dernière le choix de la configuration n'est pas optionnel.

La catégorie mode d'information (*Information mode*) fait appel à une fonction spécifique des systèmes IMA permettant d'obtenir les détails de la configuration globale du SUT et de chacune des applications logicielles qu'il héberge. Il est possible d'interroger l'état du module testé par la syntaxe `Perform Information Mode`.

La catégorie connexion au réseau (*Ethernet*) gère la connexion et la déconnexion du SUT au réseau de communication du banc de test. Deux instructions composent donc cette catégorie. L'instruction `ConnectTool` permet de connecter un outil au module ; elle est retranscrite par `Connect $Tool$ on the module`. Cette instruction référence un des outils préalablement choisis pour former la liste des outils utilisés par le chapitre. L'instruction `DisconnectFromNetwork` propose la syntaxe concrète `Disconnect the module from the network` pour formaliser l'action de déconnexion du module au réseau.

La gestion des phases de vol (*Flight Phase*) obéit à deux instructions. La première instruction (`SetFlightPhase`) permet de notifier au SUT que le comportement attendu est celui au sol ou en vol. Cette instruction propose la syntaxe `Set module in $Ground|Flight$ conditions`. Pour la deuxième, l'instruction attend le numéro exact de la phase souhaitée. Cette deuxième instruction, plus précise, permet d'indiquer par sa syntaxe le numéro associé à l'état général du SUT : `Set flight phase to $phase number$`. Les états possibles sont numérotés de 1 à 12 pour les calculateurs IMA testés.

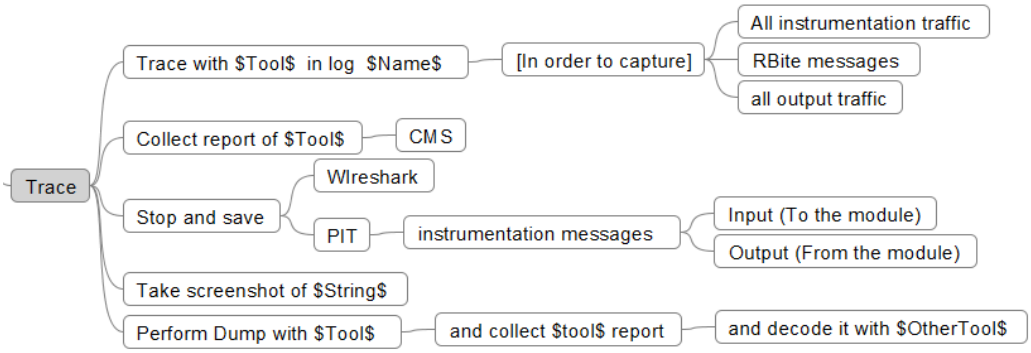
La catégorie de l'alimentation électrique (*Power Supply*) répond à trois besoins. Le premier est de mettre le système sous tension en choisissant le voltage appliqué par l'alimentation électrique. L'instruction `ApplyPowerSupply` y répond par le biais de l'écriture `Apply $integer$ V on the module`. Le deuxième est de déconnecter un module de son alimentation. L'instruction `DisconnectPowerSupply` correspond à ce besoin en utilisant la syntaxe suivante : `Disconnect $primary A/C|back-up A/C$ power supply`. Enfin, il est possible de générer des coupures électriques pendant un certain laps de temps grâce à la syntaxe `Generate power cut $duration$ on module`. Le concept optionnel `duration` est formalisé par `of $number$ $unit$`, où `$number$` correspond à la durée pendant laquelle le module ne doit plus être alimenté.

4.4.3 Instructions `Trace`

La figure 4.15 présente la carte conceptuelle dédiée aux instructions typées `Trace`. Le principal besoin identifié pour ce type est de pouvoir démarrer et éteindre les outils réalisant les traces, réaliser des captures d'écran ou encore de sauvegarder *a posteriori* des données produites par un outil. Les instructions `Trace` implémentées dans le DSTL sont au nombre de cinq comme le montre la carte conceptuelle. L'automatisation des actions de *Trace* passe par la formalisation des phrases les décrivant.

Le démarrage est formalisé *via* la syntaxe `Trace with $Tool$ in $FileName$` et l'arrêt par `Stop and save $Tool$ trace`. La collecte d'un rapport est mise en œuvre par l'instruction `Collect report` et est formalisée dans le langage par la syntaxe `Collect`

FIGURE 4.15 – Carte conceptuelle des instructions `Trace`



`$Tool$ report`. La réalisation d'une capture d'écran admet pour syntaxe `Take screenshot of $HMI`[3]`$`. Ces captures portent sur des interfaces d'outils du banc identifiées par l'instruction. Il est possible de formaliser la récupération de l'état de la mémoire interne du SUT (*dump*) à l'exécution *via* l'instruction `PerformDump` du DSTL. Cette instruction peut être combinée à l'action permettant de décoder ce *dump* grâce à un autre outil.

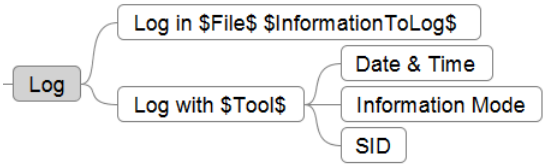
FIGURE 4.16 – Exemples d'instructions `Trace`

```
Trace with Wireshark in file Procedure Report
Stop Wireshark trace
Collect Wireshark report
Take screenshot of  Wireshark with all failures displayed
Perform an MVN dump with PIT
```

La figure 4.16 présente un exemple de chaque instruction `Trace` disponible pour l'ATA 42. La liste des outils manipulables par ces instructions est restreinte à ceux référencés par le chapitre de test auquel les instructions appartiennent.

4.4.4 Instruction `Log`

FIGURE 4.17 – Carte conceptuelle de l'instruction `Log`



3. Human-Machine Interface

La figure 4.17 décrit les besoins identifiés pour les `Log`. L'enregistrement ponctuel des données du test peut se faire soit grâce à la phrase *Log in TRA* [4] soit par *Log with $Tool$*. Dans le premier cas, les informations concernées sont décrites par `$InformationToLog$`. Cette phrase est liée optionnellement à une clause définissant les cas pour lesquels le *log* doit être exécuté. Les informations devant être stockées varient pour chaque chapitre de test. Dans le deuxième cas, un outil est utilisé pour réaliser l'enregistrement. Cet outil permet de récupérer la date et l'heure à tout moment du test, ainsi que les informations sur le SUT *via* l'information mode ou encore son SID [5].

Nous avons formalisé les deux besoins identifiés ci-dessus par l'instruction `LogInFile`. Cette instruction formalise la réalisation d'un *log* dans un fichier. Pour utiliser cette instruction, il faut au préalable que le testeur ait spécifié, dans un fichier séparé, la liste des données de test à enregistrer pour un ensemble de tests ainsi que la liste des fichiers dans lesquels il est nécessaire de les d'enregistrer.

FIGURE 4.18 – Liste des informations utilisables par l'instruction `LogInFile`

```
Information to log:
message display by the Dataloader
the PN of the loads seen in the SID and in the INFORMATION MODE
the failure use for the test (i.e. FMC, FSI and FSI title defined in [LTR's_FDS]).
all the priority.
all the accused I/O.
```

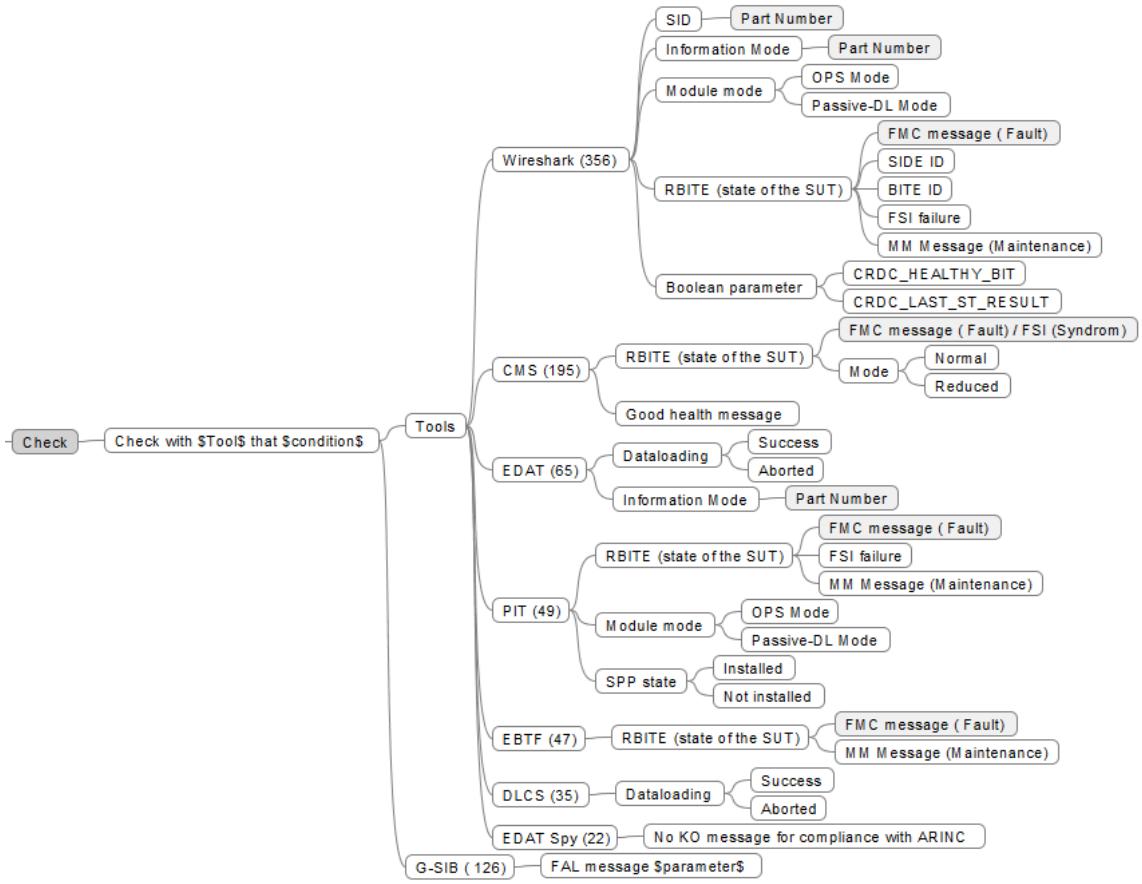
La figure 4.18 est un exemple de liste décrivant les informations qu'il est possible de loguer pour un contexte donné. Il est possible d'ajouter des éléments à cette liste à n'importe quel moment du développement et de les utiliser dans l'instruction `LogInFile` d'un chapitre formalisé. La liste des fichiers accueillant les informations enregistrées est formalisée de la même manière que la liste des informations à loguer. Les éléments de cette liste peuvent notifier l'outil à utiliser pour récolter les informations. Une même liste peut être utilisée pour plusieurs procédures ; l'importation de plusieurs listes de ce type permet de les utiliser conjointement.

FIGURE 4.19 – Exemple d'instruction `LogInFile`

```
[Log] Log in  TRA the PN of the loads seen in the SID and in the INFORMATION MODE
```

La figure 4.19 montre un exemple d'instruction `LogInFile` utilisant le fichier *TRA* pour enregistrer les informations souhaitées. Cette instruction utilise une des informations de la liste de la figure 4.18.

FIGURE 4.20 – Carte conceptuelle de l'instruction `Check`



4.4.5 Instruction `Check`

Les besoins des instructions `Check` sont englobés en une seule phrase. Les procédures étudiées utilisent des outils du banc pour répondre aux besoins de validation des comportements du SUT. Le patron de la phrase proposé pour formaliser les `Check` correspond au deuxième niveau de la hiérarchie de la carte conceptuelle de la figure 4.20. Il manipule la variable `$Tool$`, faisant référence à l'outil utilisé pour cette instruction, et la variable `$condition$` associée aux conditions de validation du comportement du SUT lors d'un cas de test.

La carte conceptuelle dédiée au `Check` met aussi en avant la liste des outils qui ont été répertoriés dans les procédures textuelles et les fonctionnalités qu'ils proposent. Ces outils sont visibles dans la carte conceptuelle en dessous du concept *Tools*. Le niveau hiérarchique le plus bas de la carte conceptuelle identifie les différents états ou comportements du système à vérifier pour chaque fonctionnalité.

L'instruction `CheckWithTool` du DSTL ATA 42 couvre les besoins des `Check`. Cette instruction utilise la composition avec le langage `Tools` pour référencer les outils du banc. Elle formalise l'appel à un outil et à un comportement ou un état du système que cet outil propose de vérifier.

FIGURE 4.21 – Liste des informations utilisables par l'instruction `CheckWithTool`

```
Checkable items:
CRDC P/N, CSW P/N, CT P/N and RBCT P/N are compliant with LTR.
no check KO is reported in order to assess compliance with ARINC 615A-2.
ABORT command with Uploading Operation Status Code = 0x1004(aborted by DLCS) is sent to CRDC.
dataloading operation is aborted at first attempt.
the Upload Operation Status Code in the last .LUS sent by CRDC contains the code "1004".
SID and Information Mode contain "NOT LOADED" for the PN of the load.
dataloading is performed and finished.
module is in Passive-DL Mode.
SID contains PN and SN for module and PN for CSW.
Maintenance Message is sent with a period from 9 to 10 seconds.
CRDC sends only the FMC triggered.
priority in MM is in line with [LTR's_FMD_SPEC].
LRU accusation in MM is in line with [LTR's_FMD_SPEC]: number and order of accused LRU.
ATAxx accusation in MM is as expected in [LTR's_Pin_Allocation]: LRU code or Functional_Designation columns.
I/O accusation in MM is in line with [LTR's_FMD_SPEC] and correspond to the involved pin.
RBITE MM content matches exactly with FMD file of [LTR's_FMD], from word 2 to 14 of ABD0100.1.4-2203-F requirement.
MM is sent no more than 3 times with disappeared status.
all traffic from the module is captured (RBITE message, instrumentation traffic).
```

La syntaxe concrète `Check with $Tool$ that $condition$` est illustrée par la figure 4.22. Cet exemple d'instruction `CheckWithTool` utilise l'outil `EDAT` et la condition formalisée par un des éléments de la liste proposée à la figure 4.21.

Aucun oracle n'a été mis en place pour automatiser l'exécution des tests car actuellement les outils sont utilisés manuellement. Un effort important est demandé pour produire les scripts automatisant les vérifications réalisées durant un test grâce à ces outils. Les

4. *Test Result Analysis*
5. *System Identification Data*

FIGURE 4.22 – Exemple d'instruction `CheckWithTool`

```
[Check] Check with EDAT that module is in Passive-DL Mode.
```

différents états ou comportements à vérifier sont spécifiques à chaque test, ce qui augmente grandement le nombre des possibilités. C'est pourquoi, nous souhaitons proposer un formalisme simple et permissif pour couvrir un maximum de cas possibles. Un utilisateur devra pouvoir formaliser les cas illustrés par la carte conceptuelle et pourra aussi formaliser de nouvelles vérifications.

Pour couvrir ces nombreux cas, nous proposons de définir, dans un fichier séparé, la liste de comportements ou d'états qu'il sera possible de vérifier. Nous laissons une entière liberté à l'utilisateur pour les décrire grâce à une simple chaîne de caractères. Cette chaîne de caractères correspondra au nom du concept qui sera référençable par l'instruction `CheckWithTool`. Dans le but d'une automatisation complète de l'exécution, chaque comportement ou état à vérifier devra alors être lié au script pilotant l'outil en charge de la vérification. La figure 4.21 présente un exemple de liste d'éléments à vérifier introduits par `Checkable items`.

4.5 Première validation par les testeurs

Une version du DSTL ATA 42 a été fournie aux experts testeurs pour qu'ils traduisent des procédures décrites en langage naturel en procédures formalisées. Ces experts n'ont pas de compétences spécifiques en informatique et n'ont pas non plus de connaissances particulières sur l'utilisation de langages projectionnels, mises à part les quelques démonstrations réalisées lors des réunions d'avancement.

FIGURE 4.23 – Procédure formalisée

```
Test procedure: Test Procedure with DSL - DL and RBCT
ID: 911
Summary:
These three test procedure have been created by ATA42 DO and respond to Dataloading and RBCT test ojectives.

Responding to: Lab Test Request CRDC integration

Test chapters:
 PROC - CRDC - SIS CONTENT
 PROC - CRDC - DATALOADING ABORT
 PROC - CRDC - FAILURE FMC 2001-2500
```

La figure 4.23 retranscrit la formalisation de la procédure *Test Procedure with DSL - DL and RBCT*. La procédure est composée de trois chapitres définis dans des fichiers séparés détaillés par la suite. Seuls les noms des chapitres sont visibles dans la procédure pour qu'un testeur n'ait pas à chercher un cas de test parmi les dizaines voire centaines de pages des fichiers *Word*. Cette procédure a été élaborée en un peu plus de deux heures par

un ingénieur en test d'intégration de l'ATA 42. La procédure est composée d'un nom, de l'identifiant 911, d'un résumé et elle référence le *LTR* contenant les objectifs auxquels les chapitres répondent.

FIGURE 4.24 – `Lab Test Request` formalisé

```
Lab Test Request CRDC integration
ID: 1
System under test:
CRDC

Objectives:
ID: OBJ-CR-008 Name: SIS Content
ID: OBJ-DL-007 Name: Dataloading Abort
ID: CRDC-RBCT-NM-04 Name: Failure FMC 2001-2500
```

La figure 4.24 montre la liste des objectifs formalisés en langage *core*. Dans cette procédure, chaque chapitre ne couvre qu'un seul objectif du `LTR`. La matrice de traçabilité des objectifs est produite grâce à la génération de la table liant les chapitres de test d'une procédure aux objectifs qu'ils couvrent (section 4.2.3).

Les figures 4.25, 4.26 et 4.27 sont les trois chapitres de test référencés dans la procédure décrite ci-dessus. Chaque chapitre valide un comportement différent du système testé par une suite d'instructions du DSTL ATA 42.

Le testeur ayant formalisé la procédure avec nos langages propose plusieurs points d'amélioration concernant l'utilisabilité et évoque de nouveaux besoins à intégrer au DSTL ATA 42. Nous résumons ces propositions par une liste contenant les principales idées et remarques. Nous proposons pour chacune d'elles des actions d'amélioration pour la prochaine itération du développement du jeu d'instructions ATA 42.

- Dans les `Step`, il est impossible de faire un « *Power Off* » du SUT alors qu'on dispose des instructions `StartModule` et `ResetModule`. D'après cette remarque, il semble important de définir une instruction permettant l'arrêt du SUT, par exemple `StopModule`. La syntaxe concrète de cette instruction a besoin d'être élaborée en consensus avec les experts de l'ATA 42 prenant part à la conception du langage.
- Aucune instruction `Step` du DSTL ne permet de s'assurer (*Ensure to*) que des conditions préalables au test sont bien réunies. Par exemple, il s'agirait de s'assurer que le SUT et le banc sont assemblés pour œuvrer ensemble avant de poursuivre l'exécution d'un test. Cependant, nous avons remarqué que les phrases commençant par *Ensure to* sont utilisées dans les procédures textuelles pour vérifier que les actions manuelles réalisées précédemment ont été correctement conduites. Ces actions ne doivent pas être confondues avec les instructions `Check` utilisées pour valider les objectifs de test. Les instructions *Ensure* seront à prendre en compte car elles ont pour objectif de s'assurer que les conditions sont réunies pour qu'un test puisse continuer son exécution, sans risque d'endommager le SUT par exemple.
- L'instruction `CheckWithTool` autorise uniquement des vérifications par le biais d'une

FIGURE 4.25 – Chapitre de test *SIS content*

```
Test chapter: PROC - CRDC - SIS CONTENT
ID: 1
Objective: SIS content
           Check that SIS content is the same than Information Mode, SID and FIND.

Test will be perform on: 2IMAF

Covered Objectives:
OBJ-CR-008

Tool:
EDAT
EDAT Spy
Wireshark
SIS Reader

<no setup>
TEST: SIS content
[Step] Perform dataloading with the configuration specified in LTR.
[Step] Apply  0 on the module
[Step] Disconnect the module from the network
[Step] Connect  EDAT to the module
[Trace] Trace with Wireshark in file TRA
[Step] Start the module <no withConfiguration>
[Step] Wait for a SID <no duringTime>
[Step] Perform Information Mode with  EDAT
[Trace] Stop Wireshark trace
[Step] Apply  0 on the module
[Step] Connect  SIS Reader to the module
[Trace] Take screenshot of  SIS Reader
[Check] Check with Wireshark that CRDC P/N, CSW P/N, CT P/N and RBCT P/N are compliant with LTR.
[Check] Check with EDAT Spy that no check KO is reported in order to assess compliance with ARINC 615A-2.
```

FIGURE 4.26 – Chapitre de test *Dataloading abort*

```
Test chapter: PROC - CRDC - DATALOADING ABORT
ID: 2
Objective: Dataloading abort
           Check that CRDC uploading can be aborted by DLCS.

Test will be perform on: 2IMAF

Covered Objectives:
OBJ-DL-007

Tool:
EDAT
EDAT Spy
Wireshark

<no setup>
TEST: Dataloader abort on empty CRDC
[Step] Start the module with the configuration not loaded.
[Trace] Trace with Wireshark in file TRA
[Step] Wait for a SID <no duringTime>
[Step] Perform dataloading with the configuration specified in LTR.
[Step] Abort dataloading
[Check] Check with EDAT that ABORT command with Uploading Operation Status Code = 0x1004(aborted by DLCS) is sent to CRDC.
[Check] Check with EDAT that dataloading operation is aborted at first attempt.
[Check] Check with Wireshark that the Upload Operation Status Code in the last .LUS sent by CRDC contains the code "1004".
[Log] Log in  TRA message display by the Dataloader
[Step] Wait for a SID <no duringTime>
[Step] Perform Information Mode with  EDAT
[Check] Check with Wireshark that SID and Information Mode contain "NOT LOADED" for the PN of the load.
[Log] Log in  TRA the PN of the loads seen in the SID and in the INFORMATION MODE
[Step] Perform dataloading with the configuration only CSW.
[Check] Check with EDAT that module is in Passive-DL Mode.
[Step] Wait for a SID <no duringTime>
[Check] Check with Wireshark that SID contains PN and SN for module and PN for CSW.
[Trace] Stop Wireshark trace
[Check] Check with EDAT Spy that no check KO is reported in order to assess compliance with ARINC 615A-2.
```

FIGURE 4.27 – Chapitre de test *Failure FMC 2001-2500*

```
Test chapter: PROC - CRDC - FAILURE FMC 2001-2500
ID: 3
Objective: External I/Os failure linked to FMC 2001-2500
           Check that CRDC reports correctly failures with FMC in range 2001-2500.

Test will be perform on: 2IMAF

Covered Objectives:
CRDC-RBCT-NM-04

Tool:
EBTF
G-SIB
Wireshark

<no setup>
TEST: FMC 2001-2500
[Reminder] This test must be performed for one FSI per type of interface (e.g. DSO_28V/OPN_250mA, DSO_GND/OPN_1.5A, etc…) on 1 CRDC position.
[Step] Generate all CRDC input traffic
[Trace] Trace with Wireshark in file TRA
[Check] Check with Wireshark that all traffic from the module is captured (RBITE message, instrumentation traffic).
[Step] Start the module with the configuration specified in LTR.
[Step] Create failure as specified in PROC FOR FAILURE GENERATION.
[Log] Log in  TRA the failure use for the test (i.e. FMC, FSI and FSI title defined in [LTR's_FDS]) and the CRDC interface(s) involved (e.g. pin reference).
[Check] Check with Wireshark that Maintenance Message is sent with a period from 9 to 10 seconds.
[Check] Check with EBTF that CRDC sends only the FMC triggered.
[Trace] Take screenshot of  EBTF with all failures displayed and full content of "Fault Message Display" column.
[Check] Check with EBTF that LRU accusation in MM is in line with [LTR's_FMD_SPEC]: number and order of accused LRU.
[Reminder] The ATAxx accusation check is not available if FUNCTIONAL_DESIGNATION = SEVERAL in [LTR's_Pin_Allocation].
[Check] Check with EBTF that ATAxx accusation in MM is as expected in [LTR's_Pin_Allocation]: LRU code or Functional_Designation columns.
[Reminder] Functional descriptions linked to LRU_Code/LRU_Decoding_Key associations, are specified in "LRU Table" stored on Bridge tool.
[Log] Log in  TRA all the priority.
[Check] Check with EBTF that I/O accusation in MM is in line with [LTR's_FMD_SPEC] and correspond to the involved pin.
[Log] Log in  TRA all the accused I/O.
[Check] Check with Wireshark that RBITE MM content matches exactly with FMD file of [LTR's_FMD], from word 2 to 14 of ABD0100.1.4-2203-F requirement.
[Step] Remove failure
[Check] Check with Wireshark that MM is sent no more than 3 times with disappeared status.
[Trace] Stop Wireshark trace
```

fonction proposée par un outil. Il serait approprié de pouvoir vérifier au cours du test des informations provenant des interfaces ou de captures d'écrans. A l'instar de la formalisation des outils et de leurs fonctionnalités par le langage *Tools*, une solution est envisageable pour de telles interfaces. Ces interfaces et captures d'écrans pourraient alors être utilisées au sein des instructions `Check`.

Les retours d'expérience fournis par les experts testeurs nous permettront d'apporter d'améliorer le jeu d'instructions du DSTL ATA 42 en modifiant certaines instructions ou en ajoutant des instructions pour couvrir de nouveaux besoins de formalisation.

4.6 Conclusion

La mise en œuvre des tests de l'ATA 42 est à ce jour manuelle, que ce soit au niveau de leur exécution ou de leur verdict. Dans ce contexte, nous avons développé un deuxième DSTL dédié à leur formalisation. Ce DSTL se base sur une structure générique des procédures communes à plusieurs domaines, implémentée par le langage *core*. Le langage *core* est indépendant des jeux d'instructions pour permettre sa réutilisation pour chacun des ATA. Il propose une classification des instructions en cinq types et réalise la traçabilité des objectifs de test. Le jeu d'instructions du DSTL ATA 42 a été développé pour être couplé au langage *core*. Ce jeu d'instructions couvre les besoins révélés par la carte conceptuelle des instructions classées selon les types proposés par le langage *core*.

Les experts ont été très impliqués durant nos travaux pour élaborer la carte conceptuelle, ce qui nous a permis d'aboutir à une version du DSTL ATA 42. Ce DSTL permet de formaliser des tests grâce à seize instructions `Step`, cinq instructions `Trace` et deux instructions génériques, une pour `Log` et une pour `Check`, en seulement quatre itérations. Cette expérimentation nous a montré qu'il était facile de combiner des langages munis d'éditeurs projectionnels, ce qui simplifie la mise en place d'une approche de programmation orientée langages.

L'utilisation d'outils de traitement du langage naturel n'a pas abouti aux résultats escomptés. La formalisation de procédures décrites en langage naturel est très complexe. Sans l'aide des experts, il semble illusoire de différencier les phrases utiles à l'intention d'un test de celles détaillant l'exécution manuelle des actions. Il n'est pas pertinent de formaliser les détails de mise en œuvre car ils interfèrent avec les instructions décrivant l'intention du test. Nous avons dû, pour converger sur une première version du DSTL, mettre en place un processus empirique impliquant les experts dans l'élaboration des instructions.

Pour la suite du développement, le DSTL ATA 42 doit prendre en compte des contraintes sémantiques afin que les utilisateurs produisent des tests cohérents. Dans l'optique de lui ajouter une sémantique opérationnelle, il est aussi nécessaire de l'intégrer à la chaîne de génération du *framework*. Il doit être couplé à un générateur de code en langage pivot, avant sa transformation en code de plus bas niveau. Un grand nombre des instructions du DSTL ATA 42 se traduiront par un `call` du langage pivot faisant appel à un outil du banc*via*

un script qui automatisera son pilotage. Le langage pivot doit aussi permettre l'exécution de tests semi-automatiques avec l'apparition de *pop-ups* bloquant l'exécution d'un test et laissent le soin au testeur de réaliser une action manuelle.

Ces travaux tendent à être expérimentés à d'autres domaines avioniques et nous souhaitons formaliser des procédures où l'automatisation des tests est plus avancée. Ils nous ont permis d'appréhender les difficultés de formalisation des procédures d'intégration avionique. Nous envisageons de généraliser notre processus collaboratif d'élaboration d'un DSTL avionique en l'appliquant à d'autres domaines et ainsi étendre les possibilités de notre *framework*. Tous ces travaux vont se poursuivre par le projet DGAC ESTET (*Early Systems TEsTing*) dont l'objectif concerne l'automatisation des tests système au plus tôt dans le cycle de développement.

Conclusion

La formalisation et l'automatisation des tests d'intégration dans un contexte avionique sont un enjeu majeur car les procédures de test sont coûteuses à concevoir et les systèmes sont coûteux à tester. De plus, la réalisation d'un avion met en œuvre différents domaines ou ATA, chacun lié à un métier et des experts spécifiques. Afin d'impliquer les experts en test système dans le processus de conception des procédures de test, nous avons choisi une démarche Agile « à la BDD » pour formaliser les différents domaines métier à l'aide de langages ubiquitaires et pour transformer ces procédures de test en scripts exécutables par l'adjonction de code glu dans un *framework* dédié. Un langage ciblé sur un domaine et dont sa syntaxe est manipulable par des experts testeurs est un DSTL.

Nous pensons qu'il est important que les concepts du langage utilisé soient en adéquation avec les concepts du domaine testé. De plus, un langage dont la syntaxe est proche de celle utilisée habituellement dans les procédures informelles sera plus facilement adopté par les experts. L'aspect projectionnel des éditeurs proposés pour les langages du *framework* rend l'utilisation des ces langages plus intuitive. Les utilisateurs sont guidés par la structure des concepts du langage et certaines erreurs de syntaxe sont évitées *a priori*. L'automatisation des tests est rendue possible par la réutilisation de patrons de transformation d'instructions simples en séquences d'instructions scriptables et la donnée d'un langage pivot permettant de minimiser le nombre de transformations à écrire et d'en simplifier leur écriture.

Dans ce travail, notre principale difficulté a été d'appréhender les tenants et les aboutissants de l'intégration des systèmes avioniques. Par exemple, la prise en compte des spécificités des calculateurs IMA, la complexité des procédures de test fournies en langage naturel et conçues pour ces calculateurs, l'appréhension des outils disponibles par un banc de test dédié et le caractère exotique des langages de scripts disponibles sur un banc illustre l'ampleur des pré-requis à assimiler et prendre en compte avant de proposer une solution adéquate. L'émergence du DSTL ATA 21 a été grandement simplifiée car ces procédures étaient déjà automatisées avec un langage *ad hoc*. La conception du DSTL ATA 42 quant à elle a été le fruit d'un processus empirique et l'automatisation de ces procédures reste à définir.

L'exécution automatique des procédures ATA 42 entraînera la manipulation de nombreux outils nécessaires à la collecte de trames réseau et le développement de scripts pour

les piloter. L'automatisation des verdicts, quant à elle, dépendra étroitement de l'observabilité des systèmes à tester, des résultats à analyser et des possibilités automatiques d'analyse de ces résultats. Certaines contraintes sémantiques ont été développées pour ces langages afin de guider les utilisateurs à produire des scénarios de test cohérents. Cette cohérence est indispensable pour garantir l'intégrité des systèmes sous test lors de l'intégration. Des règles de bonne formation de scénarios de test pour l'ATA 42 sont à éliciter avec l'aide des experts.

Notre processus empirique de conception de DSTL doit être rejoué sur d'autres ATA plus automatisables, ce qui nous permettra de mieux le formaliser et de le rendre plus systématique. Les travaux futurs seront tournés vers l'élaboration d'un processus outillé dédié à l'encadrement de l'émergence des DSTL. Un premier effort peut être initié pour définir la sémantique des cartes conceptuelles que nous utilisons. Du point de vue de l'ingénierie des langages, nous envisageons de faire davantage participer les experts en leur permettant d'améliorer eux-mêmes la sémantique de leurs DSTL, en fournissant par exemple une base de procédures incorrectes sous l'éditeur de procédures.

La preuve du concept de notre approche a été validée avec succès lors de la démonstration finale du projet ACOVAS. Elle concernait toute la chaîne de langages dédiée à l'ATA 21. La gestion Agile du projet ACOVAS lui-même a offert les conditions d'un partenariat étendu avec différentes équipes de testeurs, ce qui nous permet aujourd'hui une dissémination de notre approche poursuivie au travers d'un projet DGAC avec Airbus.

Bibliographie

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] Aircraft Data Network. Avionics Full Duplex Switched Ethernet (AFDX) Network. *ARINC Specification 664p7*, 7, 2005.

[3] A. Alsumait, A. Seffah, and T. Radhakrishnan. Use case maps : A visual notation for scenario-based user requirements. In *Proceedings of the 10th International Conference on Human-Computer Interaction, June*, pages 22–27, 2003.

[4] D. Amyot. Introduction to the user requirements notation : Learning by example. *Comput. Netw.*, 42(3) :285–301, June 2003.

[5] C. M. Ananda. General aviation aircraft avionics : Integration system tests. *IEEE Aerospace and Electronic Systems Magazine*, 24(5) :19–25, May 2009.

[6] ARINC. Arinc specification 429 part 1. *ARINC Inc*, pages 1–18, 2004.

[7] J. Barnett, R. Akolkar, R. J. Auburn, D. C. Burnett, T. Lager, and J. Roxendal. State Chart XML (SCXML) : State Machine Notation for Control Abstraction. WWW Consortium, WD-scxml-20140529, May 2014.

[8] K. Beck. *Test-driven development : by example*. Addison-Wesley Professional, 2003.

[9] K. Beck and C. Andres. *Extreme Programming Explained : Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[10] J. Bentley. Programming Pearls : Little Languages. *Commun. ACM*, 29(8) :711–721, Aug. 1986.

[11] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing : A controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 763–774, New York, NY, USA, 2016. ACM.

[12] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[13] A. Blackwell, C. Britton, A. Cox, T. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. Nehaniv, M. Petre, et al. Cognitive dimensions of notations : Design tools for

cognitive technology. *Cognitive technology : instruments of mind*, pages 325–341, 2001.

[14] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[15] R. Bussenot, H. Leblanc, and C. Percebois. Integration Testing based on Behavior Driven Development (ETSI User Conference on Advanced Automated Testing (UCAAT 2017), Berlin (Germany), 11/10/2017-13/10/2017).

[16] R. Bussenot, H. Leblanc, and C. Percebois. A Domain Specific Test Language for Systems Integration. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, pages 1–10, article 16, New York, NY, USA, 2016. ACM.

[17] R. Bussenot, H. Leblanc, and C. Percebois. Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach. In *13th System of Systems Engineering IEEE Conference (SoSE)*, 2018.

[18] J. Bywater. Quant v0.8 documentation, 2015. `https://pythonhosted.org/quant/dsl.html`.

[19] F. Campagne. *The MPS Language Workbench*, volume 1. FABIEN CAMPAGNE, 2013-2014.

[20] F. Campagne. *The MPS Language Workbench*, volume 2. FABIEN CAMPAGNE, 2015.

[21] M. Cohn. *User Stories Applied : For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[22] École Polytechnique Fédérale de Lausanne. Scala : Object-oriented meets functional. `https://www.scala-lang.org/`.

[23] Computer Desktop Encyclopedia. general-purpose language., 1981-2015. `https://encyclopedia2.thefreedictionary.com/general-purpose+language`.

[24] Cucumber. Cucumber. `https://cucumber.io/`.

[25] H. C. Cunningham. A Little Language for Surveys : Constructing an Internal DSL in Ruby. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 282–287, New York, NY, USA, 2008. ACM.

[26] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[27] B. David. *Selenium 2 Testing Tools : Beginner's Guide*. Packt Publishing, 2012.

[28] C. deLong. Mil-std-1553b digital time division command/response multiplex data bus. In *Industrial Communication Technology Handbook, Second Edition*, pages 1–25. CRC Press, 2014.

[29] Eclipse. Eclipse Modeling Framework Technology (EMFT). `https://eclipse.org/modeling/emft/`.

[30] Eclipse. Graphical Modeling. `http://www.eclipse.org/modeling/graphical.php`.

[31] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. *The State of the Art in Language Workbenches*, pages 197–217. Springer International Publishing, Cham, 2013.

[32] ETSI. TTCN-3. `http://www.ttcn-3.org/`.

[33] European Organisation for Civil Aviation Equipment. Eurocae. `https://www.eurocae.net/`.

[34] E. Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[35] B. Filyner. Open systems avionics architectures considerations. *IEEE Aerospace and Electronic Systems Magazine*, 18(9) :3–10, Sept 2003.

[36] M. Fowler. Language workbenches : The killer-app for domain specific languages. 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[37] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[38] U. Freund. Mulit-level system integration based on autosar. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 581–582, New York, NY, USA, 2008. ACM.

[39] R. Fulton and R. Vandermolen. *Airborne Electronic Hardware Design Assurance : A Practitioner's Guide to RTCA/DO-254*. CRC Press, Inc., Boca Raton, FL, USA, 2014.

[40] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3.4), 1993.

[41] J. Garrido, J. Zamorano, and J. A. de la Puente. ARINC-653 Inter-partition communications and the ravenscar profile. *ACM SIGAda Ada Letters*, 35(1) :38–45, 2015.

[42] GNU. GNU Bison. `https://www.gnu.org/software/bison/`.

[43] J. Grossmann, I. Fey, A. Krupp, M. Conrad, C. Wewetzer, and W. Mueller. *TestML - A Test Exchange Language for Model-Based Testing of Embedded Software*, pages 98–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[44] A.-R. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber. A Meta-Model for Tests of Avionics Embedded Systems. *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 5–13, 2013.

[45] M. Hardin and J. Underwood. Mars climate orbiter team finds likely cause of loss. Sept. 30, 1999, https://mars.nasa.gov/msp98/news/mco990930.html.

[46] P. Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 134–142, Jun 1998.

[47] Itemis. XText. http://www.eclipse.org/Xtext/.

[48] JBehave. JBehave. http://jbehave.org/index.html.

[49] Jetbrains. Meta-Programming System. https://www.jetbrains.com/mps/.

[50] JetBrains MPS blog. Plugin spotlight : Generate Python or Ruby with MPS-plaintextgen. https://blog.jetbrains.com/mps/2017/06/plugin-spotlight-generate-python-or-ruby-with-mps-plaintextgen/.

[51] R. John. Partitioning in Avionics Architectures : Requirements, Mechanisms, and Assurance. Technical report, 1999.

[52] M. J. Karlesky, W. I. Bereza, and C. B. Erickson. Effective test driven development for embedded software. *2006 IEEE International Conference on Electro Information Technology*, (616) :382–387, 2006.

[53] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3) :331–380, July 2005.

[54] J. C. Knight. Safety critical systems : Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[55] Lisp community. Common lisp. http://lisp-lang.org/.

[56] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages : Collected experiences. In *4 th Workshop on Domain-Specific Modeling*, 2004.

[57] P. Manhart and K. Schneider. Breaking the Ice for Agile Development of Embedded Software : An Industry Experience Report. In *Proceedings of the 26th International Conference on Software Engineering*, pages 378–386. IEEE, 2004.

[58] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics : System Demonstrations*, pages 55–60, Baltimore, Maryland, 2014. Association for Computational Linguistics.

[59] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Trans. Software Eng.*, 10(5) :564–574, 1984.

[60] D. North. Introducing BDD. https://dannorth.net/introducing-bdd/.

[61] D. North. What's in a story. https://dannorth.net/whats-in-a-story/.

[62] A. Ott. *System Testing in the Avionics Domain*. PhD thesis, University of Bremen, 2007.

[63] S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts : An adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. ACM.

[64] Radio Technical Commission for Aeronautics. Rtca. `https://www.rtca.org/`.

[65] RSpec. Rspec. `http://rspec.info/`.

[66] RTCA. Software considerations in airborne systems and equipment certification, December 1992.

[67] SAE Aerospace Recommended Practices. *SAE ARP 4754 : Certification Considerations for Highly-Integrated or Complex Aircraft Systems.* SAE International, November 1996.

[68] O. Salo and P. Abrahamsson. Agile methods in European embedded software development organisations : a survey on the actual use and usefulness of Extreme Programming and Scrum. *Software, IET*, 2(1) :58–64, February 2008.

[69] I. Schieferdecker and E. Bringmann. Continuous ttcn-3 : Testing of embedded control systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, SEAS '06, pages 29–36, New York, NY, USA, 2006. ACM.

[70] R. S. Scowen. Extended BNF-a generic base standard. Technical report, Technical report, ISO/IEC 14977. http ://www. cl. cam. ac. uk/mgk25/iso-14977. pdf, 1998.

[71] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 451–464, New York, NY, USA, 2006. ACM.

[72] M. R. Smith, A. K. C. Kwan, A. Martin, and J. Miller. E-TDD - embedded test driven development a tool for hardware-software co-design projects. In *6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, pages 145–153, 2005.

[73] S. Soderland. Learning information extraction rules for semi-structured and free text. *Mach. Learn.*, 34(1-3) :233–272, Feb. 1999.

[74] M. Soeken, R. Wille, and R. Drechsler. Assisted Behavior Driven Development using Natural Language Processing. In C. A. Furia and S. Nanz, editors, *Objects, Models, Components, Patterns*, pages 269–287, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[75] C. Solís and X. Wang. A study of the characteristics of behaviour driven development. *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, 2011.

[76] I. Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[77] T. Stålhane, T. Myklebust, and G. Hanssen. The application of safe scrum to IEC 61508 certifiable software. In *11th International Probabilistic Safety Assessment and Management Conference 2012, PSAM11 ESREL 2012*, volume 8, pages 6052–6061. Curran Associates, Inc., 2012.

[78] A. G. Stephenson and L. S. LaPiana. Mars Climate Orbiter Mishap Investigation Board. November 10, 1999, `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`.

[79] S. tech entreprises. ATA100 and section headings. `http://www.s-techent.com/ATA100.htm`.

[80] Terence Parr. ANother Tool for Language Recognition (ANTLR). `http://www.antlr.org/`.

[81] J.-P. Tolvanen and S. Kelly. Metaedit+ : Defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 819–820, New York, NY, USA, 2009. ACM.

[82] A. Ulrich, S. Jell, A. Votintseva, and A. Kull. The ETSI Test Description Language TDL and its Application. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 601–608, 2014.

[83] A. Van Deursen, P. Klint, J. Visser, et al. Domain-specific languages : An annotated bibliography. *Sigplan Notices*, 35(6) :26–36, 2000.

[84] A. Van Lamsweerde. Goal-oriented requirements engineering : A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[85] E. Visser. Spoofax. `http://strategoxt.org/Spoofax`.

[86] M. Voelter. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011.

[87] M. Voelter and S. Lisson. Supporting diverse notations in mps'projectional editor. In *GEMOC@ MoDELS*, pages 7–16, 2014.

[88] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. *Towards User-Friendly Projectional Editors*, pages 41–61. Springer International Publishing, Cham, 2014.

[89] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[90] E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *ACM SIGPLAN Notices*, volume 50, pages 29–38. ACM, 2014.

[91] M. P. Ward. Language oriented programming. *Software—Concepts and Tools*, 15 :147–161, 1995.

[92] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development : current practice. *IEEE Software*, 15(2) :34–45, Mar 1998.