Une approche dirigée par les modèles pour le développement de tests pour systèmes avioniques embarqués

Alexandru Robert Ciprian Guduvan

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

Présentée et soutenue par :
Alexandru-Robert-Ciprian GUDUVAN

le jeudi 18 avril 2013

Titre :

Une approche dirigée par les modèles pour le développement de tests pour systèmes avioniques embarqués
(Synthèse)
A Model-Driven Development of Tests for Avionics Embedded Systems
(Summary)

École doctorale et discipline ou spécialité :
ED MITT : Sureté de logiciel et calcul de haute performance

Unité de recherche :
Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS)

Directeur(s) de Thèse :

Mme Hélène WAESELYNCK (Directrice de thèse)
Mme Virginie WIELS (Co-Directrice de thèse)

Jury :

M. Fabrice BOUQUET (Rapporteur)
M. Benoît COMBEMALE (Examinateur)
M. Yann FUSERO (Examinateur)
M. Yves LE TRAON (Rapporteur)
Mme Hélène WAESELYNCK (Directrice de thèse)
Mme Virginie WIELS (Co-Directrice de thèse)

Rapporteur: Fabrice BOUQUET

Rapporteur: Yves LE TRAON

Date de la soutenance: le 18 avril, 2013

Abstract

Le développement de tests pour les systèmes d'avioniques met en jeu une multiplicité de langages de test propriétaires, sans aucun standard émergent. Les fournisseurs de solutions de test doivent tenir compte des habitudes des différents clients, tandis que les échanges de tests entre les avionneurs et leurs équipementiers / systémiers sont entravés. Nous proposons une approche dirigée par les modèles pour s'attaquer à ces problèmes: des modèles de test sont développés et maintenus à la place du code, avec des transformations modèle-vers-code vers des langages de test cibles. Cette thèse présente trois contributions dans ce sens. La première consiste en l'analyse de quatre langages de test propriétaires actuellement déployés. Elle nous a permis d'identifier les concepts spécifiques au domaine, les meilleures pratiques, ainsi que les pièges à éviter. La deuxième contribution est la définition d'un méta-modèle en EMF Ecore, qui intègre tous les concepts identifiés et leurs relations. Le méta-modèle est la base pour construire des éditeurs de modèles de test et des motifs de génération de code. Notre troisième contribution est un démonstrateur de la façon dont ces technologies sont utilisées pour l'élaboration des tests. Il comprend des éditeurs personnalisables graphiques et textuels pour des modèles de test, ainsi que des transformations basées-motifs vers un langage du test exécutable sur une plateforme de test réelle.

Table des Matières

Liste des Figures

LISTE DES FIGURES

Liste des Tableaux

Glossaire

ADIRS	Air Data Inertial Reference System
AFDX	Avionics Full-Duplex Switched Ethernet
ARINC	Aeronautical Radio, Incorporated
ASN.1	Abstract Syntax Notation One
ATL	Automatic Test Language (internal to Cassidian Test & Services)
ATLAS	Automatic Test Language for All Systems
ATML	Automatic Test Mark-up Language
BDD	Binary Decision Diagram
BNF	Backus-Naur Form
CAN	Controller Area Network
CBCTC	CycleByCycleTestComponent
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standards Institute
FSM	Finite State Machines
FWS	Flight Warning System
GMF	Graphical Modeling Framework
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HiL	Hardware-in-the-Loop
HSPL	High-Level Scripting Programming Language
ICD	Interface Control Document
IDE	Integrated Development Environment
IMA	Integrated Modular Avionics
IP	Internet Protocol
ITU	International Telecommunication Union
LCS	Launch Control System
LTS	Labelled Transition Systems
M2M	Model to Model
M2T	Model to Text
MAC	Media Access Control
MaTeLo	Markov Test Logic
MATLAB	Matrix Laboratory
MC/DC	Modified Condition/Decision Coverage
MCUM	Markov Chain Usage Model
MDE	Model-Driven Engineering
MiL	Model-in-the-Loop
MMI	Man Machine Interface
MOF	Meta Object Facility
MTC	Main Test Component
MTL	Model to Text Language
NASA	National Aeronautics and Space Administration
OCL	Object Constraint Language
OMG	Object Management Group
OOPL	Object-Oriented Programming Language
OSI	Open Systems Interconnection
PL	Proprietary Language
PLTL	Propositional Linear Temporal Logic
PTC	PeriodicTestComponent
SCARLETT	SCAlable & ReconfigurabLe Electronics plaTforms and Tools
SDL	Specification and Description Language
SiL	Software-in-the-Loop
SMV	Symbolic Model Verifier
STC	SequentialTestComponent

GLOSSAIRE

$STELAE$	Systems TEst LAnguage Environment
SUT	System under Test
TM	TestMonitor
$TTCN-3$	Testing and Test Control Notation Version 3
UC	User Code
UIO	Unique Input Output
UML	Unified Modeling Language
$UMTS$	Universal Mobile Telecommunications System
UTP	UML Testing Profile
$VDM-SL$	Vienna Development Method Specification Language
VL	Virtual Link
$WCET$	Worst-Case Execution Time
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1

Introduction

Cette thèse de doctorat est une collaboration entre trois entités:

- un partenaire industriel: Cassidian Test & Services (une société EADS) de Colomiers, France,
- et deux partenaires académiques - laboratoires de recherche de Toulouse, France:
  - le Laboratoire d'Analyse et d'Architecture de Systèmes (LAAS-CNRS),
  - l'Office National d'Études et de Recherches Aérospatiales (ONERA).

Le test est l'un des moyens principaux de vérification et de validation, qui cible l'élimination des fautes [1]. Dans le domaine de l'avionique, il est une étape cruciale pour la certification des systèmes qui sont installés à l'intérieur d'un avion [2]. L'automatisation de l'exécution des tests est un sujet de recherche de longue date, qui est aussi important aujourd'hui que jamais en raison des avantages importants qu'il offre par rapport aux tests manuels.

Le contexte industriel de ce travail est le test d'intégration des systèmes avioniques embarqués. Nous avons été motivés par le constat que les solutions existantes pour l'automatisation des tests dans notre contexte industriel ne répondent plus aux besoins des parties concernées. Le monde des langages de test, dans lesquels les cas de test sont développés, comprend de nombreux langages propriétaires et est hétérogène: chaque langage de test propose des fonctionnalités différentes et les fonctionnalités communes sont offertes de différentes manières. Comme aucune norme ne se dessine dans ce domaine et que les langages de test existants sont différents, l'échange d'artefacts de test entre les différents acteurs est entravé. Cette situation est difficile aussi pour les fournisseurs de solutions de test, qui doivent s'adapter aux besoins individuels de chaque client. En outre, les ingénieurs de test ne profitent pas des dernières méthodologies et technologies avancées issues du génie logiciel.

Afin de répondre à tous ces besoins, nous proposons le transfert des méthodologies et technologies de l'ingénierie dirigée par les modèles, issues du génie logiciel, dans le domaine de la mise en œuvre de tests pour systèmes avioniques embarqués. Ainsi, des modèles de test de haut-niveau, indépendants des plate-formes de test, remplaceraient la pratique actuelle. Pour leur exécution, les modèles de test seront traduits en langages de test exécutables existants. Cette proposition est motivée par le fait que les cas de test sont des logiciels et par conséquent peuvent bénéficier des dernières avancées en matière de développement logiciel.

Le Chapitre 2 présente l'état de l'art du domaine du test. Nous discutons les travaux existants en nous concentrant sur deux questions principales: la conception des tests (sélection et oracle de test) et la mise en œuvre des tests (méthodologies et formalismes pour le développement des tests).

Le Chapitre 3 aborde les spécificités de notre contexte industriel.

Notre première contribution (Chapitre 4) consiste en l'analyse d'un échantillon de langages de test actuellement utilisés dans l'industrie. Nous avons ciblé un échantillon formé de quatre langages de test propriétaires utilisés de façon opérationelle dans l'avionique, et de deux langages de test utilisés respectivement dans le domaine des télécommunications et systèmes distribués et celui de l'automobile. Cette analyse nous a permis d'identifier l'ensemble des fonctionnalités/concepts spécifiques au domaine, les meilleures pratiques, ainsi que les pièges à éviter.

Notre deuxième contribution (Chapitre 5) consiste en la définition d'un méta-modèle de test qui intègre un ensemble riche de concepts spécifiques au domaine, tout en tenant compte des meilleures pratiques.

La troisième contribution (Chapitre 6) consiste en la mise en œuvre d'un premier prototype implémenté sur une plate-forme de test d'intégration réelle, disponible dans le commerce: U-TEST Real-Time System [3] développée chez Cassidian Test & Services. Nous avons utilisé une transformation modèle-vers-texte basée-motifs, ciblant un langage de test exécutable propriétaire.

Le prototype permet d'illustrer le développement de modèles de test, leur implémentation et leur exécution, sur deux cas d'utilisation d'une complexité simple et moyenne (Chapitre 7).

Le Chapitre 8, comprenant les conclusions et les perspectives de nos travaux, conclut ce document.

2

État de l'art - Le test

Ce chapitre présente l'état de l'art sur le thème du **test**.

Le test est une méthode de vérification dynamique visant à l'élimination des fautes [1]. Il consiste en la stimulation du système sous test avec des entrées valuées et en l'analyse de ses résultats par rapport au comportement attendu.

Dans le domaine du test, deux défis principaux sont habituellement traités : **la conception du test** et **l'implémentation du test**. La conception du test porte sur la définition de la logique du test, tandis que l'implémentation concerne le développement du code de test pour automatiser l'exécution de cette logique sur des plates-formes données.

La conception du test peut être divisée en deux questions: **la sélection de test** (Section 2.1) et **l'oracle de test** (Section 2.2). La sélection du test concerne le choix du comportement à tester et des données d'entrée associées. L'oracle de test concerne la manière dont un verdict peut être émis sur la conformité du comportement observé par rapport à celui attendu attendu.

L'implémentation du test met en jeu des **méthodologies de développement de tests** (Section 2.3) et des **formalismes de développement de tests** (Section 2.4). Les méthodologies de développement de tests donnent les lignes directrices que les ingénieurs de test doivent suivre pour implémenter les tests. Les formalismes de développement de tests permettent la définition de tests qui sont automatiquement exécutables.

La plupart des travaux universitaires existants se concentrent sur la conception du test, la sélection du test étant un sujet de recherche plus souvent abordé que l'oracle de test. Bien que les publications sur l'implémentation du test soient moindres, ce sujet présente néanmoins des défis importants. Automatiser l'exécution des tests est un enjeu majeur dans un contexte industriel.

2.1 Sélection de test

À l'exception de cas triviaux, le test exhaustif de tous les comportements possibles que peut présenter un système n'est pas réalisable. Par conséquent, un petit nombre de cas de test qui couvrent un sous-ensemble du domaine d'entrée doit être identifié. La sélection peut être guidée par des critères liés à un modèle de la structure du système ou des fonctions qu'il offre. Ces deux catégories donnent lieu au **test structurel** [24, 25] et **fonctionnel**.

Le test structurel définit des éléments couvrir dans un modèle de type graphe de contrôle [24], éventuellement enrichi par des informations sur le flot des données [25].

Pour le test fonctionnel, il n'existe pas de modèle standard (tel que le graphe de contrôle). Par conséquent, le test fonctionnel couvre un large éventail de méthodes et de techniques, qui dépendent du formalisme utilisé pour modéliser le comportement du système. Des approches bien connues incluent :

- **des décompositions du domaine d'entrée en classes de valeurs** [28, 29],
- **des modélisations à états** [30, 31, 32, 34, 35, 43],
- **des spécifications algébriques** [49, 50],
- **des tables de décisions**,
- **des profil d'utilisation probabilistes** [54, 55].

Pour une présentation détaillée de la sélection de test, y compris des informations sur son industrialisation, le lecteur peut se référer aux ouvrages existants [20, 21, 22, 23].

2.2 Oracle de test

L'analyse automatique des résultats de test est souhaitable et devient nécessaire quand un grand nombre d'entrées du système sous test ont été sélectionnées.

Les solutions les plus satisfaisantes sont basées sur l'existence d'une spécification formelle du logiciel cible, utilisable pour déterminer les résultats attendus.

Le test dos-à-dos représente une autre solution pour le problème de l'oracle de test, en particulier pour les systèmes qui ont de la diversification fonctionnelle à des fins de tolérance aux fautes. D'autres solutions d'oracles de test partiels peuvent être employées, en utilisant des contrôles de vraisemblance sur les résultats des tests. Des contrôles plus sophistiqués utilisent des invariants qui peuvent également prendre en compte des aspects temporels [64, 65]. Des assertions/contrats exécutables peuvent également être inclus dans le code logiciel [66, 67, 68].

2.3 Méthodologies de développement de tests

Les méthodologies de développement de test donnent des lignes directrices que les ingénieurs de test utilisent quand ils développent des tests. L'objectif est de délivrer du code de test de haute qualité (par exemple, annoté avec des commentaires, structuré, compréhensible, documenté et réutilisable). L'ouvrage [69] offre une vue d'ensemble des pratiques existantes, ainsi que des études sur des cas réels.

Cet ouvrage discute notamment des techniques d'écriture de scripts de test. Cinq techniques de script sont présentées, chacune avec ses forces et ses faiblesses: linéaire, structurée, partagée, guidée par les données de script et guidée par des mots clés.

La comparaison de la sortie du système sous test avec la sortie attendue (problème de l'oracle) est également discutée. Enfin, [69] se penche sur les questions concernant les outils de gestion de tests, l'organisation et la conduite des tests, ainsi que sur les paramètres permettant la mesure de la qualité des tests.

Ces méthodologies sont la plupart du temps informelles, se trouvant dans les documents textuels que les ingénieurs de test sont censés lire et appliquer.

2.4 Formalismes de développement de tests et outils associés

Les formalismes de développement de tests et les outils associés sont des technologies qui permettent la production de cas de test qui sont automatiquement exécutables sur une plate-forme de test donnée. Trois grands types de solutions peuvent être identifiés: les **outils de capture & rejeu**, les **frameworks de test**, les **langages de test** et la **modélisation des tests**. Des exemples illustrant chaque type de solution sont discutés plus en détail dans la Section 2.4 du tome en anglais de ce mémoire.

Modélisation de tests La modélisation de tests se base sur l'hypothèse que les cas de test sont des logiciels, et en tant que tels peuvent bénéficier des méthodes et technologies avancées issues du génie logiciel, comme par exemple l'ingénierie dirigée par les modèles (MDE) [92]. L'utilisation des approches MDE dans le monde du test est un domaine de recherche actif, bien que les publications existantes ne soient pas nombreuses. Nous ne parlons pas ici des techniques de sélection de test basées-modèles telles que celles déjà discutées dans la Section 2.1. L'accent est ici sur l'implémentation des tests.

La plupart des travaux existants sur l'implémentation de tests utilisent le *Unified Modeling Language* (UML) [93] pour les modèles de test. Un profil UML pour le test - *UML Testing Profile* (UTP) [94] - a été standardisé par le *Object Management Group* (OMG) [129]. Plusieurs projets ont porté sur l'intégration de UTP et du langage de

test *Testing and Test Control Notation Version 3* (TTCN-3) [83], tels que [95]. Un méta-modèle pour TTCN-3 peut être trouvé dans [96], plus tard encapsulé dans la plate-forme TTworkbench [97]. Un projet similaire chez Motorola [98] utilise la suite d'outils TAU [99].

Certains auteurs ont proposé leurs propres profils UML. Un profil UML et des transformations de modèles pour le test d'applications Web sont discutés dans [100]. Dans l'avionique, la modélisation UML de logiciels de simulation pour le test modèle-en-boucle est proposée dans [101]. Toujours dans l'avionique, [102] propose des modèles de test conformes à un méta-modèle de test (intégrant des formalismes basées-automates), pour la deuxième génération de Integrated Modular Avionics (IMA) [103]. Des travaux antérieurs par les mêmes auteurs incluent la modélisation de tests basée sur des automates pour leur plate-forme RT-Tester [104, 105].

Les méta-modèles de langages de l'ITU-T, tels que TTCN-3 ou *Specificaton and Description Language* (SDL), sont étudiés dans [106]. L'objectif est de s'abstraire des grammaire concrètes *Backus–Naur Form* (BNF) et utiliser un ensemble de concepts fondamentaux partagés par une famille de langages. Des concepts identifiés par des experts sont utilisés comme support pour la construction de méta-modèles pour chaque langage.

D'autres artefacts utilisés pendant les tests, en plus des cas de tests eux-mêmes, peuvent bénéficier de l'ingénierie dirigée par les modèles. Par exemple, la modélisation de l'environnement du système sous test est discutée dans [107].

Dans le même esprit que les différents travaux ci-dessus, nous nous sommes concentrés sur l'utilisation de l'ingénierie dirigée par les modèles pour le développement de tests. Le partenaire industriel de ce projet, Cassidian Test & Services, est un fournisseur d'outils d'automatisation de l'exécution des tests et de plates-formes de test pour des composants et systèmes avioniques embarqués.

2.5 Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art sur le test. Nous avons discuté les défis majeurs dans ce domaine (la conception et l'implémentation des tests), ainsi que les solutions qui ont été ou qui sont actuellement étudiées.

La conception du test peut être divisée en deux questions: la sélection du test et l'oracle de tests. La sélection du test est largement discutée dans la littérature, contrairement à l'oracle de test. Les solutions au problème de la sélection de tests pertinents comprennent des approches telles que le test fonctionnel et structurel.

L'implémentation du test peut être scindée dans les deux problématiques suivantes : les méthodologies de développement de tests et les formalismes/outils. Les

méthodologies de développement des tests guident les ingénieurs de test dans leur activité de codage, tandis que les formalismes de développement de tests et les outils leur permettent d'automatiser le codage (au moins partiellement). Dans le milieu universitaire, ces questions ont reçu beaucoup moins d'intérêt que les questions de sélection du test. Néanmoins, elles jouent un rôle important dans les environnements industriels, où l'automatisation des tests peut fournir de nombreux avantages que les techniques de test manuelles ne pourront jamais atteindre.

Notre travail se positionne dans le domaine des formalismes de développement de tests. Nous nous sommes concentrés sur l'introduction des méthodologies et technologies modernes de l'ingénierie dirigée par les modèles dans le domaine du test des systèmes avioniques embarqués. Le Chapitre 3 présente les spécificités de notre contexte industriel, qui justifient et motivent la contribution proposée par cette thèse de doctorat.

3

Contexte industriel

Nous discutons brièvement notre contexte industriel dans ce chapitre : le test d'intégration en-boucle de systèmes avioniques embarqués. L'analyse des caractéristiques de notre contexte industriel montre qu'une nouvelle approche dirigée par les modèles serait utile, justifiant ainsi la pertinence du sujet que nous avons choisi de traiter par ces travaux.

3.1 Système avionique embarqué

Dans cette section, nous discutons les spécificités des systèmes avioniques embarqués : leur cycle de vie (Sous-section 3.1.1), leurs interfaces (Sous-section 3.1.2) et leur comportement (Sous-section 3.1.3).

3.1.1 Cycle de vie et activités de test

La vérification et la validation d'un système avionique embarqué comprend un processus rigoureux, avec des activités de test spécifiques attachées aux différentes phases du cycle de vie [108]. Le présent document se concentre sur les phases de test en boucle qui se produisent pendant le processus de développement. Un système avionique est étroitement couplé à son environnement. Le test en boucle résout ce problème en ayant un modèle de l'environnement pour produire les données.

Le test en boucle se présente sous diverses formes :

- modèle-en-boucle (en anglais, *model-in-the-loop* ou MiL),
- logiciel-en-boucle (en anglais, *software-in-the-loop* ou SiL),
- matériel-en-boucle (en anglais, *hardware-in-the-loop* ou HiL).

3.1.2 Document de contrôle d'interface (ICD)

Dans le domaine de l'avionique, les interfaces d'un système sont présentées dans un *Interface Control Document* (ICD). Quelque soit le format spécifique, le document contient des informations sur les interfaces, à plusieurs niveaux hiérarchiques (Figure 3.1).

Interface Control Document (ICD)
NOM du SYSTEME: SUT_1

#	NOM du CONNECTEUR	PIN du CONNECTEUR	NOM du BUS	TYPE de CONNECTION
CONNECTEUR	CONNECTOR_1	PIN_1	ARINC_429_1	ARINC_429
...				

#	NOM du BUS	DESCRIPTION du BUS	CONFIGURATION du BUS	NOM du CONNECTEUR / PIN
SORTIE - BUS ARINC 429	ARINC_429_1	...	...	CONNECTOR_1 / PIN_1
...				

#	NOM du LABEL	CONFIGURATION du LABEL	NOM du BUS	NOM du PARAMETRE APPLICATIF
SORTIE - LABEL ARINC 429	LABEL_1	...	ARINC_429_1	AC_SPEED
...				

Figure 3.1: Exemple de Interface Control Document (ICD) -

Généralement un tuple est utilisé pour identifier de manière unique une instance particulière d'un élément de l'ICD. Un tuple peut représenter un chemin dans la structure arborescente de l'ICD :

$$\ll \texttt{SUT\_1/ARINC\_429\_1/LABEL\_1/AC\_SPEED} \gg.$$

Dans les procédures de test, il est souvent pratique de disposer de noms symboliques plus courts.

3.1.3 Comportement réactif

Les composants d'un système avionique présentent généralement un comportement cyclique, où un cycle d'exécution lit les données d'entrée et calcule les sorties. Les composants consommateurs s'attendent à recevoir leurs données d'entrée dans les fenêtres temporelles imposées par la communication cyclique.

Globalement, le test en boucle correspond à une approche complètement différente de celle utilisée pour tester les systèmes distribués asynchrones, où l'envoi de quelques messages déclenche l'activité fonctionnelle d'un système autrement au repos.

3.2 La plateforme de test

Pour un système avionique embarqué, une plate-forme de test a normalement les éléments suivants :

- un contrôleur de test,
- des ressources de test,
- un réseau de test,
- un langage du test.

Le traitement de la logique de test est centralisé: le contrôleur de test exécute les tests écrits dans un langage pris en charge par la plate-forme. Au cours de l'exécution, des commandes sont envoyées aux ressources de test permettant d'effectuer les interactions réelles avec le système sous test. Le réseau de test comporte deux parties, l'une reliant le contrôleur de test aux ressources de test (réseau de commande de test) et l'autre reliant les ressources de test au système sous test (réseau d'interaction de test).

3.3 Intervenants et évolution des besoins

Le test d'intégration des systèmes avioniques inclut les types suivants d'intervenants :

- des fournisseurs de solutions de test,
- des utilisateurs de solutions de test :
  - avionneurs,
  - équipementiers,
  - systémiers.

Nous avons eu l'occasion de discuter et recueillir les besoins exprimés par des représentants de tous ces types d'intervenants.

Historiquement, l'avionneur était en charge de toute l'activité d'intégration des systèmes embarqués avioniques. De nos jours, il y a un transfert de l'activité de l'avionneur vers les fournisseurs d'équipements et ceux-ci sont invités à participer à la première phase d'intégration. Les équipementiers deviennent des systémiers, avec de nouveaux besoins en termes d'échanges de tests avec l'avionneur. Dans la pratique,

la portabilité des tests d'un environnement à un autre ne peut pas être facilement atteinte, ce qui freine ces échanges. Au fait de la multitude d'outils d'exécution de test propriétaires, les fournisseurs de solutions de test ont de leur côté à tenir compte des différentes habitudes de leurs clients. Des capacités de personnalisation sont donc exigées pour répondre aux demandes des utilisateurs.

Aucun langage de test standard n'existe ou émerge pour le test en-boucle de systèmes avioniques embarqués. Ceci contraste avec d'autres domaines qui utilisent des normes internationales, par exemple: *Abbreviated Test Language for All Systems* (ATLAS) [112] et *Automatic Test Mark-up Language* (ATML) [113] pour le test du matériel (phases de production et de maintenance du cycle de vie) ou TTCN-3 [82] dans le domaine du test des protocoles de télécommunication et des systèmes distribués. Ces solutions standardisées ne sont pas conçues pour répondre aux spécificités de notre contexte industriel et ne sont pas directement réutilisables en tant que telles.

D'autres besoins portent sur la lisibilité et la facilité d'écriture des tests par les ingénieurs.

Toutes ces exigences ont motivé notre orientation vers des descriptions de tests de haut-niveau, indépendantes des plate-formes de test, avec des fonctionnalités de génération automatique de code. Ce travail se concentre sur l'introduction d'une approche dirigée par les modèles pour le développement de tests pour le test en boucle de systèmes avioniques embarqués.

3.4 Conclusion

Dans ce chapitre, nous avons présenté le contexte industriel de la thèse de doctorat : les spécificités des systèmes avioniques embarqués (cycle de vie, interfaces et comportement), les plates-formes de test associées, avec l'évolution des besoins exprimés par les intervenants dans ce domaine : les avionneurs, les fournisseurs d'équipements/systèmes et les fournisseurs de solutions de test.

Ce contexte industriel est complexe :

- le système sous test (SUT) est vérifié à différents niveaux de maturité (modèle, logiciel, logiciel intégré sur du matériel),
- le SUT a un large éventail de types d'interfaces à plusieurs niveaux hiérarchiques, organisées dans un document de type ICD,
- un grand nombre de parties interagissent (avionneurs, fournisseurs d'équipements et de systèmes, fournisseurs de solutions de test), chacun ayant ses propres habitudes et outils.

Les langages de test existants ne répondent plus à la complexité du contexte industriel et aux besoins exprimés par les intervenants. En outre, les concepteurs de tests n'ont pas accès aux dernières avancées en génie logiciel, comme l'ingénierie dirigée par les modèles. Cette thèse propose de transférer ces technologies au développement de tests pour systèmes avioniques. Les modèles de test deviendraient ainsi le cœur de l'activité de développement de tests, remplaçant les approches actuelles où le code de test occupe cette position. La fondation de l'ingénierie dirigée par les modèles est le méta-modèle, qui contraint la définition de modèles de la même manière que les grammaires des langages de programmation contraignent la définition de code. Afin d'être en mesure de définir notre propre méta-modèle, spécifique au test en boucle des systèmes avioniques embarqués, nous avons analysé un échantillon de langages de test dans le Chapitre 4.

4

Analyse des langages de test

Nous présentons et analysons ici un échantillon de six langages de test. Nous avons choisi quatre langages de test propriétaires utilisés dans notre contexte industriel, ainsi que deux langages de test supplémentaires issus d'autres domaines. Cette analyse nous a permis d'identifier l'ensemble des concepts spécifiques au domaine, qui ont par la suite été intégrées dans le méta-modèle de test sous-jacent à notre approche. En outre, cette analyse nous a également permis d'identifier les meilleures pratiques, ainsi que les pièges à éviter.

La Section 4.1 présente notre échantillon de six langages de test et décrit notre méthode d'analyse. La Section 4.2 discute les fonctionnalités génériques présentées par les langages de test. Les Sections 4.3 à 4.6 discutent les fonctionnalités liées au test. Nous en avons identifié quatre grandes catégories :

- l'organisation des tests (Section 4.3),
- l'abstraction et l'accès aux interfaces du SUT (Section 4.4),
- les instructions du langage pour piloter les tests et interagir avec le SUT (Section 4.5),
- la gestion du temps (Section 4.6).

La Section 4.7 présente une liste de principes qui ont guidé notre travail ultérieur de méta-modélisation, basée sur les résultats de l'analyse des langages de test. La Section 4.8 conclut ce chapitre.

4.1 Échantillon de langages de test

L'échantillon de langages de test comprend :

- quatre langages de test propritaires du domaine avionique, qui seront nommés PL_1, PL_2, PL_3 et PL_4,
- TestML [114] du domaine de l'automobile,
- *Testing and Test Control Notation Version 3* (TTCN-3) [82, 83] du domaine des réseaux et télécommunications.

Les quatre langages de test de propriétaires, de PL_1 à PL_4, sont des langages utilisés de façon opérationnelle dans l'industrie aéronautique. Le premier représente l'offre de Cassidian Test & Services sur la plate-forme de test d'intégration U-TEST Real-Time System [3]. L'identité des trois autres ne peut pas être divulguée.

TestML est issu d'un projet de recherche dans le domaine de l'automobile. Son objectif était de concevoir un langage d'échange, dans le sens indiqué par la Figure 4.1. TestML représente une tentative pour synthétiser des préoccupations découlant de la pratique des tests en boucle, son examen a donc été jugé pertinent pour nous.

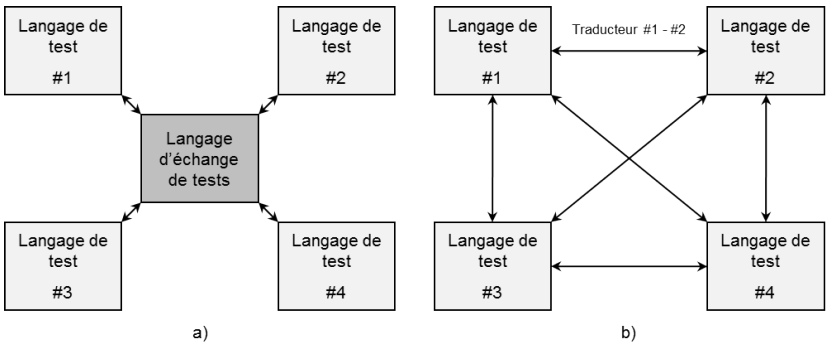

Figure 4.1: Langage d'échange de tests pour limiter le nombre de traducteurs
-

TTCN-3 est utilisé dans le test de systèmes distribués et des protocoles de communication. Il est inclus dans notre échantillon, car les systèmes embarqués avioniques sont également des systèmes distribués, et leur communication est basée sur des interfaces conformes à un large éventail de protocoles de communication.

Notre approche a consisté à identifier les fonctionnalités que les langages de test dans notre échantillon offrent, conjointement avec les ressemblances et les différences sur la façon dont elles sont offertes. Dans le cas où une fonctionnalité n'est pas offerte par un langage de test ou est offerte de différentes manières, nous avons contacté les ingénieurs de test pour des clarifications.

Dans le tome en anglais de ce mémoire, les caractéristiques que nous avons analysées sont synthétisées dans des vues schématiques puis discutées. Nous nous bornons ici à reprendre la liste des idées clefs qui conclut chaque discussion. Nous allons d'abord

donner les caractéristiques génériques de l'échantillon de langages de test (Section 4.2), et par la suite continuer avec les fonctions liées au test (Sections 4.3 à 4.6).

4.2 Caractéristiques génériques

Idées clefs

- Les langages de test spécifiques offrent un vocabulaire personnalisable, concis et de haut-niveau, spécifique au domaine.
- Les langages de test basés sur des langages génériques de programmation permettent la réutilisation des fonctionnalités et des outils existants.
- Les langages de test interprétés ne nécessitent pas une nouvelle étape de compilation chaque fois que le code est modifié.
- La pratique actuelle, parmi les partenaires de Cassidian Test & Services, favorise un paradigme de programmation impératif.
- Une solution de test indépendante de la plateforme de test est souhaitable, avec une séparation claire entre le noyau du langage de test et les adaptateurs de test spécifiques à la plate-forme.

4.3 L'organisation des tests

L'organisation intra-test se réfère à la description structurée d'un test. Nous nous concentrons ici sur les types d'organisation intra-test, parce que l'organisation inter-test est généralement gérée par des outils externes.

Nous discutons des formes d'organisation intra-test telles que: l'organisation sémantique des instructions (Sous-section 4.3.1) et l'organisation des flots de contrôle parallèles (Sous-section 4.3.2).

4.3.1 Organisation sémantique des instructions

Idées clefs

- Les sections de test sont utiles pour l'identification d'un test - en-tête de test, et pour l'organisation du comportement d'un test en parties distinctes.
- Des sections de test peuvent être définies suivant une méthodologie basée sur des annotations ou en les intégrant à l'intérieur du langage de test.

4.3.2 L'organisation des flots de contrôle parallèles

Idées clefs

- Deux types de flots de contrôle ont été identifiés: explicite (les ingénieurs de test définissent le comportement parallèle et contrôlent son exécution, en employant des composants de test) et implicite (les ingénieurs de test utilisent des instructions prédéfinies, comme une stimulation de rampe, qui s'exécutent en parallèle).
- Des constructions de haut-niveau, telles que les composants de test, masquent les détails d'implémentation multi-tâche.
- La communication entre composants de test peut utiliser des événements ou des données partagées.
- Les interfaces formelles permettent la réutilisation et l'instanciation multiple de composants de test.
- Différents types de composants de test sont utilisés, tels que: des composants de test périodiques (pour l'élaboration de modèles d'environnement du SUT et de stimulations complexes) et des moniteurs de test (comportement condition $\rightarrow$ action, pour une visibilité améliorée de la logique de test).
- Les concepts comportementaux et structurels peuvent être séparés, avec de la génération automatique de code pour les derniers.
- L'architecture des liens entre les composants de test est statique dans notre contexte industriel.

4.4 Lien avec les interfaces du système sous test

4.4.1 Niveaux hiérarchiques ciblés et abstraction des interfaces du SUT

Idées clefs

- Les interfaces du SUT sont toujours abstraites par l'intermédiaire d'identifiants dérivés de l'ICD.
- L'architecture des liens entre les interfaces des composants de test et les interfaces du SUT est statique dans notre contexte industriel.

4.4.2 Identifiants et accès aux interfaces du SUT

Idées clefs

- Les mécanismes d'accès aux interfaces du SUT sont hétérogènes, d'un niveau hiérarchique à l'autre, mais ainsi au sein d'un même niveau hiérarchique de l'ICD.
- La structure de l'ICD n'est que partiellement reflétée par les langages de test.
- Passer l'identifiant d'un élément ICD en paramètre comme une chaîne de caractères ne permet pas la vérification statique des types.
- L'accès global aux éléments de l'ICD est pratique.

4.5 Instructions des langages de test

Les principales catégories des instructions liées au test sont les suivantes : l'interaction avec le système sous test (Sous-section 4.5.1) et la gestion du verdict (Sous-section 4.5.2).

4.5.1 Interaction avec le système sous test

Idées clefs

- Les langages de test pour le test en boucle offrent de nombreuses instructions d'interaction avec le système sous test, de différents types.
- L'organisation des instructions à l'intérieur des langages de test est hétérogène.
- Une politique d'organisation cohérente serait souhaitable, avec des points d'extension prédéfinis.
- Passer les actions de test comme paramètres textuels à des instructions génériques ne permet pas la vérification statique de type (par exemple, il est possible d'appeler une action de test incompatible avec le type de l'interface du SUT ciblée).
- Avoir des interfaces fortement-typées, auxquelles on attache les actions de test offertes, est un principe intéressant d'organisation.

4.5.2 Gestion du verdict

Idées clefs

- La gestion du verdict de test et sa synthèse automatique est quasi-absente des langages de test propriétaires.

4.6 Gestion du temps

Idées clefs

- Le temps peut être mesuré en unités physiques ou logiques, selon que l'on teste avec le matériel ou le modèle dans la boucle.
- Les instructions de stimulation et observation offertes peuvent être dépendantes du temps.
- La gestion du temps est réalisée à différents niveaux de granularité, allant de blocs d'instructions à des tests complets.
- Des facilités d'activation périodique et d'excution bornée dans le temps sont offertes.
- Un contrôle pas-à-pas (cycle-par-cycle) de l'interaction avec le SUT serait utile, pour des tests plus précis de l'équipement avionique réel.

4.7 Principes guidant la méta-modélisation

L'analyse des langages de test a été utile pour notre travail ultérieur de méta-modélisation. Le Tableau 4.1 présente la liste des principes qui ont guidé notre formalisation des différents concepts spécifiques au domaine (par exemple, le cas de test, le composant de test, le verdict de test) à l'intérieur du méta-modèle de test. Cette liste est tirée des idées clefs données dans les sous-sections précédentes.

Le Chapitre 5 présente le méta-modèle de test que nous avons défini. Il montre la manière dont les principes directeurs mentionnés ci-dessus ont été pris en compte par notre travail de méta-modélisation.

Catégories	Principes
Principes généraux	• Personnalisation/Extension contrôlée par le fournisseur de la solution de test. • Approche de modélisation homogène pour tous les niveaux hiérarchiques de l'ICD. • Eléments fortement-typés en faveur de l'évitement des fautes. • Séparation entre les éléments structurels et comportementaux.
Principes d'organisation des tests	• Concepts pour l'organisation inter-test: cas de test, groupe de tests, suite de tests. • Concepts pour l'organisation intra-test: section de test, composant de test.
Principes de description des interfaces du SUT	• Vue structurée des interfaces du SUT, permettant la navigation à travers les niveaux hiérarchiques. • Points d'extension pour enrichir l'ensemble des types d'éléments ICD disponibles.
Principes de description des architectures de test	• Différents types de composants de test: composants séquentiel, périodique, cycle-par-cycle et moniteurs de test. • Architecture de test statique (sans création dynamique de composants de test). • Accès direct à tous les éléments ICD déclarés, à partir de tout composant de test (pas besoin de déclarer explicitement une connexion). • Accès direct à un ensemble de données auxiliaires (partage) et d'événements (synchronisation): un producteur et potentiellement plusieurs consommateurs. • Accès indirect par l'intermédiaire des interfaces formelles des composants de test, connectables à tout élément de l'ICD ou de l'ensemble de données auxiliaires et d'événements, pour une multiple instanciation et réutilisation.
Principes de description des comportements	• Paradigme de programmation impératif. • Instructions pour le contrôle du flot d'exécution (par exemple, instructions conditionnelles et de répétition). • Gestion automatique du verdict: synthèse du verdict global à partir des verdicts locaux. • Points d'extension pour enrichir l'ensemble d'actions de test disponibles pour les différents types d'éléments ICD. • Actions de test attachées aux éléments ICD selon leur type, appelables par tout composant de test. • Contrôle de l'exécution des composants de test (par exemple, démarrer un composant de test). • Type de comportement spécifiable contraint par le type de composant. • Exécution dépendante du temps pour les composants de test périodique et cycle-par-cycle. • Actions de test ayant une durée (par exemple, une rampe). Paramètre durée explicite avec des bornes supérieures du temps d'exécution, permettant la vérification d'aspects temporels (par exemple, compatibilité avec la périodicité).

Table 4.1: Principes guidant la méta-modélisation

4.8 Conclusion

Dans ce chapitre, nous avons analysé six langages de test. Nous nous sommes concentrés sur quatre langages propriétaires (de PL_1 à PL_4) qui sont actuellement utilisés dans l'industrie aéronautique, pour le test en boucle de systèmes avioniques embarqués, à des niveaux intégration (par exemple, niveau des composants, des systèmes et des multi-systèmes) et de la maturité (c'est à dire, modèle/logiciel/matériel-en-boucle) variés du système sous test. À des fins de comparaison, nous avons également examiné un langage de test issu d'un projet de recherche dans l'industrie de l'automobile (TestML) - couvrant le même type d'activité de test, ainsi qu'une norme internationale mature (TTCN-3) utilisée pour le test de systèmes distribués dans le domaine des réseaux et télécommunications.

Notre analyse a porté sur un certain nombre de fonctionnalités et sur la façon dont elles sont offertes par chaque langage de test. L'analyse a confirmé l'hétérogénéité des langages de test : tous les langages de test n'offrent pas les mêmes fonctionnalités et les fonctionnalités communes sont offertes de différentes manières, et parfois un langage offre une même fonctionnalité sous différentes formes.

L'analyse des langages de test nous a convaincus que les langages de test existants standardisés utilisés dans d'autres domaines ne sont pas facilement portables vers le notre.

Notre analyse nous a aussi convaincus que la multiplicité des solutions propriétaires doit être abordée à un plus haut-niveau, celui des concepts de test. Cela nous a conduit à proposer une approche dirigée par les modèles, où des modèles de test sont développés, maintenus et partagés, et sont ensuite traduits automatiquement en langages de test exécutables cibles (éventuellement propriétaires).

Afin de poursuivre notre approche, nous avons développé un méta-modèle pour les tests en boucle, qui intègre les concepts que nous avons identifiés comme étant d'intérêt. Nous présentons le méta-modèle de test par la suite, dans le Chapitre 5.

5

Méta-modèle de test

Ce chapitre présente le méta-modèle de test. Nous avons présenté le méta-modèle de test dans [118].

Le méta-modèle de test permet la personnalisation et la maintenance de la solution de test, en fournissant une séparation claire entre les éléments du fournisseur de la solution de test et ceux des utilisateurs, avec des points d'extension prédéfinis. Le méta-modèle de test conserve également une séparation entre les concepts structurels et comportementaux. Les concepts structurels sont entrés à l'aide d'un éditeur graphique, un éditeur textuel étant offert pour les concepts comportementaux. Tous les éléments sont intégrés de façon homogène, avec des restrictions sur le comportement qui dépendent du type de la structure. Dans l'ensemble, l'approche dirigée par les modèles devrait contribuer non seulement à une certaine homogénéisation à un niveau abstrait, mais aussi à l'évitement de fautes dans les tests. Certaines erreurs de programmation sont évitées par construction ou détectées par des contrôles automatisés effectués sur les modèles de test.

Dans la Section 5.1 nous présentons brièvement le langage de méta-modélisation que nous avons utilisé: *Eclipse Modeling Framework* (EMF) Ecore. Les sections suivantes présentent les éléments du méta-modèle, un aperçu étant donné dans la Figure 5.1. Nous présentons d'abord la séparation entre la partie du fournisseur de la solution de test et celle de l'utilisateur de la solution de test dans la Section 5.2. Ensuite, nous discutons la partie de l'utilisateur et le contexte de test dans la Section 5.3. Ses concepts structurels haut/bas-niveau (par exemple, des cas de test, des composants de test, des sections de test pour un cas de test) sont présentés dans les Sections 5.4 et 5.5 respectivement, tandis que ses concepts comportementaux (par exemple, l'appel d'actions de test) sont discutés dans la Section 5.6. Enfin, nous discutons l'éditeur de test mixte (graphique et textuel) dans la Section 5.7. La Section 5.8 présente la conclusion.

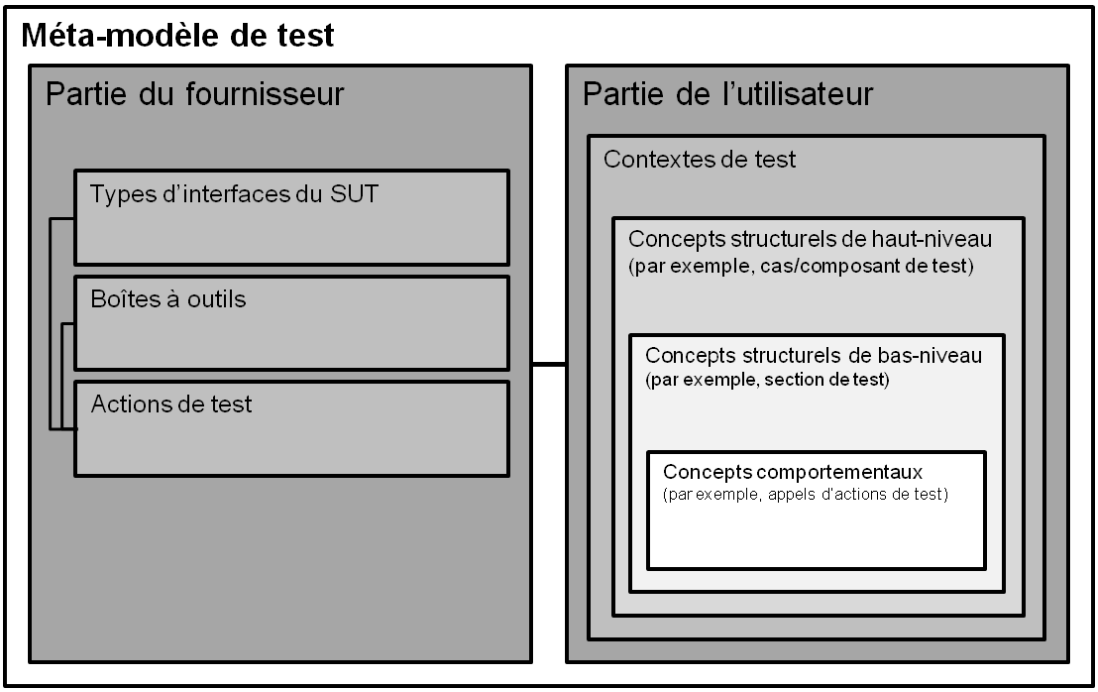

Figure 5.1: Méta-modèle de test - Vue haut-niveau -

5.1 Eclipse Modeling Framework (EMF) Ecore

Nous avons retenu EMF Ecore [119] comme langage de méta-modélisation. EMF permet d'accéder à des outils associés pour produire des éditeurs graphiques spécialisés (Graphical Modeling Framework (GMF) [121], Graphiti [122]), éditeurs textuels (Xtext [123]), vérificateurs (Object Constraint Language (OCL) [124]), ainsi que des générateurs de code (Acceleo [125], Xpand [126]).

5.2 ProviderData et UserData

La racine du méta-modèle de test est la `Database`. Elle contient deux parties: `ProviderData` et `UserData`. Il s'agit d'un premier choix important de méta-modélisation : l'utilisateur de la solution de test reçoit un modèle de test pré-instancié de la part du fournisseur de la solution de test, où seule la partie `ProviderData` est remplie. Le fournisseur de la solution de test a la possibilité de créer différentes variantes pour ses clients, ainsi que mettre à jour celles qui existent déjà. La partie du fournisseur de la solution de test offre des points d'extension pour des types de SUT, des types d'interfaces, des boîtes à outils et des actions de test associées.

5.3 TestContext

La partie `UserData` contient des éléments `TestContext`. Le concept de contexte de test est inspiré de celui proposé dans le UML Testing Profile (UTP) [94]. Il sert de conteneur pour une collection de cas de test appliqués à un SUT, avec une architecture de composants de test.

Le `SystemUnderTest` décrit les interfaces de l'entité testée selon son ICD. Le `TestCase` contrôle l'exécution parallèle d'éléments `TestComponent`. Un composant de test peut être instancié plusieurs fois dans un cas de test, via des éléments `TestComponentInstance`. Les composants de test interagissent avec le SUT. Ils peuvent également interagir les uns avec les autres, par l'intermède d'éléments `SharedData` et `Event` déclarés globalement dans le contexte de test. Pour l'architecture de test, nous avons défini une politique avec un producteur et potentiellement plusieurs consommateurs. La `TestArchitecture` associée à un cas de test détermine l'instance du composant de test qui produit une certaine donnée globale ou un événement. Elle relie également les interfaces formelles (le cas échéant) d'un composant de test avec les interfaces du SUT, les données partagées ou les événements. Les cas de test peuvent être regroupés au sein soit d'éléments `TestSuite` (pour la définition de l'ordre d'exécution) soit `TestGroup` (pour grouper des tests qui partagent une propriété commune).

Sur le plan conceptuel, le contexte de test est en fait divisé en trois niveaux hiérarchiques :

- concepts structurels de haut-niveau (par exemple, `TestCase`, `TestComponent`),
- concepts structurels de bas-niveau (par exemple, la `TestSection` d'un `TestCase`),
- concepts comportementaux (par exemple, `TestCaseStatement`, `TestComponentStatement`).

5.4 Concepts structurels de haut-niveau

Les diférents concepts structurels de haut-niveau de notre méta-modèle sont:

- `SystemUnderTest`,
- `TestCase`,
- `TestComponent` et la gestion du verdict,
- `TestArchitecture`,
- `TestGroupe` et `TestSuite`.

Ces concepts sont décrits dans le tome en anglais de ce mémoire, dans les Sous-sections 5.4.1 à 5.4.6.

5.5 Concepts structurels de bas-niveau

Tous les concepts structurels de haut-niveau (le cas de test et les types de composants de test) ont leur comportement organisé en des concepts structurels de bas-niveau.

Un cas de test organise son comportement à l'intérieur de conteneurs `TestSection`. Un composant de test organise son comportement à l'intérieur de conteneurs `TestComposantElement`. Un moniteur de test a un `TestMonitorPredicateElement` et un `TestMonitorContainerElement`. Un composant de test séquentiel a un ensemble d'éléments `SequentialBlock`.

Enfin, un composant de test cycle-par-cycle a un certain nombre de concepts structurels de bas-niveau qui permettent à un ingénieur de test de spécifier des comportements différents pour chaque cycle ou séquence de cycles : `Cycle`, `IteratedCycle` et `ConditionRepeatedCycle`. En instanciant les éléments décrits ci-dessus à plusieurs reprises, un ingénieur de test peut facilement spécifier un comportement complexe, tel que celui présenté dans la Figure 5.2.

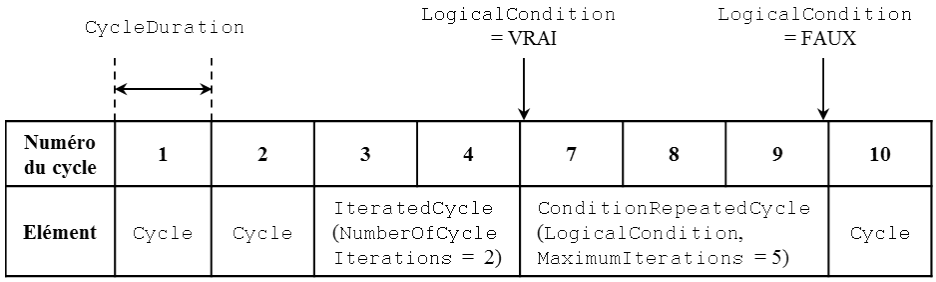

Figure 5.2: CycleByCycleTestComponent - Exemple de comportement complexe -

5.6 Concepts comportementaux

Chacun des concepts structurels de bas-niveau possède un `Behavior` spécifique en fonction de son type, ce comportement étant lui-même composé d'éléments `Statement` spécifiques.

Trois types d'instructions ont été définis: `ExecutionFlowStatement`, `BasicStatement` et `SpecificStatement`.

Ces types d'instructions sont décrits dans le tome en anglais de ce mémoire, dans les Sous-sections 5.6.1 à 5.6.3.

5.7 Environnement de développement de modèles de test

L'Eclipse Modeling Framework (EMF) [119] offre l'accès à un large éventail d'outils, tels que ceux mentionnés au début de ce chapitre. Il est capable de générer automatiquement un éditeur graphique de modèles à partir d'un méta-modèle. Cet éditeur graphique a été utilisé pour les concepts structurels haut/bas-niveau du méta-modèle de test. Pour les concepts comportementaux, un éditeur textuel a été élaboré avec Xtext [123].

L'éditeur graphique présente une structure en forme d'arbre. Des menus contextuels sont offerts. L'éditeur graphique ne montre que les concepts structurels des parties `UserData` et `ProviderData`. GMF [121] et Graphiti [122] offrent la possibilité de concevoir des éditeurs graphiques encore plus ergonomiques et riches.

Xtext [123] est un environnement open-source pour développer des langages de programmation et des langages dédiés (DSL). Un environnement de développement intégré (IDE) basé sur Eclipse est généré. L'IDE propose, entre autres fonctionnalités : la coloration syntaxique et la complétion de code. Les concepts comportementaux sont traités dans l'éditeur textuel.

Dans le Chapitre 7, nous présentons une étude de cas où nous utilisons cet environnement de développement de tests basé-modèles.

5.8 Conclusion

Dans ce chapitre, nous avons présenté le méta-modèle de test que nous avons défini.

L'un des défis consistait à intégrer de façon homogène l'ensemble riche de concepts spécifiques au domaine. Nous avons atteint cet objectif, mais le méta-modèle de test résultant est naturellement complexe. Dans la perspective d'une future industrialisation, cette complexité pourrait être masquée par le développement d'assistants pour guider l'ingénieur de test et remplir automatiquement certains éléments du modèle de test.

Un choix a dû être fait concernant les différentes parties où cette complexité devait être insérée. Par exemple, certaines informations pouvaient apparaître dans le méta-modèle de test, les règles OCL, les classes Java d'analyse des modèles de test ou les documents informels, ainsi qu'à l'intérieur de la représentation textuelle. Notre choix a plutôt été de privilégier le méta-modèle, en y insérant le plus possible d'informations sur les concepts et leurs relations.

Un autre problème que nous avons rencontré concerne les deux niveaux utilisés dans l'ingénierie dirigée par les modèles : le méta-modèle et le modèle avec une seule étape d'instanciation entre eux. Nous avons rencontré des cas où trois niveaux sont nécessaires. Dans de tels cas, nous avons utilisé le motif de conception *Type Objet*

[127]. Bien que ce motif offre une flexibilité remarquable, son utilisation a été l'un des facteurs qui ont conduit à la croissance de la complexité du méta-modèle de test.

Concernant les règles OCL, nous les avons trouvées très pratiques pour exprimer des contraintes sur les modèles de test. Toutefois, elles ont tendance à croître rapidement en complexité. En outre, nous avons constaté qu'il est difficile d'exprimer certains comportements avec OCL. Dans ces cas, il est possible d'appeler des opérations extérieures codées en Java à partir d'OCL, qui feraient les vérifications nécessaires.

Le développement des éditeurs graphique et textuel pour notre démonstrateur a été très rapide, car notre premier prototype ne nécessitait que des fonctionnalités de base. Le prototype est appelé STELAE (*Systems TEst LAnguage Environment*). Notre évaluation du développement d'éditeurs plus ergonomiques et plus riches, avec des technologies telles que Graphical Modeling Framework (GMF) [121] ou Graphiti [122], nous amène à penser que pour un produit industriel il faudrait un effort beaucoup plus grand que pour notre premier prototype.

Enfin, nous tenons à mentionner que certaines activités effectuées manuellement par les ingénieurs de test sur les modèles de test doivent être automatisées dans le futur, comme la définition des interfaces du SUT qui peut être automatisée par l'analyse du document ICD.

6

Implémentation des modèles de test

Dans ce chapitre, nous illustrons la manière dont les modèles de test (conformes au méta-modèle de test) sont implémentés par des transformations modèle-vers-texte basées-motifs. Pour cette tâche, l'outil Acceleo [125] a été choisi en raison de son étroite intégration au sein d'EMF [119]. Le langage de test cible est basé sur le langage de programmation Python et a été développé par Cassidian Test & Services. Nous nous référons à ce langage de test comme PL_5 dans le reste de ce document. PL_5 est exécuté au-dessus de la plate-forme de test d'intégration U-TEST Real-Time System [3].

Notre mise en œuvre ne couvre qu'un sous-ensemble des concepts du méta-modèle de test, qui ont été considérés comme suffisants en ce qui concerne les études de cas (Chapitre 7). Nous avons suivi une méthodologie pour couvrir les concepts qui sont offerts par PL_5 complètement, partiellement ou pas du tout, afin d'étudier les spécificités et les difficultés de la génération automatique de code dans ces différents cas.

La Section 6.1 présente les fonctionnalités de l'outil Acceleo. La Section 6.2 présente le langage de test exécutable ciblé basé sur Python PL_5. La Section 6.3 discute la manière dont les concepts du méta-modèle de test ont été implémentés par de la génération automatique de code basée-motifs, en ciblant les fonctionnalités de PL_5. La Section 6.4 conclut ce chapitre.

6.1 Outil modèle-vers-texte Acceleo

Acceleo est une implémentation du standard Meta-Object Facility (MOF) [130] Model to Text Language (MTL) [131] par le Object Management Group (OMG) [129]. Dans notre cas, il prend en entrée un modèle de test et des motifs de génération automatique de code. Il produit en sortie des fichiers dans le langage de test cible. Un motif de

génération automatique de code définit le lien entre les concepts du méta-modèle de test et la structure des fichiers et du code du langage de test (Figure 6.1).

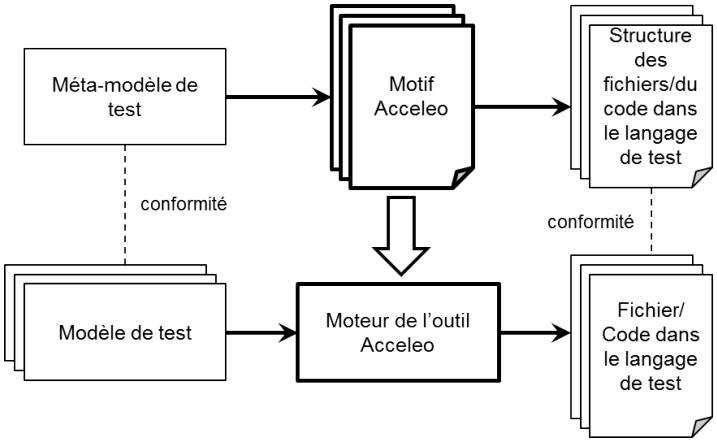

Figure 6.1: Architecture de l'outil Acceleo -

L'approche commence par la définition d'un modèle l'écriture du code et des fichiers correspondants. Ce n'est qu'ensuite que nous définissons les modules/motifs de génération automatique de code qui produisent le code et les fichiers à partir du modèle. Il s'agit d'une approche bien connue dans la littérature qui a montré ses avantages [132].

Les éléments EReference avec une valeur "vrai" pour le Containment apparaissent au niveau des modules Acceleo comme des déclarations d'importation. Les éléments EReference avec une valeur "faux" pour le Containment apparaissent comme des déclarations d'importation aussi, mais au niveau des modules Python. La projection de l'architecture du méta-modèle sur les modules/motifs Acceleo et sur les fichiers de code générés automatiquement peut faciliter la maintenance de la fonctionnalité de génération automatique de code. Les ingénieurs peuvent facilement trouver un module existant Acceleo. Lors de l'ajout de nouveaux motifs Acceleo, ils doivent être ajoutés en respectant cette contrainte.

Pour plus d'informations, le lecteur pourra se reporter au tome en anglais de ce mémoire.

6.2 PL_5: un langage de test basé sur Python

Le tome en anglais de ce mémoire présente et illustre un sous-ensemble des fonctionnalités offertes par PL_5 :

- cas de test,
- accès aux interfaces du SUT et actions de test associées,

- composant de test,
- gestion du verdict,
- suite de test.

Ces fonctionnalités sont discutées dans les Sous-sections 6.2.1 à 6.2.5 du tome en anglais de ce mémoire.

Pour les cas où PL_5 n'offre pas nativement une fonctionnalité présente dans le méta-modèle de test, ou ne la propose que partiellement ou d'une manière incompatible avec le méta-modèle, nous proposons les solutions suivantes :

- implémenter les fonctionnalités ou les parties manquantes à l'aide de fonctionnalités de bas-niveau (par exemple, un composant de test implémenté avec du multi-threading Python),
- générer une sur-couche autour des fonctionnalités existantes (par exemple, les instructions PL_5 d'accès aux interfaces du SUT à l'aide de leur identifiant textuel sont enveloppées dans des structures orientées-objet navigables fortement-typées), et le code qui utilise la sur-couche,
- définir des transformations modèle-vers-texte seulement pour les modèles de test qui sont compatibles avec les fonctionnalités offertes (par exemple, PL_5 offre des moniteurs de test dont la seule action possible est de changer le verdict de test),
- abandonner la mise en œuvre de la fonctionnalité (par exemple, l'accès aux interfaces du SUT de bas-niveau, telles que les bus physiques, n'est pas offerte par PL_5).

La deuxième solution est intéressante afin de minimiser les différences entre le méta-modèle de test et PL_5, pour une définition et maintenance des motifs de génération automatique de code facilitées.

6.3 Architecture des modules/motifs Acceleo

Les concepts du méta-modèle de test ont été mis en œuvre dans le démonstrateur de la façon suivante :

- le `ProviderData` : les éléments `ConnectionPointType` et les éléments `TestAction` associés du méta-modèle de test → sur-couche autour des instructions PL_5,
- le `SystemUnderTest` du méta-modèle de test → utilisation de la sur-couche du `ProviderData`,

- le `TestCase` du méta-modèle de test → spécialisations de la classe ATLTestCase de PL_5,
- le `TestComponent` (à l'exception du `TestMonitor`) du méta-modèle de test → génération de code utilisant des fonctionnalités bas-niveau de Python,
- le `TestMonitor` du méta-modèle de test → implémentation partielle vers les spécialisation de la classe ATLAsyncTest de PL_5,
- la `TestArchitecture` du méta-modèle de test → génération de code utilisant des fonctionnalités bas-niveau de Python,
- la `TestSuite` du méta-modèle de tes → implémentation partielle vers les suites de test de l'environnement de PL_5,
- Verdict Management → PL_5 fonctionnalité mise-à-jour par génération de code utilisant des fonctionnalités bas-niveau de Python.

Pour plus d'informations et des exemples le lecteur pourra se rapporter aux Sous-sections 6.3.1 à 6.3.5 du tome en anglais de ce mémoire.

6.4 Conclusion

Dans ce chapitre, nous avons présenté la mise en œuvre de certains concepts du méta-modèle de test (cas de test, composant de test, interfaces du SUT et actions de test correspondantes, la gestion du verdict de test), grâce à des transformations modèle-vers-texte basées-motifs. Nous avons ciblé un langage de test nouveau que nous avons appelé PL_5, qui est exécutable sur la plate-forme de test d'intégration U-TEST Real-Time System [3]. L'outil Acceleo [125] a été utilisé pour la génération automatique de code. Pour notre application, nous avons pris la structure du méta-modèle de test comme principe d'organisation des modules/motifs Acceleo, ainsi que des fichiers et du code Python générés automatiquement. Nous avons constaté que cette approche était utile et nous a permis de mieux naviguer dans le riche ensemble de fonctionnalités spécifiques au domaine. Nous avons ciblé des fonctionnalités déjà offertes par PL_5 (par exemple, le cas de test), des fonctionnalités qui manquaient (par exemple, le composant de test), ainsi que des fonctionnalités qui étaient partiellement offertes (par exemple, le moniteur de test).

La génération automatique de code à partir de modèles de test est quasi-instantanée.

Une approche bien connue pour une définition simple et rapide de motifs de génération automatique de code est d'abord de choisir un exemple simple de la source (modèle de test), puis définir la cible attendue (ce que les fichiers générés et le code seraient) et seulement ensuite développer des motifs qui mettent en correspondance les deux

[132]. Notre expérience le confirme : nous n'avons rencontré aucune difficulté lors de l'élaboration des motifs, en étant guidés par des cas d'utilisation. Actuellement, environ 40% des concepts présents dans le méta-modèle de test ont été implémentés.

La mise en œuvre n'a pas soulevé des questions importantes concernant le méta-modèle de test, qui n'a pas subi de nombreuses modifications. La mise en œuvre n'a pas soulevé des questions importantes concernant l'analyse des langages de test non plus. Comme PL_5 n'existait pas quand nous avons finalisé notre analyse de langages de test, son utilisation pour la mise en œuvre de notre approche a permis de mettre à l'épreuve la généricité de notre analyse, ainsi que notre méta-modèle de test. Même si PL_5 avait existé à ce moment-là, il n'aurait pas changé les résultats de notre analyse de langages de test, ni la structure du méta-modèle de test.

L'approche basée sur des transformations modèle-vers-texte avec des motifs structurés a suffi pour notre mise en œuvre. Nous n'avons pas jugé utile de recourir à des techniques plus complexes pour implémenter les modèles de test [132, 133].

Nous tenons également à mentionner une question qui est importante dans un environnement industriel: le débogage. Ce sujet peut être discuté à partir de deux points de vue complémentaires. Le premier concerne la mise au point des motifs de génération automatique de code par les ingénieurs du fournisseur de la solution de test. Le second concerne la mise au point de modèles de test par les ingénieurs de test (les ingénieurs de test ne doivent pas déboguer le code généré automatiquement, en conformité avec la philosophie de l'ingénierie dirigée par les modèles).

En ce qui concerne les fautes à l'intérieur des motifs de génération de code, nous n'avons actuellement pas de solution de validation. Ce serait un problème difficile à résoudre en raison du fait que les langages de test cibles propriétaires n'ont pas de sémantique formelle. Nous avons essayé de simplifier le problème en proposant :

- une sur-couche autour des instructions des langages de test cibles, sur la base de la représentation textuelle du méta-modèle de test, afin d'améliorer la clarté du lien entre le code généré automatiquement et la représentation textuelle (aller encore plus loin avec la méthode de définition de sur-couches serait possible),

- une approche agile dirigée par des cas d'utilisation, les motifs de génération automatique de code sont définis et vérifiés sur la base d'un cas d'utilisation contenant le modèle d'entrée et les fichiers et le code de sortie attendus,

- une organisation des motifs de génération des fichiers et du code basée sur l'architecture du méta-modèle de test, qui facilite leur définition, navigation et maintenance (de plus, l'environnement Acceleo propose l'auto-complétion des motifs avec la navigation du méta-modèle et l'insertion des données choisies dans les motifs).

En ce qui concerne les fautes à l'intérieur des modèles de test, ce problème est commun à toutes les approches dirigées par les modèles, car les outils de débogage existants ont été conçus pour travailler au niveau du code. Pour notre approche spécifique, diverses solutions ont été identifiées :

- aborder la question du débogage au niveau du modèle de test, par exemple en insérant le concept de point d'arrêt dans le méta-modèle de test, en utilisant des techniques d'ingénierie aller-retour basées sur des annotations, en ajoutant des règles OCL supplémentaires ou en développant un outil spécifique,
- rapprocher les grammaires de la représentation textuelle de la partie comportementale du méta-modèle de test et du langage de test cible : en modifiant la syntaxe concrète et en ajoutant des sur-couches (nous avons suivi cette approche, mais il y a des limites: la syntaxe concrète que nous étions en mesure de définir n'était pas exactement celle de Python),
- une solution plus radicale (autorisée par la modularité du méta-modèle de test) serait de dissocier les aspects structurels et comportementaux du méta-modèle de test, avec la réutilisation des motifs pour la partie structurelle, tandis que la partie comportementale serait définie directement dans le langage de test cible. Cette solution perdrait des facilités d'évitement de fautes, de personnalisation et de portabilité que le méta-modèle de test offre actuellement, mais permettrait la réutilisation des outils existants (au moins pour le comportement, si ce n'est pas pour les concepts structuraux). De telles utilisations partielles de nos travaux sont discutées dans les perspectives du chapitre 8.

Le chapitre suivant présente deux études de cas inspirés de la réalité.

7

Cas d'étude

Dans ce chapitre, nous illustrons les fonctionnalités de STELAE sur deux études de cas simplifiées inspirées de la vie réelle. Elles ciblent des modèles simplifiés du *Flight Warning System* (FWS) (Section 7.1) et du *Air Data Inertial Reference System* (ADIRS) (Section 7.2). Nous avons présenté STELAE et l'étude de cas ADIRS dans [135].

7.1 FWS - Synthèse d'alarme incendie moteur

Cette première étude de cas est inspirée d'un exemple réel ciblant un système de gestion des alarmes en vol. Nous avons choisi cette étude inspirée du FWS car nous avons eu accès au document ICD réel. Notre FWS simplifié emploie des bus AFDX pour le transport des paramètres applicatifs d'entrée et de sortie. Nous avons développé une simulation simple du comportement du FWS en PL_1.

Un FWS surveille d'autres systèmes de l'avion, et au cas où des pannes ou des conditions de vol dangereuses sont détectées, informe l'équipage de ces problèmes par l'intermédiaire d'alarmes.

Nous considérons ici la synthèse d'une alarme de sortie pour une situation d'incendie moteur, basée sur quatre alarmes partielles d'entrée. Cette logique est validée en deux étapes. La première active les quatre détecteurs partiels et vérifie le démarrage de l'alarme de sortie en moins d'une seconde. Ensuite, deux parmi les quatre alarmes d'entrée sont désactivées et l'arrêt de l'alarme de sortie après 17 secondes est vérifié.

7.2 ADIRS - Valeur consolidée de la vitesse de l'avion

Cette étude de cas porte sur la fonctionnalité de calcul d'une valeur de vitesse avion consolidée. Elle est inspirée d'un exemple réel ciblant un système qui estime des paramètres inertiels et air de l'avion [65]. Nous n'avons pas eu accès au document ICD réel, ni à la conception réelle de l'ADIRS.

Nous avons développé une simulation simple d'une partie du comportement de l'ADIRS en Python. L'ADIRS traite l'acquisition de plusieurs paramètres applicatifs nécessaires pour le système de contrôle de vol. Pour chacun de ces paramètres applicatifs, des capteurs redondants existent et une valeur consolidée est calculée à partir de l'ensemble des valeurs d'entrée disponibles.

Nous considérons ici la vitesse de l'avion. Les valeurs des trois paramètres en entrée (AC_SPEED_1/2/3) sont utilisées pour calculer la valeur consolidée du paramètre de sortie (AC_SPEED).

La logique de l'ADIRS est la suivante :

- **comportement nominal** : La valeur consolidée est la médiane des trois valeurs d'entrée, si la médiane ne s'écarte pas des deux autres valeurs. La divergence est mesurée comme la différence entre la valeur de la médiane et les deux autres valeurs. La médiane est divergente si ces différences dépassent un certain seuil au cours de trois cycles d'exécution consécutifs.

- **comportement dégradé** : Si la médiane des valeurs d'entrée diverge par rapport aux deux autres pendant plus de trois cycles, la source divergente est définitivement éliminée. La valeur consolidée est alors la moyenne des deux valeurs restantes.

Pour ce système sous test, nous considérons ici deux cas de test vérifiant son comportement :

- **cas de test pour le comportement nominal** : Vérifier que la valeur consolidée reste égale à la médiane des trois valeurs d'entrée en présence d'une oscillation sinusoïdale de faible amplitude qui ne rend pas les valeurs d'entrée divergentes.

- **cas de test pour le comportement dégradé** : Injecter une divergence sur l'une des trois valeurs d'entrée et vérifier que la valeur consolidée est égale à la moyenne des deux valeurs restantes. Vérifier que la source divergente est définitivement éliminée, même si l'écart est corrigé.

Ces cas de test doivent être exécutés sur toutes les combinaisons de paramètres applicatifs en entrée. Comme le comportement testé est identique dans tous les cas, la réutilisation d'un composant de test paramétré serait pratique.

Comme on peut le voir sur la Figure 7.1, nous avons modélisé ces cas de test, ainsi que d'autres. Dans la partie centrale de la figure nous pouvons observer l'éditeur graphique arborescent, pour les éléments structurels des tests. À sa droite nous pouvons observer l'éditeur textuel, pour les éléments comportementaux des tests. La vue "Model Projects"montre les fichiers générés automatiquement en PL_5. La vue "Console"montre les traces d'exécution du cas de test pour le comportement dégradé. Pour ce cas de test

vous pouvez observer deux verdict locaux `Pass`. La vue "Test Management"montre la synthèse automatique du verdict global des cas de test à partir des verdict locaux.

La Figure 7.2 montre la vue "Runtime"du composant U-TEST MMI, où on peut observer la modification des valeurs de nos différents paramètres applicatifs pendant l'exécution de différents cas de test.

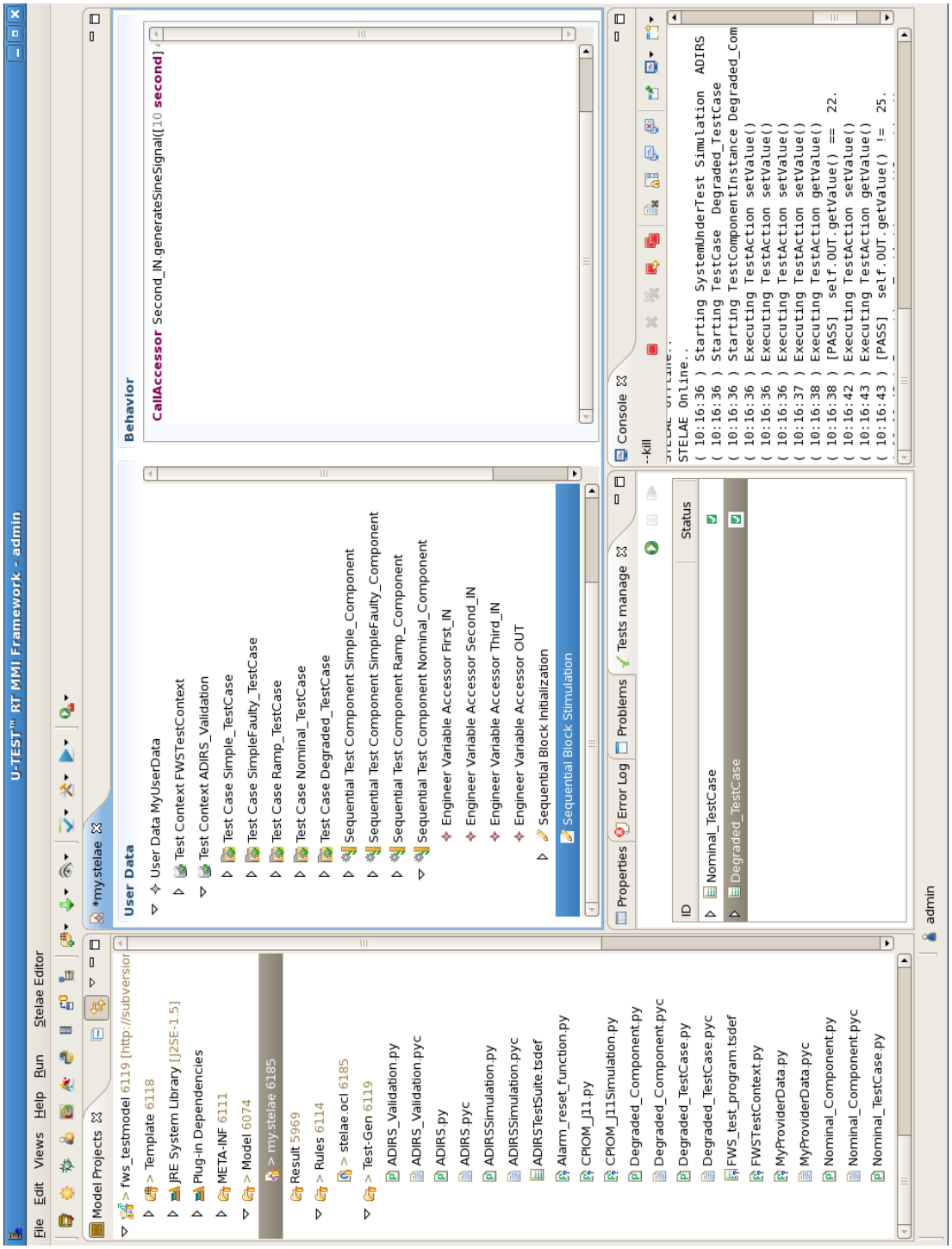

Figure 7.1: U-TEST MMI - Perspective STELAE - ADIRS -

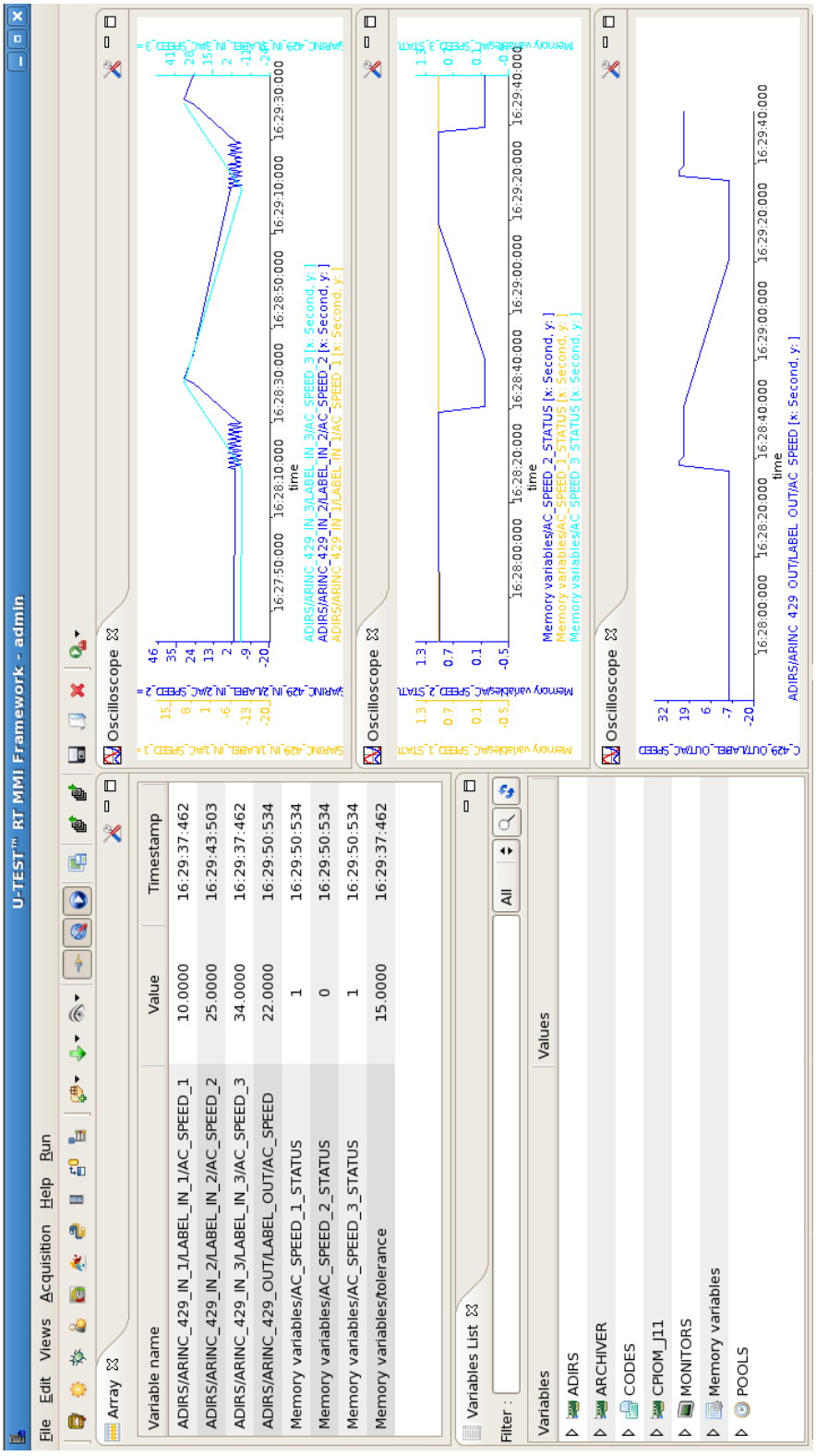

Figure 7.2: U-TEST MMI - Perspective Runtime - ADIRS -

7.3 Conclusion

Dans ce chapitre, nous avons illustré les fonctionnalités de notre premier prototype appelé STELAE sur deux études de cas inspirées de la vie réelle.

Nous avons couvert différentes activités des ingénieurs de test sur ces études de cas: la définition de modèles de test, la génération automatique de fichiers et de code, ainsi que l'exécution sur une plate-forme de test d'intégration réelle: U-TEST Real-Time System [3]. Actuellement nous pouvons démontrer ces fonctionnalités sur des cas de test de complexité faible à moyenne.

Une fois les interfaces du SUT entrées, la définition des différents modèles de test de nos études de cas n'a nécessité que quelques minutes.

Une de nos perspectives est d'illustrer les fonctionnalités de STELAE sur des études de cas plus complexes. Une autre perspective serait de définir une batterie de spécifications de cas de test de référence, et ensuite de comparer l'expérience des ingénieurs de test sur STELAE et avec des langages de test existants.

8

Conclusion et perspectives

Ces travaux ont étudié la définition d'une approche d'ingénierie dirigée par les modèles pour le développement des tests en boucle pour les systèmes avioniques embarqués. Nous étions motivés par le fait que les solutions existantes de développement de tests utilisées dans notre domaine ne répondent plus aux besoins exprimés par les intervenants: avionneurs, équipementiers/systémiers et fournisseurs de solutions de test. Nous proposons une approche dirigée par les modèles dans laquelle les modèles de test prennent la position centrale de l'activité de développement des test, en remplaçant le code de test qui occupe actuellement cette position. Afin de définir notre propre méta-modèle de test sous-jacent, nous avons eu besoin de poursuivre l'analyse de la pratique actuelle.

Notre première contribution consiste en l'analyse d'un échantillon de quatre langages de test propriétaires actuellement utilisés dans notre domaine industriel. À des fins de comparaison, nous avons également choisi deux langages de test en dehors du domaine de l'avionique: TestML [114], qui est issu d'un projet de recherche dans le domaine de l'automobile, et TTCN-3 [82], qui est un standard international mature utilisé pour le test de protocoles de télécommunications et de systèmes distribués. Nous nous sommes concentrés sur quatre grandes catégories de caractéristiques présentées par les langages de test :

- l'organisation des tests,
- l'abstraction et l'accès aux interfaces du SUT,
- les instructions du langage pour contrôler l'exécution du test et réaliser les interactions de test,
- la gestion du temps.

Notre analyse a suivi une méthodologie claire. Pour chaque fonctionnalité qui a été identifiée, nous avons examiné la façon dont elle est offerte par les langages de

test. Cette analyse a été basée sur notre accès à des informations disponibles pour chaque langage de test, telles que : manuels d'utilisation, présentations, articles et livres, ainsi que des spécifications et des procédures de test dédiées. Dans le cas où un langage de test n'offrait pas une fonctionnalité ou l'offrait d'une manière différente par rapport aux autres langages de test de notre échantillon, nous avons discuté avec les ingénieurs de test, en leur demandant des éclaircissements. Cette analyse nous a permis d'identifier un ensemble riche de fonctionnalités/concepts spécifiques au domaine, ainsi que les meilleures pratiques et les pièges à éviter. L'analyse a confirmé les hypothèses de départ qui ont motivé notre travail: le monde des langages de test dans notre contexte industriel est hétérogène, mais il a été possible d'identifier des concepts communs. Les conclusions de notre analyse de langages de test ont été résumées dans une liste de principes directeurs de méta-modélisation qui ont guidé la définition de notre méta-modèle de test. En plus de représenter le fondement de notre travail de méta-modélisation, les résultats de cette première contribution sont immédiatement utiles aux intervenants du domaine, leur permettant d'identifier les manières dont ils pourraient améliorer les langages de test existants ou en définir de nouveaux à partir de zéro.

La seconde contribution consiste en la définition d'un méta-modèle de test qui intègre un ensemble riche de concepts spécifiques au domaine d'une manière homogène, tout en tenant compte des meilleures pratiques. Le méta-modèle de test sépare les préoccupations des intervenants: fournisseur de solutions de test et utilisateurs de solutions de test. Par conséquent, il permet au fournisseur de la solution de test de personnaliser et maintenir plus facilement la solution de test. Le méta-modèle de test sépare également les aspects structurels des aspects comportementaux. Cette séparation a conduit au développement d'un éditeur mixte, où les éléments structurels sont saisis graphiquement et les comportements sont codés sous une forme textuelle. En dépit de cette séparation, les éléments structurels et comportementaux sont intégrés d'une manière cohérente dans le méta-modèle de test, avec des contraintes sur le type de comportement spécifiable pour un élément structurel. Ces contraintes visent l'objectif d'évitement de fautes. En outre, le méta-modèle de test propose des éléments fortement typés pour les interfaces du SUT et pour les actions de test associées. Nous avons utilisé la méta-modélisation non seulement comme un instrument afin de formaliser nos concepts et leurs relations, mais aussi pour avoir accès à un riche ensemble d'outils existants libres : éditeurs de modèles graphiques et textuels, outils de vérification et de génération de code automatiques. Cela nous a permis de développer rapidement notre premier prototype.

Notre troisième contribution a consisté en la mise en œuvre d'un premier prototype au-dessus d'une plate-forme de test d'intégration réelle disponible dans le commerce: U-TEST Real-Time System [3] développée par Cassidian Test & Services. Nous avons

appelé le prototype: System TEst LAnguage Environment (STELAE). Nous avons utilisé des transformations modèle-vers-texte basées-motifs, en ciblant un langage de test exécutable basé sur Python. Le langage de test cible n'existait pas à l'époque de notre première analyse du domaine et par conséquent nous a fourni l'occasion de mettre à l'épreuve la généricité de notre méta-modèle de test, ainsi que notre analyse de langages de test. Nous avons proposé une méthode pour organiser les motifs de génération automatique de code, ainsi que le code et les fichiers générés automatiquement, afin de simplifier la tâche de leur développement, en se basant sur l'architecture du méta-modèle de test. Nous n'avons pas considéré des fonctionnalités plus avancées de transformation, l'approche basée-motifs étant suffisante pour nos besoins. Pour le développement de tests par les ingénieurs de test, STELAE intègre un environnement mixte (graphique et textuel) personnalisable de développement de modèles de test, qui a été ajouté au composant Man-Machine Interface (MMI)de U-TEST Real-Time System. Enfin, ce prototype nous a permis d'évaluer la technologie employée à l'égard d'une possible l'industrialisation future. Notre méta-modèle a résisté à l'épreuve de l'implémentation, ne nécessitant que des adaptations très limitées pendant cette phase de notre projet.

Notre prototype STELAE a été utilisé pour la modélisation de deux cas d'études inspirés de la vie réelle. On peut actuellement illustrer la définition de modèles de test, leur implémentation et exécution, pour des cas de test de complexité simple à moyenne.

Une première perspective de notre travail consiste à élargir l'échantillon de langages de test analysés avec des langages de test utilisés dans d'autres domaines, afin d'évaluer la possibilité d'avoir une solution unique homogène entre des différents secteurs industriels (par exemple, entre les industries avionique et de l'automobile).

Une deuxième perspective concerne la sémantique du méta-modèle de test. Actuellement, la plupart de la sémantique statique se trouve à l'intérieur du méta-modèle de test, le reste étant réparti à l'intérieur de règles OCL, classes Java, documents textuels informels, et dans la représentation textuelle. Une sémantique formelle serait souhaitable, en particulier pour la dynamique d'exécution des modèles de test. Un problème majeur est que les langages de test exécutables ciblés pour la mise en œuvre des modèles de test n'ont pas une sémantique formelle eux-mêmes. Le même problème se pose pour TTCN-3. En outre, une sémantique unique semble difficilement définissable pour le méta-modèle de test, en raison des différents points de variation. Par exemple, la sémantique d'un composant de test périodique dans une situation matériel-en-boucle serait différente de celle d'un même type de composant de test utilisé dans une situation modèle-en-boucle, où une vue logique des cycles d'exécution suffit.

Une autre perspective serait de finaliser la mise en œuvre de l'environnement de développement de modèles de test. Il faudrait ajouter les fonctionnalités ergonomiques nécessaires dans un environnement de production, tout en réduisant les problèmes de

synchronisation multi-éditeurs que nous avons identifiés. L'automatisation des activités actuellement manuelles dans notre prototype, telles que l'extraction des interfaces du SUT par analyse d'un ICD, serait un ajout bienvenu. Par ailleurs, des assistants guidant les ingénieurs de test et leur permettant l'accès uniquement à des parties restreintes du méta-modèle de test, en fonction de leurs choix, pourrait simplifier l'expérience utilisateur et cacher la complexité du méta-modèle de test.

Une quatrième perspective importante porte sur l'étude de meilleures fonctionnalités de débogage. Ce problème peut être analysé de deux points de vue: qu'il s'agisse de la mise au point effectuée par le fournisseur de solutions de test sur les motifs de génération automatique de code ou de la mise au point effectuée par l'ingénieur de test sur le modèle de test.

En ce qui concerne les fautes affectant les motifs de génération automatique de code, notre travail a eu comme objectif d'atténuer le problème. Des sur-couches ont été définies autour des instructions du langage de test cible, afin d'améliorer la clarté de la liaison entre le code généré automatiquement et la représentation textuelle du comportement dans le modèle de test. Cette direction peut encore être renforcée. Une approche agile basée sur des cas d'utilisation (des modèles d'entrée et leurs fichiers et code de sortie attendus) a guidé notre définition des motifs de génération automatique de code. Les cas d'utilisation ont aussi servi pour la vérification de la transformation. Enfin, nous avons proposé une organisation des motifs de génération automatique de code et des fichiers/du code générés automatiquement basée sur la structure du métamodle de test, ce qui facilite leur maintenance par des ingénieurs connaissant le méta-modèle de test.

En ce qui concerne les fautes au niveau des modèles de test, trois approches peuvent être envisagées. La première approche consiste à aborder la question de débogage au niveau du modèle de test, par exemple en insérant le concept de point d'arrêt dans le méta-modèle de test, en utilisant des techniques d'ingénierie aller-retour basées sur des annotations ou encore en développant des débogueurs spécifiques. La seconde approche consiste à rapprocher les grammaires de la représentation textuelle de la partie comportementale du méta-modèle de test et du langage de test. La syntaxe concrète définissable pour le comportement du modèle de test sera néanmoins limitée par la structure du méta-modèle de test: bien que la syntaxe concrète de STELAE ressemble à celle de Python, elles ne sont pas identiques. La troisième approche est plus drastique, profitant de la modularité du méta-modèle de test pour remplacer sa partie comportementale par le langage de test cible. Dans ce cas, les outils de débogage existants pourront être réutilisés (pour la partie comportementale), mais les caractéristiques de l'évitement des fautes, de personnalisation et de portabilité de notre approche seraient perdues. Les motifs de génération automatique de code déjà définis, qui ciblent les concepts structurels, pourraient être réutilisés ainsi. Le langage d'implémentation utilisé

devra néanmoins être mis à niveau avec des fonctionnalités offertes par le méta-modèle de test (par exemple, l'exécution en temps borné, les contraintes sur le comportement en fonction de la structure). Il s'agit d'une question importante à analyser dans le futur. De la génération automatique de code à-la-volée serait nécessaire pour accéder à l'intérieur du langage de test à des éléments de structure définis dans le modèle de test. Nous considérons qu'une évaluation plus approfondie de ces questions est nécessaire.

D'autres utilisations partielles de notre travail consistent en la mise à niveau de langages de test existants avec les concepts et les meilleures pratiques issues de notre analyse des langages de test et du méta-modèle de test. En interne Cassidian Test & Services, des discussions en cours portent sur la mise à niveau de PL_5 avec des composants de test et une gestion plus riche du verdict, ainsi que la mise à niveau de la représentation des éléments ICD dans U-TEST Real-Time System avec des constructions fortement-typées, et une liste d'actions attachées aux types d'interfaces du SUT. L'éditeur graphique peut également servir comme base pour la génération de squelettes de procédures de test plus riches que celles actuellement offertes et l'amélioration de l'expérience utilisateur avec PL_1 et PL_5.

Enfin, le lien entre notre approche actuelle et les techniques de test basées sur des modèles (pour la génération de tests à partir de spécifications fonctionnelles du SUT) peut être envisagé. Des cas de test abstraits issus de la spécification du comportement du SUT pourraient être traduits vers nos modèles de test, à des fins d'implémentation et d'exécution.

Bibliographie

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 1, 3

[2] G. Bartley and B. Lingberg. Certification concerns of integrated modular avionics (ima) systems. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, DASC '08, pages 1.E.1–1–1.E.1–12, oct. 2008. doi: 10.1109/DASC.2008.4702766. 1

[3] U-test real-time system. URL `http://www.eads-ts.com/web/products/software/utest.html`. 2, 14, 27, 30, 36, 38

[4] R. G. Ebenau, S. H. Strauss, and S. S. Strauss. *Software Inspection Process*. McGraw-Hill Systems Design & Implementation Series. McGraw-Hill, 1994. ISBN 9780070621664. URL `http://books.google.fr/books?id=kSYiAQAAIAAJ`.

[5] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8. URL `http://dl.acm.org/citation.cfm?id=647348.724445`.

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012.

[7] P. Schnoebelen. *Vérification de logiciels: Techniques et outils du model-checking*. Vuibert informatique. Vuibert, 1999. ISBN 9782711786466. URL `http://books.google.fr/books?id=u0-_PAAACAAJ`.

[8] O. Grumberg and H. Veith. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540698493, 9783540698494.

[9] J. F. Monin. *Comprendre les méthodes formelles: Panorama et outils logiques*. Collection technique et scientifique des télécommunications. Masson, 1996. ISBN 9782225853043. URL `http://books.google.fr/books?id=kLApAQAACAAJ`.

[10] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001. ISBN 0-444-50812-0.

[11] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Survey*, 41(2):9:1–9:76, 2009. ISSN 0360-0300. doi: 10.1145/1459352.1459354. URL `http://doi.acm.org/10.1145/1459352.1459354`.

[12] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990. URL `http://dx.doi.org/10.1109/IEEESTD.1990.101064`.

[13] H. Waeselynck P. Thévenod-Fosse and Y. Crouzet. Software statistical testing. *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series:253–72, 1995.

[14] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Proceedings of the 2001 IEEE 16th International Conference on Automated Software Engineering*, ASE '01, pages 5–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=872023.872550`.

[15] M. Petit and A. Gotlieb. Uniform selection of feasible paths as a stochastic constraint problem. In *Proceedings of the 7th International Conference on Quality Software*, QSIC '07, pages 280–285, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3035-4. URL `http://dl.acm.org/citation.cfm?id=1318471.1318535`.

[16] S. Poulding and J. A. Clark. Efficient software verification: Statistical testing using automated search. *IEEE Transactions of Software Engineering*, 36(6):763–777, 2010. ISSN 0098-5589. doi: 10.1109/TSE.2010.24. URL `http://dx.doi.org/10.1109/TSE.2010.24`.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136. URL `http://dx.doi.org/10.1109/C-M.1978.218136`.

BIBLIOGRAPHIE

[18] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 158–171, New York, NY, USA, 1996. ACM. ISBN 0-89791-787-1. doi: 10.1145/229000.226313. URL http://doi.acm.org/10.1145/229000.226313.

[19] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001. ISBN 0-7923-7323-5. URL http://dl.acm.org/citation.cfm?id=571305.571314.

[20] B. Beizer. *Software Testing Techniques (2nd Edition)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0. 4

[21] G. J. Myers and C. Sandler. *The Art of Software Testing, Second Edition*. John Wiley & Sons, 2004. ISBN 0471469122. 4

[22] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 9780521880381. URL http://books.google.fr/books?id=BMbaAAAAMAAJ. 4

[23] B. Legeard, F. Bouquet, and N. Pickaert. *Industrialiser le test fonctionnel. Pour maîtriser les risques métier et accroître l'efficacité du test*. Collection InfoPro. Dunod, 2011. URL http://hal.inria.fr/hal-00645019. 2ème édition. 304 pages. ISBN : 9782100566563. 4

[24] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.6165. 4

[25] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.232226. URL http://dx.doi.org/10.1109/TSE.1985.232226. 4

[26] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9 (5):193–200, sep 1994. ISSN 0268-6961.

[27] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage - nasa technical report. Technical report, 2001.

[28] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, June 1988. ISSN 0001-0782. doi: 10.1145/62959.62964. URL http://doi.acm.org/10.1145/62959.62964. 4

[29] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006. ISSN 1382-3256. doi: 10.1007/s10664-006-9024-2. URL http://dx.doi.org/10.1007/s10664-006-9024-2. 4

[30] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, FME '93, pages 268–284, London, UK, 1993. Springer-Verlag. ISBN 3-540-56662-7. URL http://dl.acm.org/citation.cfm?id=647535.729387. 4

[31] R. M. Hierons. Testing from a z specification. In *Journal of Software Testing, Verification and Reliability*, volume 7, pages 19–33, London, UK, UK, 1997. 4

[32] L. Van Aertryck, M. V. Benveniste, and D. Le Mtayer. Casting: A formally based software test generation method. In *Proceedings of the 1997 IEEE 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 101–, 1997. URL http://dblp.uni-trier.de/db/conf/icfem/icfem1997.html#AertryckBM97. 4

[33] S. Behnia and H. Waeselynck. Test criteria definition for b models. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems*, volume 1 of *FM '99*, pages 509–529, London, UK, 1999. Springer-Verlag. ISBN 3-540-66587-0. URL http://dl.acm.org/citation.cfm?id=647544.730455.

[34] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models: Research articles. *Software Testing, Verification and Reliability*, 14 (2):81–103, 2004. ISSN 0960-0833. doi: 10.1002/stvr.v14:2. URL http://dx.doi.org/10.1002/stvr.v14:2. 4

[35] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003. URL http://dblp.uni-trier.de/db/journals/stvr/stvr13.html#OffuttLAA03. 4

[36] J. C. Huang. An approach to program testing. *ACM Computing Survey*, 7(3):113–128, 1975. ISSN 0360-0300. doi: 10.1145/356651.356652. URL http://doi.acm.org/10.1145/356651.356652.

[37] Tsunoyama M. Naito, S. Fault detection for sequential machines by transition tours. *Procedings of the 1981 IEEE Fault Tolerant Computing Symposium*, pages 238–243, 1981.

[38] J. C. Rault. An approach towards reliable software. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 220–230, Piscataway, NJ, USA, 1979. IEEE Press. URL http://dl.acm.org/citation.cfm?id=800091.802942.

[39] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978. ISSN 0098-5589. doi: 10.1109/TSE.1978.231496. URL http://dx.doi.org/10.1109/TSE.1978.231496.

[40] K. Sabnani and A. Dahbura. A new technique for generating protocol test. *SIGCOMM Compututer Communication Review*, 15(4):36–43, 1985. ISSN 0146-4833. doi: 10.1145/318951.319003. URL http://doi.acm.org/10.1145/318951.319003.

[41] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992. ISSN 0140-3664. doi: 10.1016/0140-3664(92)90092-S. URL http://dx.doi.org/10.1016/0140-3664(92)90092-S.

[42] A. R. Cavalli, B.-M. Chin, and K. Chon. Testing methods for sdl systems. *Computer Networks and ISDN Systems*, 28(12):1669–1683, 1996. ISSN 0169-7552. doi: 10.1016/0169-7552(95)00125-5. URL http://dx.doi.org/10.1016/0169-7552(95)00125-5.

[43] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 348–359, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5. URL http://dl.acm.org/citation.cfm?id=647765.735847. 4

[44] J. Tretmans. Testing concurrent systems: A formal approach. In JosC.M. Baeten and Sjouke Mauw, editors, *Concurrency Theory*, Lecture Notes in Computer Science - 1664, pages 46–65. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66425-3. doi: 10.1007/3-540-48320-9_6. URL http://dx.doi.org/10.1007/3-540-48320-9_6.

[45] B. Nielsen and A. Skou. Automated test generation from timed automata. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '01, pages 343–357, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41865-2. URL http://dl.acm.org/citation.cfm?id=646485.694454.

[46] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods System Design*, 34(3):238–304, 2009. ISSN 0925-9856. doi: 10.1007/s10703-009-0065-1. URL http://dx.doi.org/10.1007/s10703-009-0065-1.

[47] R. Castanet, O. Kone, and P. Laurencot. On the fly test generation for real time protocols. In *Proceedings of the 1998 7th International Conference on Computer Communications and Networks*, pages 378 –385, October 1998. doi: 10.1109/ICCCN.1998.998798.

[48] T. Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240:167–184, 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.05.051. URL http://dx.doi.org/10.1016/j.entcs.2009.05.051.

[49] M.-C. Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95: Theory and Practice of Software Development, Proceedings of the 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995*, Lecture Notes in Computer Science - 915, pages 82–96. Springer, May 1995. ISBN 3-540-59293-8. 4

[50] B. Marre. Loft: A tool for assisting selection of test data sets from algebraic specifications. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 799–800, London, UK, UK, May 1995. Springer-Verlag. ISBN 3-540-59293-8. URL http://dl.acm.org/citation.cfm?id=646619.697561. 4

[51] B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *Proceedings of the 2000 IEEE 15th International Conference on Automated Software Engineering*, ASE '00, pages 229–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0710-7. URL http://dl.acm.org/citation.cfm?id=786768.786972.

[52] G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *Proceedings of the 13th International Symposium on Software*

Reliability Engineering, ISSRE '02, pages 3–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-8186-1763-3. URL http://dl.acm.org/citation.cfm?id=851033.856301.

[53] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y.

[54] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2): 14–32, 1993. ISSN 0740-7459. doi: 10.1109/52.199724. URL http://dx.doi.org/10.1109/52.199724. 4

[55] C. Wohlin and P. Runeson. Certification of software components. *IEEE Transactions on Software Engineering*, 20(6):494–499, 1994. ISSN 0098-5589. doi: 10.1109/32.295896. URL http://dx.doi.org/10.1109/32.295896. 4

[56] J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering Methodologies*, 2(1):93–106, 1993. ISSN 1049-331X. doi: 10.1145/151299.151326. URL http://doi.acm.org/10.1145/151299.151326.

[57] Matelo - product — all4tec. URL http://www.all4tec.net/index.php/All4tec/matelo-product.html.

[58] W. Dulz and F. Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Proceedings of the 3rd International Conference on Quality Software*, QSIC '03, pages 336–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4. URL http://dl.acm.org/citation.cfm?id=950789.951283.

[59] M. Beyer, W. Dulz, and Fenhua Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. In *12th Asian Test Symposium*, ATS '03, pages 102–105, November 2003. doi: 10.1109/ATS.2003.1250791.

[60] G.H. Walton, R.M. Patton, and D.J. Parsons. Usage testing of military simulation systems. In *Proceedings of the Winter Simulation Conference*, volume 1, pages 771–779, 2001. doi: 10.1109/WSC.2001.977366.

[61] A. Ost and D. van Logchem. Statistical usage testing applied to mobile network verification. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '01, pages 160–163, 2001. doi: 10.1109/ISPASS.2001.990694.

[62] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001. ISSN 0098-5589. doi: 10.1109/32.965342.

[63] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *Software Reliability Engineering, 2004 15th International Symposium on*, ISSRE '04.

[64] Y. Falcone, J.-C. Fernandez, T. Jéron, H. Marchand, and L. Mounier. More testable properties. In *Proceedings of the IFIP WG 6.1 22nd International Conference on Testing Software and Systems*, ICTSS '10, pages 30–46, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16572-9, 978-3-642-16572-6. URL http://dl.acm.org/citation.cfm?id=1928028.1928032. 4

[65] G. Durrieu, H. Waeselynck, and V. Wiels. Leto - a lustre-based test oracle for airbus critical systems. In D. D. Cofer and A. Fantechi, editors, *FMICS*, Lecture Notes in Computer Science - 5596, pages 7–22. Springer, 2008. ISBN 978-3-642-03239-4. URL http://dblp.uni-trier.de/db/conf/fmics/fmics2008.html#DurrieuWW08. 4, 33

[66] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. *SIGSOFT Software Engineering Notes*, 27(4): 70–80, 2002. ISSN 0163-5948. doi: 10.1145/566171.566183. URL http://doi.acm.org/10.1145/566171.566183. 4

[67] L. Du Bousquet, Y. Ledru, O. Maury, C. Oriat, J. l. Lanet, L. Bousquet, Y. Ledru, O. Maury, and C. Oriat. Case study in jml-based software validation. In *Proceedings of the 19th International Conference on Automated Software Engineering*. Press, 2004. 4

[68] Y. Le Traon, B. Baudry, and J.-M. Jezequel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.79. URL http://dx.doi.org/10.1109/TSE.2006.79. 4

[69] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. ISBN 0-201-33140-3. 5

[70] Test automation - silk test. URL http://www.borland.com/products/silktest/.

[71] Abbot framework for automated testing of java gui components and programs. URL http://abbot.sourceforge.net/doc/overview.shtml.

[72] A. Ruiz and Y. W. Price. Test-driven gui development with testng and abbot. *IEEE Software*, 24(3):51–57, May 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.92. URL http://dx.doi.org/10.1109/MS.2007.92.

[73] Extensible markup language (xml) 1.0 (fifth edition), 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/.

[74] Selenium - web browser automation. URL http://seleniumhq.org/.

[75] A. Sirotkin. Web application testing with selenium. *Linux Journal*, 2010(192), 2010. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1767726.1767729.

[76] A. Holmes and M. Kellogg. Automating functional tests using selenium. In *Proceedings of the 2006 Conference on Agile Software Development*, AGILE '06, pages 270–275, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2562-8. doi: 10.1109/AGILE.2006.19. URL http://dx.doi.org/10.1109/AGILE.2006.19.

[77] D. Xu, W. Xu, B. K. Bavikati, and W. E. Wong. Mining executable specifications of web applications from selenium ide tests. In *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability*, SERE '12, pages 263–272. IEEE, 2012. ISBN 978-0-7695-4742-8. URL http://dblp.uni-trier.de/db/conf/ssiri/sere2012.html#XuXBW12.

[78] Cunit home. URL http://cunit.sourceforge.net/.

[79] Sourceforge.net: cppunit. URL http://cppunit.sourceforge.net/.

[80] Junit. URL http://junit.sourceforge.net/.

[81] Pyunit - the standard unit testing framework for python. URL http://pyunit.sourceforge.net/.

[82] *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. 11, 14, 37

[83] C. Willcock, T. Deiss, and S. Tobies. *An Introduction to TTCN-3*. Wiley, Chichester, 2005. 6, 14

[84] J. Grossmann, D. Serbanescu, and I. Schieferdecker. Testing embedded real time systems with ttcn-3. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 81–90, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3601-9. doi: 10.1109/ICST.2009.37. URL http://dx.doi.org/10.1109/ICST.2009.37.

[85] Z. R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 - A Real-Time Extension for TTCN-3. Technical report, SIIM Technical Report SIIM-TR-A-01-14, Schriftenreihe der Institute für Informatik/Mathematik, Medical University of Lübeck, Germany, 23. November 2001, 2001.

[86] I. Schieferdecker, E. Bringmann, and J. Gro. 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. ISBN 1-59593-402-2. doi: 10.1145/1138474.1138481. URL http://doi.acm.org/10.1145/1138474.1138481.

[87] S. Schulz and T. Vassiliou-Gioles. Implementation of ttcn-3 test systems using the tri. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, TestCom '02, pages 425–442, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. ISBN 0-7923-7695-1. URL http://dl.acm.org/citation.cfm?id=648132.748152.

[88] Introduction to asn.1. URL http://www.itu.int/ITU-T/asn1/introduction/index.htm.

[89] Ibm - embedded software test automation framework - ibm rational test realtime - rational test realtime - software. URL http://www-01.ibm.com/software/awdtools/test/realtime/.

[90] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Software Engineering Notes*, 14(8):210–218, 1989. ISSN 0163-5948. doi: 10.1145/75309.75332. URL http://doi.acm.org/10.1145/75309.75332.

[91] M. Bennett, R. Borgen, K. Havelund, M. Ingham, and D. Wagner. Development of a prototype domain-specific language for monitor and control systems. In *2008 IEEE Aerospace Conference*, pages 1–18, March 2008. doi: 10.1109/AERO.2008.4526660.

[92] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700. 5

[93] Omg unified modeling language^{TM} (omg uml) - infrastructure, version 2.4.1, January 2012. URL http://www.omg.org/spec/UML/2.4.1/. 5

[94] Uml testing profile (utp), version 1.1. URL http://www.omg.org/spec/UTP/1.1/PDF. 5, 22

[95] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din. From u2tp models to executable tests with ttcn-3 - an approach to model driven testing. In *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, TestCom '05, pages 289–303, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-26054-4, 978-3-540-26054-7. doi: 10.1007/11430230_20. URL http://dx.doi.org/10.1007/11430230_20. 6

[96] I. Schieferdecker and G. Din. A meta-model for ttcn-3. In Manuel Nez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio, editors, *FORTE Workshops*, Lecture Notes in Computer Science - 3236, pages 366–379. Springer, 2004. ISBN 3-540-23169-2. URL http://dblp.uni-trier.de/db/conf/forte/fortew2004.html#SchieferdeckerD04. 6

[97] Ttworkbench - the reliable test automation platform, testing technologies. URL http://www.testingtech.com/products/ttworkbench.php. 6

[98] P. Baker and C. Jervis. Testing uml2.0 models using ttcn-3 and the uml2.0 testing profile. In *Proceedings of the 13th International SDL Forum Conference on Design for Dependable Systems*, SDL '07, pages 86–100, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74983-7, 978-3-540-74983-7. URL http://dl.acm.org/citation.cfm?id=1779934.1779942. 6

[99] Rational tau, ibm. URL http://www01.ibm.com/software/awdtools/tau/. 6

[100] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for gui systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 82–92, New York, NY, USA, 1998. ACM. ISBN 0-89791-971-8. doi: 10.1145/271771.271793. URL http://doi.acm.org/10.1145/271771.271793. 6

[101] Y. Yongfeng, L. Bin, Z. Deming, and J. Tongmin. On modeling approach for embedded real-time software simulation testing. *Systems Engineering and Electronics, Journal of*, 20(2):420–426, April 2009. 6

[102] C. Efkemann and J. Peleska. Model-based testing for the second generation of integrated modular avionics. In *Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 55–62, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4345-1. doi: 10.1109/ICSTW.2011.72. URL http://dx.doi.org/10.1109/ICSTW.2011.72. 6

[103] Scarlett project, . URL hhttp://www.scarlettproject.eu. 6

[104] Rt-tester 6.x product information. URL http://www.verified.de/_media/en/products/rt-tester_information.pdf. 6

[105] M. Dahlweid, O. Meyer, and J. Peleska. Automated testing with rt-tester - theoretical issues driven by practical needs. In *In Proceedings of the FM-Tools 2000, number 2000-07 in Ulmer Informatik Bericht*, 2000. 6

[106] J. Fischer, M. Piefel, and M. Scheidgen. A metamodel for sdl-2000 in the context of metamodelling ulf. In Daniel Amyot and Alan W. Williams, editors, *Proceedings of the 4th International SDL and MSC Conference on System Analysis and Modeling*, Lecture Notes in Computer Science - 3319, pages 208–223. Springer, 2004. ISBN 3-540-24561-8. URL http://dblp.uni-trier.de/db/conf/sam/sam2004.html#FischerPS04. 6

[107] J. Großmann, P. Makedonski, H.-W. Wiesbrock, J. Svacina, I. Schieferdecker, and J. Grabowski. *Model-Based X-in-the-Loop Testing*, chapter 12. CRC Press, 2011. 6

[108] A. Ott. *System Testing in the Avionics Domain*. Doctoral dissertation, Universität Bremen, 2007. Available at http://nbn-resolving.de/urn:nbn:de:gbv:46-diss000108814. 9

[109] I. Land and J. Elliott. *Architecting ARINC 664, Part 7 (AFDX) Solutions*. Xilinx, May 2009.

[110] Arinc specification 429p1-18 digital information transfer system (dits), part 1, functional description, electrical interfaces, label assignments and word formats. URL https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=1941.

[111] Mil-std-1553 digital time division command/response multiplex data bus. URL http://quicksearch.dla.mil/basic_profile.cfm?ident_number=36973&method=basic.

[112] Ieee standard c/atlas test language. *IEEE Std 716-1989*, 1989. doi: 10.1109/IEEESTD.1989.101066. 11

[113] Ieee standard for automatic test markup language (atml) for exchanging automatic test equipment and test information via xml. *IEEE Std 1671-2010 (Revision of IEEE Std 1671-2006)*, pages 1–388, 20 2011. doi: 10.1109/IEEESTD.2011.5706290. 11

[114] J. Grossmann, I. Fey, A. Krupp, M. Conrad, C. Wewetzer, and W. Mueller. Model-driven development of reliable automotive services. chapter TestML - A Test Exchange Language for Model-Based Testing of Embedded Software, pages 98–117. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-70929-9. doi: 10.1007/978-3-540-70930-5_7. URL http://dx.doi.org/10.1007/978-3-540-70930-5_7. 14, 37

[115] Xml schema 1.1. URL http://www.w3.org/XML/Schema.html.

[116] Mathworks france - matlab - le langage du calcul scientifique. URL http://www.mathworks.fr/products/matlab/.

[117] Y. Hernandez, T. M. King, J. Pava, and P. J. Clarke. A meta-model to support regression testing of web applications. In *Proceedings of the 20th International Conference on Software Engineering Knowledge Engineering, San Francisco, CA, USA, July 1-3, 2008*, SEKE '08, pages 500–505. Knowledge Systems Institute Graduate School, 2008. ISBN 1-891706-22-5.

[118] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, M. Schieber, and Y. Fusero. A meta-model for tests of avionics embedded systems. *to appear in Proceedings of MODELSWARD 2013 1st International Conference on Model-Driven Engineering and Software Development*, February 2013. 21

[119] Eclipse modeling emft - home. URL http://www.eclipse.org/modeling/emft/?project=ecoretools. 22, 24, 27

[120] Omg mof 2 xmi mapping specification, version 2.4.1, August 2011. URL http://www.omg.org/spec/XMI/2.4.1/.

[121] Graphical modeling framework. URL http://www.eclipse.org/modeling/gmp/. 22, 24, 25

[122] Graphiti home. URL http://www.eclipse.org/graphiti/. 22, 24, 25

[123] Xtext. URL http://www.eclipse.org/Xtext/. 22, 24

[124] Object constraint language, version 2.3.1, January 2011. URL http://www.omg.org/spec/OCL/2.3.1/. 22

[125] Acceleo. URL http://www.eclipse.org/acceleo/. 22, 27, 30

[126] Xpand - eclipse. URL http://www.eclipse.org/modeling/m2t/?project=xpand/. 22

[127] R. Johnson and B. Woolf. The Type Object Pattern, 1997. 25

[128] Atl. URL http://www.eclipse.org/atl/.

[129] Object management group. URL http://www.omg.org/. 5, 27

[130] Omg's metaobject facility (mof) home page. URL http://www.omg.org/mof/. 27

[131] Mof model to text transformation language (mofm2t), version 1.0, January 2008. URL http://www.omg.org/spec/MOFM2T/1.0/. 27

[132] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systemts Journal*, 45(3):621–645, 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621. URL http://dx.doi.org/10.1147/sj.453.0621. 28, 30, 31

[133] S. Zschaler and A. Rashid. Towards modular code generators using symmetric language-aware aspects. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, pages 6:1–6:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0892-2. doi: 10.1145/2068776.2068782. URL http://doi.acm.org/10.1145/2068776.2068782. 31

[134] Scade suite :: Products, . URL http://www.esterel-technologies.com/products/scade-suite/.

[135] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, M. Schieber, and Y. Fusero. Stelae - a model-driven test development environment for avionics systems. *to appear in Proceedings of ISORC 2013 : 16th IEEE Computer Society International Symposium on Object Component Service-oriented Real-time Distributed Computing*, June 2013. 33

Abstract

Le développement de tests pour les systèmes d'avioniques met en jeu une multiplicité de langages de test propriétaires, sans aucun standard émergent. Les fournisseurs de solutions de test doivent tenir compte des habitudes des différents clients, tandis que les échanges de tests entre les avionneurs et leurs équipementiers / systémiers sont entravés. Nous proposons une approche dirigée par les modèles pour s'attaquer à ces problèmes: des modèles de test sont développés et maintenus à la place du code, avec des transformations modèle-vers-code vers des langages de test cibles. Cette thèse présente trois contributions dans ce sens. La première consiste en l'analyse de quatre langages de test propriétaires actuellement déployés. Elle nous a permis d'identifier les concepts spécifiques au domaine, les meilleures pratiques, ainsi que les pièges à éviter. La deuxième contribution est la définition d'un méta-modèle en EMF Ecore, qui intègre tous les concepts identifiés et leurs relations. Le méta-modèle est la base pour construire des éditeurs de modèles de test et des motifs de génération de code. Notre troisième contribution est un démonstrateur de la façon dont ces technologies sont utilisées pour l'élaboration des tests. Il comprend des éditeurs personnalisables graphiques et textuels pour des modèles de test, ainsi que des transformations basées-motifs vers un langage du test exécutable sur une plateforme de test réelle.