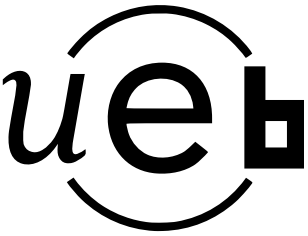
Test basé sur les modèles appliqué aux lignes de produits

Hamza Samih

ANNÉE 2014



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Hamza SAMIH

préparée à l'unité de recherche INRIA
Rennes Bretagne Atlantique

Test basé sur les modèles appliqué aux lignes de produits

Thèse soutenue à Rennes
le 5 Décembre 2014

devant le jury composé de :

Franck BARBIER
Professeur, Université de Pau / *Rapporteur*

Laurence DUCHIEN
Professeur, Université Lille 1 / *Rapporteuse*

César VIHO
Professeur, IRISA Rennes - Bretagne Atlantique / *Examinateur*

Jean-Claude ROYER
Professeur, École des Mines de Nantes / *Examinateur*

Benoit BAUDRY
Chercheur, INRIA Rennes - Bretagne Atlantique / *Directeur de thèse*

Hélène LE GUEN
Chef de projet, ALL4TEC - Laval / *Co-directrice de thèse*

Table des matières

Chapitre 1

Introduction

Notre société effectue aujourd'hui une transition de la production de masse vers la personnalisation de masse. Cette nouvelle approche vise à combiner les avantages de coût de la production de masse à l'adéquation fine des demandes d'utilisateurs. On peut déjà en voir les effets dans le secteur automobile ou encore dans la téléphonie mobile. La diversité des exigences suscitées par différents clients conduit à un développement de systèmes de plus en plus complexes comportant de nombreuses variantes. Les lignes de produits (LP) sont utilisées dans l'industrie pour parvenir à un développement logiciel plus efficace.

L'ingénierie des lignes de produits (ILP) est une approche utilisée pour développer des familles de produits. Ces produits partagent un ensemble de points communs et un ensemble de points de variation. L'ingénierie des lignes de produits permet de réduire les coûts de développement, de maintenance, d'accroître la productivité et enfin de réduire le délai de mise sur le marché en facilitant la réutilisation des artefacts logiciels, dans le but d'offrir des produits faciles à maintenir, tout en restant soucieux de leur qualité. Les *features* ou les caractéristiques, en tant qu'un aspect visible par l'utilisateur ou distinctive important d'un système logiciel, sont largement utilisées pour distinguer les différentes variantes de configurations d'une ligne de produits [KKL$^+$98, CHS08].

Aujourd'hui, la variabilité concerne également le monde des systèmes critiques. Chaque variante du système doit être testée. Toutefois, ces variantes partagent souvent des points communs et peuvent donc constituer une ligne de produits.

Ces variantes des systèmes critiques doivent se soumettre à des contraintes fortes de certification et à la réglementation des normes de la sécurité fonctionnelle, par exemple la norme générique CEI 61508 ou, plus spécifiquement dans le domaine avionique comme les normes de sécurité RTCA/DO-178C, qui exigent des activités de vérification et de validation rigoureuses pour démontrer la conformité avec les exigences de navigabilité applicables. Par conséquent, des efforts considérables sont consacrés pour tester séparément chaque variante. En raison de la forte pression sur les coûts de validations des systèmes complexes, l'industrie est obligée d'optimiser les processus de validation et de passer des tests mono-produit à des tests de lignes de produits plus efficaces.

1.1 Motivation

Les partenaires avec lesquels nous avons interagi au cours de cette thèse s'appuient, depuis de nombreuses années, sur une approche du *model-based testing* pour renforcer et systématiser la V & V (Vérification et Validation).

Le *model-based testing* (MBT) est une technique de génération automatique de suites de tests basée sur un modèle des exigences fonctionnelles, décrivant certains aspects du système sous test (SUT) [UL07]. Le *model-based testing* vise à réduire les efforts de test en générant automatiquement des cas de test à partir des modèles de test. Le modèle de test d'un système peut être représenté selon plusieurs formalismes comme un diagramme d'activité ou avec une machine à états finie. Nous nous intéressons en particulier aux modèles de test pouvant être assimilés à des chaînes de Markov appelé modèles d'usage, afin de présenter et qualifier l'usage du système.

Lors de l'intégration du MBT dans leur cycle de développement, il y a quelques années, le modèle de test capturaient les comportements attendus d'un système et était utilisé pour générer des cas de test pour ce système. Au cours des années, les évolutions du marché, des clients et du matériel ont conduit nos partenaires à spécialiser leur produit et à produire des variantes correspondant à différents besoins. Cette évolution a également nécessité une adaptation des pratiques de MBT pour intégrer ces variants dans le processus de V&V. Aujourd'hui, cette adaptation consiste à bâtir un seul modèle de test, paramétré, qui inclut tous les comportements de la famille de produits à tester. Cependant, pour valider une variante, il faut générer les cas de test correspondants à partir de ce modèle de test paramétré. Sauf que, les outils de *model-based testing* n'intègrent pas la gestion de la variabilité et ne peuvent pas générer les tests pour un produit particulier à partir d'un modèle de test d'une famille de produits. Puisque, le processus du MBT n'est effectif que pour un système unique. En particulier, le modèle d'usage n'est actuellement actualisé que pour tester des systèmes individuels

En outre, la validation est une activité disjointe du processus de développement des lignes de produits. L'effort et les moyens fournis dans les campagnes de test de chaque produit peuvent être optimisés dans un contexte plus global au niveau de la ligne de produits. Enfin, le manque de gestion explicite de la variabilité dans le modèle de test constitue la problématique centrale traitée dans cette thèse.

Afin de relever ce défi, nous proposons une approche qui relie le modèle de variabilité de la ligne de produits avec le modèle d'usage de la famille de produits. La principale raison de l'introduction de la modélisation de la variabilité dans le MBT, est la réduction des coûts lorsque les artefacts de test tels que les modèles d'usage peuvent être stratégiquement réutilisés pour différentes variantes, vu l'effort nécessaire pour réaliser un modèle d'usage pour une variante. L'amélioration de la qualité est une raison parmi d'autres, puisque les artefacts de test sont examinés et utilisés dans plusieurs variantes. Ensuite, l'idée est d'établir des correspondances formelles entre les *features*, les exigences et le modèle d'usage, afin de systématiser la dérivation des variantes de modèles d'usages – chaque variante sera exploitée par la suite pour générer les cas de test d'un produit spécifique de la ligne de produits, en réutilisant le processus traditionnel du *model-based testing*.

1.2 Contribution

Dans le cadre de nos travaux, nous voulons équiper les modèles d'usage avec la variabilité afin de documenter formellement ce qui peut varier dans un modèle d'usage. À cet effet, nous proposons de construire la relation entre le modèle de variabilité et le modèle d'usage de la ligne de produits, afin d'extraire automatiquement des variantes du modèle d'usage.

Plus précisément, nous abordons les questions de recherche suivantes : 1) Comment insuffler la variabilité dans un modèle de test et établir des relations explicites entre un modèle de variabilité et un modèle d'usage ? 2) Comment extraire une suite de tests complète et valide pour un produit spécifique à partir d'un modèle d'usage global, selon une combinaison valide de *featuress* (configurations) ?

Nous proposons trois contributions majeures pour répondre à la problématique de la validation des lignes de produits, accompagnées d'une présentation détaillée des trois projets de tests industriels qui représentent l'industrie de l'automobile et l'industrie de l'aéronautique issus de trois entreprises : Airbus Defence & Space dans le cadre du projet Européen MBAT [1] et deux projets automobiles qui concernent les tableaux de bord automobile et la configuration des véhicules.

- La *première contribution* est une approche globale pour relier un modèle de variabilité, un ensemble d'exigences fonctionnelles, et un modèle d'usage. La description de la variabilité est formellement définie, non intrusive (séparée), et peut fonctionner avec un modèle d'usage existant.

- La *deuxième contribution* est un algorithme de dérivation qui synthétise automatiquement pour une sélection de configurations, les variantes du modèle d'usage correspondantes, à partir d'un modèle d'usage global (150 % du modèle de test représentant les *features* communes et variables). Le modèle d'usage dérivé représente les scénarios de test d'une configuration sous test, cependant le modèle dérivé doit être valide et complet par rapport à la syntaxe et la sémantique définies pour le modèle d'usage MaTeLo.

- La *troisième contribution* est l'outil MaTeLo Product Line Manager (MPLM) que nous avons développé dans le cadre de la thèse pour implémenter et expérimenter notre approche avec plusieurs projets industriels. MPLM est un outil basé sur la plateforme Eclipse RCP qui étend l'outil industriel MaTeLo [2]. MPLM offre la possibilité de relier un modèle d'usage MaTeLo de la ligne de produits avec OVM le modèle de variabilité, en établissant des correspondances formelles entre les caractéristiques, les exigences et le modèle d'usage. Les relations sont ensuite exploitées pour synthétiser automatiquement les variantes de modèle d'usage pour les configurations souhaitées, extraites à partir d'un ensemble de combinaisons

1. Combined Model-based Analysis and Testing of Embedded Systems, `http://www.mbat-artemis.eu`
2. Markov Test Logic, `http://www.all4tec.net/index.php/en/model-based-testing`

de caractéristiques valides. Ensuite, les variantes du modèle d'usage dérivées sont utilisées pour générer des cas de test pour les variantes souhaitées, en se basant sur le processus de base du *model-based testing*, pour la génération des suites de tests à l'aide de l'outil MaTeLo.

Nous évaluons notre approche et MPLM dans une étude de cas industriel dans le domaine aéronautique. Les praticiens font état d'une réduction du coût pour le développement des cas de test et ils mettent en évidence la simplicité de la solution afin que les exigences et les modèles d'usage établis puissent être réutilisés. Cette expérimentation nous apprend aussi que la méthodologie de la variabilité impacte la façon dont les exigences sont spécifiées.

1.3 Exemple illustratif

Pour illustrer clairement la contribution de cette thèse, nous utilisons un échantillon d'une ligne de produits du domaine de l'automobile : un extrait simplifié du tableau de bord automobile.

Features	**Type**	**Catégorie**	**Destination**
Compteur de vitesse en Km/h	Analogique	BG	EU
Compteur de vitesse en Km/h	Numérique	HG	EU
Compteur de vitesse en mph	Analogique	BG	USA
Compteur de vitesse en mph	Numérique	HG	USA
Compte-tour moteur en tr/min	Analogique	BG / HG	EU
Compte-tour moteur en rpm	Analogique	BG / HG	USA
Thermomètre moteur (°C)	Analogique	BG / HG	EU
Thermomètre moteur (°F)	Analogique	BG / HG	USA
Thermomètre moteur	2 Voyants	BG / HG	EU/USA
Niveau carburant	Jauge	BG / HG	EU/USA
ESP	Voyant	BG / HG	EU/USA
ABS	Voyant	BG / HG	EU/USA
Écran	Monochrome	BG	EU/USA
Écran	Couleur	HG	EU/USA
Navigation	Option	HG	EU/USA

TABLE *1.1 – Tableau récapitulatif des configurations possibles du tableau de bord automobile*

Dans une voiture, le tableau de bord renseigne le conducteur sur l'état du véhicule. Il fournit diverses informations comme la vitesse du véhicule, le régime du moteur, la température du moteur, le niveau de carburant dans le réservoir, la pression de l'huile dans le moteur et la mise en marche des feux de route.

Cette liste peut varier selon le constructeur, le modèle, la version et la catégorie

du véhicule. Cependant, certaines caractéristiques d'un tableau de bord sont imposées par les standards internationaux et sont donc communes à tous les constructeurs. C'est le cas de l'indicateur de pression d'huile moteur, de l'indicateur des feux de route ou bien encore des clignotants. Nous allons nous limiter à quelques fonctions de base d'un tableau de bord avec quelques variations.

Dans cet exemple de la ligne de produits, deux catégories de configurations sont considérées : des produits haut de gamme (HG) et des produits bas de gamme (BG), destinés à la fois à l'Europe (EU) et aux États-Unis (USA). L'ensemble des fonctions que nous souhaitons vérifier et valider dans ces deux catégories sont le compteur de vitesse, le compte-tour moteur, le thermomètre moteur, les voyants, les jauges, l'écran et les options. Le tableau 1.1 liste l'ensemble des instances des points de variation de la ligne de produits exemple.

1.4 Plan

La première partie de la thèse est consacrée à l'état de l'art :

Le **chapitre 2** présente les fondamentaux de l'ingénierie des lignes de produits, ainsi que deux types de documentation de la variabilité : intégrée et orthogonale. Nous introduisons également l'*Orthogonal Variability Model*, le formalisme utilisé par nos travaux pour modéliser la variabilité de la ligne de produits.

Le **chapitre 3** présente plusieurs définitions des termes liés au test logiciel en général, suivies d'une descriptions des techniques d'analyse statique et de tests dynamiques. Ensuite, nous introduisons le *model-based testing*, une méthodologie de tests sur laquelle les travaux présentés reposent. Nous détaillons les grands principes de cette technique, ses mécanismes de génération des cas de test, ainsi que l'outil MaTeLo qui implémente les techniques du MBT.

Le **chapitre 4** présente un ensemble de travaux de recherche liés à la validation et à la vérification des lignes de produits. Nous discutons un ensemble d'approches basées sur le *model-based testing* et nous présentons un tableau comparatif des approches.

La deuxième partie de cette thèse est consacrée aux contributions scientifiques :

Le **chapitre 5** présente une extension de la définition formelle d'OVM, afin d'inclure les exigences et décrire leur relation avec les *features*. Par la suite, nous introduisons le modèle d'usage d'une ligne de produits.

Dans le **chapitre 6**, nous décrivons la mise en correspondance entre le modèle OVM, le modèle d'usage d'une ligne de produits et les exigences fonctionnelles, ensuite nous décrivons le processus de dérivation automatique des variantes du modèle d'usage.

La troisième partie est consacrée à l'implémentation et l'évaluation de l'approche :

Le **chapitre 7** présente l'implémentation des fondements théoriques de notre approche par l'outil MPLM intégré dans la plateforme de l'outil MaTeLo basée sur Eclipse RCP.

Le **chapitre 8** est consacré à l'évaluation de notre approche dans un contexte industriel, nous discutons également la mise en échelle de MPLM et l'algorithme de dérivation des variantes du modèle d'usage.

La dernière partie est consacrée à la conclusion et aux perspectives ouvertes par ces travaux de recherche.

1.5 Publication

Nous présentons la liste des publications exposées durant des conférences internationales avec comités de relecture :

- Le journal *"Étendre le test basé sur des modèles d'usage pour prendre en compte la variabilité"* [Sam14] écrit par Hamza Samih décrit la démarche globale de la dérivation des variantes du modèle d'usage avec son implémentation et son évaluation. Cet article a été publié par TSI 2014.

- L'article *"Deriving Usage Model Variants for Model-based Testing : An Industrial Case Study"* [SAB+14] écrit par Hamza Samih, Hélène Le Guen, Ralf Bogusch, Mathieu Acher et Benoit Baudry montre les résultats d'évaluation de l'approche d'adaptation du *model-based testing* pour le test de la variabilité avec une étude de cas industriel, réalisée avec Airbus Defence & Space. Cet article a été présenté à la conférence ICECCS 2014.

- L'article *"MPLM – MaTeLo Product Line Manager "* [SB14] écrit par Hamza Samih et Ralf Bogusch introduit l'outil MPLM, pour la dérivation automatique des variantes du modèle d'usage. Cet article a été présenté à la conférence SPLC 2014.

- L'article *"An Approach to Derive Usage Models Variants for Model-based Testing"* [SLB+14] écrit par Hamza Samih, Hélène Le Guen, Ralf Bogusch, Mathieu Acher et Benoit Baudry présente l'approche de dérivation des variantes du modèle d'usage. Cet article a été présenté à la conférence ICTSS 2014.

Ces travaux ont fait l'objet d'une présentation en workshop et en conférence industrielle :

- L'article *"Extension du Model-Based Testing pour la prise en compte de la variabilité dans les systèmes complexes"* [HS12] écrit par Hamza Samih, Hélène Le Guen et Benoit Baudry décrit l'approche de validation des lignes de produits, à l'aide du processus du *model-based testing*. Cet article a été présenté à la conférence GDR-GPL en 2012 en tant que poster.

- L'article *"Relating Variability Modeling and Model-Based Testing for Software Product Lines Testing"* [Sam12] écrit par Hamza Samih décrit comment étendre le *model-based testing* pour introduire la variabilité afin de générer les cas de test pour une ligne de produits. Cet article a été présenté à la conférence ICTSS 2012.

- L'article *"An Approach of Combining Model-Based Testing with Product Family Management"* [HS13] écrit par Hamza Samih et Ralf Bogusch montre les résultats d'évaluation de l'outil MPLM dans le cadre du projet européen MBAT par Airbus Defence & Space. Cet article a été présenté à la conférence UCAAT 2013.

Première partie

État de l'art

Chapitre 2

Ingénierie des lignes de produits

L'ingénierie des lignes de produits logiciels (ILP) propose de développer des familles de produits en se basant sur la réutilisation des composants. Ce chapitre présente un aperçu général des travaux autours des lignes de produits et les formalismes existants pour modéliser la variabilité. Nous détaillons également leurs grands principes et leurs particularités. Nous nous focalisons sur *Orthogonal Variability Model* qui servira de support aux activités de recherche présentées dans ce document de thèse.

2.1 Les fondamentaux de l'ingénierie des lignes de produits

Une ligne de produits (LP's) ou une famille de produits est un ensemble d'applications qui partagent une architecture commune. Les produits de la même famille se distinguent par leurs *features* qui satisfont les exigences d'un domaine particulier [CN02]. Dans l'ingénierie des lignes de produits, nous trouvons plusieurs définitions attribuées au terme *feature*. Pour Kang et al. [KCH+90] une *feature* est *"un aspect visible par l'utilisateur ou distinctive important, comme la qualité ou une feature d'un système logiciel ou des systèmes"*, c'est-à-dire, une *feature* est définie comme l'expression des exigences définies par l'utilisateur. Les *features* ont été définies différemment selon le domaine d'application. Le tableau 2.1 représente une liste non-exhaustive des différentes définitions que nous avons trouvées dans l'état de l'art et qui se rapproche de la définition de Kang.

La notion de *feature* est largement utilisée en génie logiciel, en particulier pour les lignes de produits logiciels. Les *features* sont considérées comme des abstractions de premier ordre, qui façonnent le raisonnement des ingénieurs. Cependant, cette notion semble être source de confusion, puisqu'elle mélange divers aspects des problèmes et solutions.

Classen et al. [CHS08] proposent de représenter les *features* pour l'ingénierie des exigences comme des expressions de problèmes à résoudre par les produits de la ligne de produits. En outre, comme présenté dans le tableau 2.1, il existe de nombreuses autres définitions différentes de celle de Classen. Cela est source de confusion quant à

Définition du terme *features*	Référence
Les abstractions fonctionnelles typiquement identifiables qui doivent être mises en œuvre, testées, livrées et maintenues	[KCH+90]
Une unité logique de comportement spécifié par un ensemble d'exigences fonctionnelles et non fonctionnelles	[Bos00]
D'un points de vue utilisateur, c'est les caractéristiques d'un produit qui se compose essentiellement d'un ensemble cohérent d'exigences individuelles	[CZZM05]

TABLE *2.1 – Les définitions trouvées dans l'état de l'art pour le terme feature*

ce que représente une *feature* en général. Toutefois, ces définitions ont du sens dans leur domaine respectif. Juger ces définitions et les comparer les unes aux autres, indépendamment de leur contexte ne serait pas très judicieux. Dans le cadre de nos travaux de recherche, nous avons choisi d'éclaircir ce qui signifie le terme *feature* dans notre contexte, en adoptant la définition de Kang et al. [KCH+90].

2.1.1 Framework de l'ingénierie des lignes de produits

L'ingénierie des lignes de produits logiciels est un paradigme pour développer des applications logicielles à l'aide d'une plateforme commune et de la personnalisation de masse [PBVDL05].

En ce qui concerne le processus de développement de ligne de produits, nous suivons les définitions de Pohl et al. [PBVDL05] qui présentent un *framework* consacré à l'ingénierie des lignes de produits issue des projets européen, *FAMILIES*, *café*, *ESAPS* [VdL02] puisque leur modèle de processus est devenu un quasi-standard dans la communauté de ligne de produits. Ce *framework* encourage à adopter la réutilisation plus tôt durant le cycle de vie d'une ligne de produits, pour faciliter l'élaboration d'un développement performant de la diversité des applications partageant une architecture commune. La figure 2.1 illustre le *framework* de l'ingénierie des lignes de produits, qui a été récemment adopté dans le cadre de la norme ISO/IEC #26550 *(Software and systems engineering – Reference model for product line engineering and management)*. Ce *framework* est composé de deux phases distinctes :

La première phase concerne *l'ingénierie du domaine* représentée par la partie supérieure de la figure 2.1. *L'ingénierie du domaine* permet le développement des parties communes et variables de la ligne de produits, en favorisant un développement pour la réutilisation. Ces artefacts sont la réalisation réelle de la variabilité et des points en communs de la ligne de produits. Ils représentent les modèles d'exigences, les diagrammes de classes, le code, les modèles de test et d'autres artefacts. Ce processus de développement propose de séparer la variabilité des modèles de bases en utilisant un artefact dédié, dans le but de gérer la variabilité d'une façon indépendante et d'avoir une meilleure visibilité de sa taille, d'anticiper son évolution et de rendre sa maintenance plus commode, par exemple, le modèle de variabilité OVM proposé par Polh et

al [PBVDL05].

La deuxième phase représentée par la partie inférieure de la figure 2.1 présente *l'ingénierie d'application*, qui consiste à développer des nouvelles applications de la ligne de produits en se basant sur la réutilisation des artefacts logiciels définis par l'ingénierie du domaine. En se basant sur la réutilisation, le processus exploite les liens de traçabilité entre les éléments de la variabilité et les modèles de bases afin de dériver un ensemble valide de produits de la ligne de produits répondant aux exigences du domaine d'application. En outre, une ligne de produits doit s'aligner sur les exigences communes et variables [PBVDL05]. Les exigences communes doivent être satisfaites par chaque produit de la ligne de produits et les exigences variables doivent décrire les fonctionnalités individuelles d'un produit de la ligne de produits.

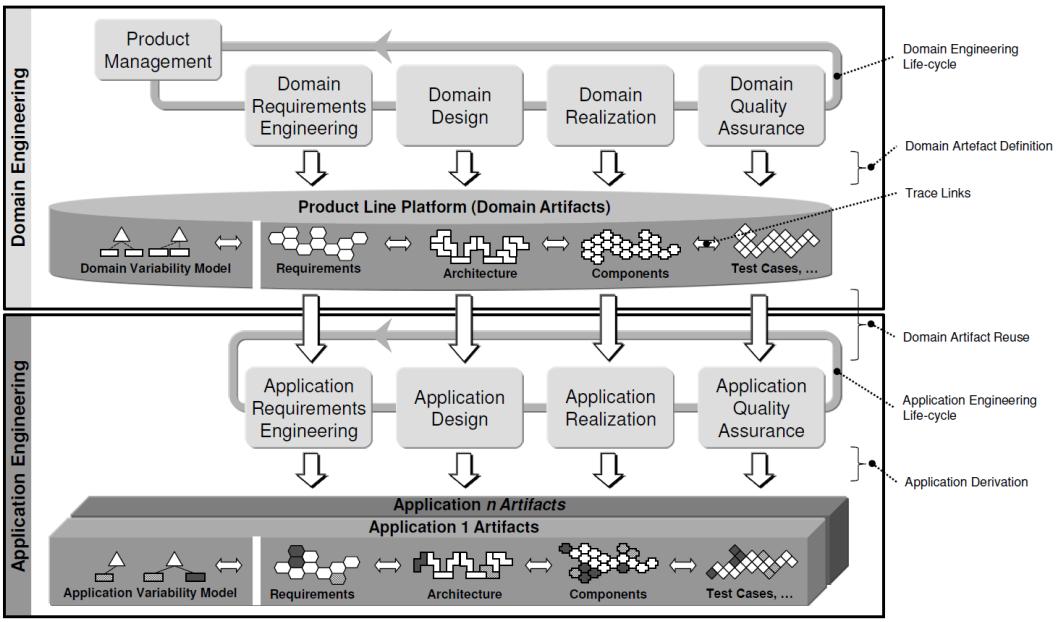


FIGURE *2.1 – Le framework de l'ingénierie des lignes de produits logiciel*

Les deux phases présentées par le *framework* permettent de définir une plateforme commune qui englobe tous les modèles de base de la ligne de produits. Les deux parties importantes de cette plateforme sont le domaine d'exigences et l'architecture de la ligne de produits selon Pohl et al. & Van der linden et al. qui l'appellent *l'architecture de référence de la ligne de produits* [LSR07].

Cette architecture de référence fournit une structure commune de haut niveau pour tous les produits de la ligne de produits. Cependant, le domaine d'exigences définit les *features* variables et communes ainsi que les exigences de qualités de la ligne de produits. Le *framework* de l'ingénierie des lignes de produits n'exige pas une séquence d'exécution des deux processus du *framework* de l'ingénierie des lignes de produits,

l'ordre d'exécution de l'activité dépend de l'organisation du processus de développement. Nous distinguons trois façons de faire :

- *Une approche proactive* qui consiste d'abord à développer toute la partie domaine avant de l'ingénierie d'application.
- *Une approche rétroactive* qui favorise le développement de tous les artefacts communs de la ligne de produits et ne commence à développer les modules variables qu'à la demande des clients.
- *Une approche de rétroinginierie* qui transforme un ensemble de produits partageant un nombre important de *features* de base en une ligne de produits.

En somme, ce *framework* sert d'architecture de base pour définir pertinemment des nouvelles applications de la ligne de produits. Cette stratégie permet de développer des produits à moindre coût et d'encourager la réutilisation.

2.1.2 Une introduction à la variabilité

La variabilité logicielle est la capacité des systèmes ou des objets logiciels à être élargis de manière efficace, modifiés, adaptés ou configurés pour une (ré-) utilisation dans un contexte particulier. La variabilité définit un ensemble de changements anticipés dans un système, par opposition à des changements dus à des erreurs constatées [GWT+14]. Svahnberg et al. [SVGB05] dénotent la variabilité logicielle comme une manière de gérer les parties d'un processus de développement logiciel et de ses artefacts résultants, mis en œuvre pour se différencier entre produits ou dans certains cas pour un seul produit.

Par contre, la variabilité de la ligne de produits doit être définie explicitement en plus de la variabilité du logiciel, car elle ne peut être identifiée uniquement en analysant la variabilité documentée par les artefacts logiciels. En outre, la variabilité de la ligne de produits décrit la variation entre les applications d'une ligne de produits logiciels en termes de propriétés, telles que les fonctionnalités proposées ou les exigences fonctionnelles ou non-fonctionnelles qui doivent être remplies [GWT+14]. Les parties communes et variables de la ligne de produits permettent de définir le champ d'application de la ligne de produits.

Une ligne de produits offre une architecture commune qu'on peut faire évoluer et spécialiser à des produits concrets. Les différences entre ces produits peuvent être discutées à l'aide du concept de *features*, par le fait que la variabilité peut être plus facilement identifiées [KCH+90], [KKL+98], [BGGB02], en représentant les *features* en tant qu'une abstraction des exigences fonctionnelles. En conséquence, les *features* qui ont tendance à varier selon les produits de la ligne de produits, peuvent être identifiées à partir des exigences fonctionnelles. Face à l'augmentation de la configurabilité des produits logiciels, la définition de la variabilité exige des décisions explicites [SVGB05]. Par ailleurs, la variabilité de ligne de produits doit être définie explicitement en plus de la variabilité logiciel. L'objectif de la modélisation de la variabilité des lignes de produits est de créer et gérer de nombreuses variantes d'un produit. Le modèle de variabilité permet de documenter et d'explorer "l'espace" des points communs et de la variabilité de la ligne de produits et son évolution potentielle.

Un autre aspect traité dans la recherche est la visibilité de la variabilité. Metzger et al. [MPH+07] distinguent entre la variabilité logicielle (cachée des clients et interne à la mise en œuvre) et la variabilité de la ligne de produits (visible pour les clients et externe, représentée par les *features*). Ils ont utilisé un modèle OVM et un modèle de *features* pour décrire les deux types de variabilité.

2.2 Modèles de variabilité

Le modèle de variabilité est un formalisme qui représente graphiquement les propriétés communes et variables de la ligne de produits. La variabilité de la ligne de produits doit être définie explicitement afin de permettre une gestion et une analyse efficace. Cependant, une ligne de produits peut être modélisée de différentes façons en fonction de différents points de vue. Principalement, on peut séparer l'espace de problème de l'espace de solution. Dans cette section nous présentons une sélection de formalismes de modélisation de la variabilité de la ligne de produits.

2.2.1 Modélisation de la variabilité de la ligne de produits

La modélisation de la variabilité est l'un des points clés dans la gestion de la variabilité, comme présentée dans la section 2.1.2. Il y a deux façons principales présentées dans la recherche pour documenter explicitement la variabilité d'une ligne de produits :

2.2.1.1 La documentation intégrée

La documentation intégrée de la variabilité consiste à introduire la modélisation dédiée à la variabilité dans les artefacts en implémentant des mécanismes comme l'héritage, *template*, *framework* qui peuvent être utilisés pour mettre en œuvre les points de variation et variantes, ou dans les langages de modélisation existants comme UML.

Ziadi et al. [ZHJ04] proposent un profil UML pour la variabilité avec des stéréotypes pour exprimer les options, les variantes et la virtualité (raffinement des concepts pour d'autres produits), cette extension concerne uniquement les diagrammes de séquence et les diagrammes de classes. Ce profil peut être utilisé pour modéliser le comportement de la ligne de produits avec des diagrammes de séquence UML.

Dans le cadre de lignes de produits, deux types de variabilité sont introduits et modélisés en utilisant les stéréotypes.

- Optionalité : les éléments optionnels d'une ligne de produits. *Optional* utilisé pour présenter "l'optionalité" au niveau des diagrammes UML. Elle peut concerner une classe, package, attribut, ou bien une opération.
- Variation : les points de variation sont modélisés grâce à l'héritage et les stéréotypes. Chaque point de variation est défini par une classe abstraite et un ensemble de sous-classes. La classe abstraite est définie avec les stéréotypes *Variation* et *Variant*.

Les choix alternatifs sont modélisés grâce à OCL (*Object Constraint Language*) qui permet de décrire les contraintes entre les points de variation et les variantes.

En outre, pour la documentation intégrée de la variabilité de la ligne de produits, les modèles de *feature* sont le plus souvent utilisés. Nous présentons dans la suite le modèle de *features*.

2.2.1.2 Les modèles de *features*

Depuis sa première présentation en 1990, le modèle des *features* [KCH+90] est la technique la plus utilisée pour modéliser les similitudes et la variabilité d'une ligne de produits [LKL02]. Les points communs et la variabilité illustrent les *features* visibles des produits dans une ligne de produits [KL13].

L'utilisation des *features* pour distinguer les produits de la ligne de produits est un concept clé sur lequel Kang et al. [KCH+90] se sont basés, pour proposer la méthode d'analyse de domaine orientée *features*, appelée *FODA* (*Feature-Oriented Domain Analysis*). Cette méthode traite les *features* en tant qu'entités de première classe au niveau des exigences et propose de les présenter avec un modèle de *features*, utilisé pour organiser les exigences d'un ensemble de produits similaires dans une ligne de produits, afin de faciliter la personnalisation des exigences. Le modèle de *features* est un arbre ou un graphe hiérarchique acyclique orienté *features*. Une *feature* peut être décomposée en sous-*features*. Les relations entre une *feature* parent et ses enfants (ou sous-*features*) sont organisées comme suit :

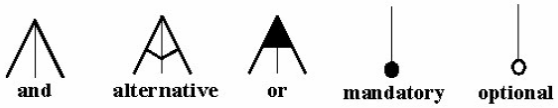


FIGURE *2.2 – La notation graphique du modèle de features*

- ***And*** — toutes les sous-*features* du parent doivent être sélectionnées dans chaque configuration.
- ***Alternative*** — seulement une des sous-*features* du parent doit être sélectionnée à la fois.
- ***Or*** — une ou plusieurs sous-*features* peuvent être sélectionnées.
- ***Mandatory*** — une *feature* doit être sélectionnée.
- ***Optional*** — une *feature* optionnelle, qui peut être sélectionnée ou non.

La figure 2.3 représente un exemple de modèle de *features* de tableaux de bord. Le modèle de *features* englobe aussi deux types de relations conceptuelles entre les *features*, appelé *contraintes*. Nous distinguons deux types de contraintes ***requires*** et ***excludes***. Dans le cas d'étude tableau de bord, la *feature navigation* ***requires*** la *feature écran couleur*, signifie que, si la *navigation* est sélectionnée, alors la *feature écran couleur* doit l'être aussi. La *feature navigation* ***excludes*** la *feature écran monochrome*, signifie que, si la *navigation* est sélectionnée, alors *l'écran monochrome* ne peut pas l'être et vice-versa.

L'objectif principal du modèle de *features* est d'identifier et de classifier les simi-

litudes et les différences dans un domaine en termes de "*features* des produits". Les résultats de l'analyse des modèles de *features* sont ensuite utilisés pour développer des artefacts réutilisables, afin de faciliter le développement des lignes de produits. Le modèle de *features* a été largement utilisé dans le monde académique [LKL02].

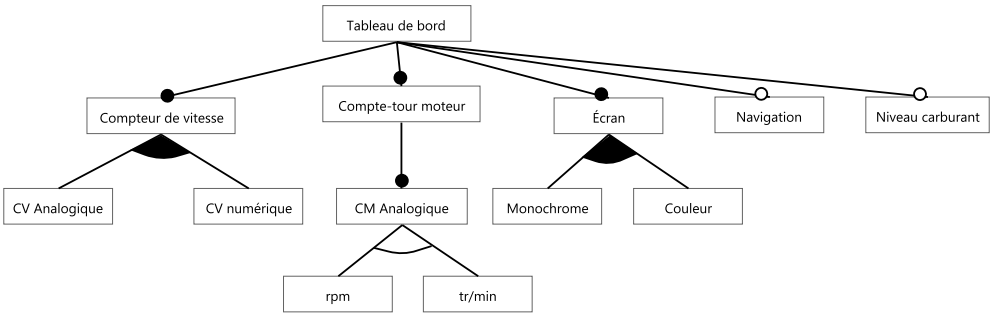


FIGURE *2.3 – Un extrait de modèle de features de tableaux de bord automobile*

Après l'introduction de FODA [KCH$^+$90], de nombreux chercheurs ont étendu le modèle de *features* en introduisant des nouvelles primitives de modélisation telles que la cardinalité des *features* et les attributs, en outre les extensions sont toujours en cours. Basés sur l'expressivité de ces extensions, les modèles de *features* peuvent être regroupés en trois catégories : les modèles de *features* offrant des fonctionnalités de base tel que présenté auparavant, les modèles de *features* avec les cardinalités (Les multiplicités UML pour la sélection des *features* [m..n]) et les modèles de *features* étendus (ajout des attributs, par exemple, pour exprimer la variation des exigences de qualité) [MP14].

Czarnacki et al. [CK05] ont proposé un modèle de *features* attribué avec de la cardinalité. Les multiplicités UML ont été utilisées dans ce modèle comme une extension de la notation originale de FODA [KCH$^+$90] pour appuyer le clonage des *features*. La principale différence avec FODA est que chaque *feature* est associée à une cardinalité qui spécifie combien de clones de la *feature* sont autorisés dans une configuration spécifique. Le clonage des *features* est utile afin de définir de multiples copies d'une partie du système qui peut être configurée différemment. En outre, les auteurs ont formalisé la notion de la cardinalité du groupe, donnant une nouvelle sémantique pour la cardinalité existante. Cette cardinalité définit le nombre minimum et le nombre maximum des membres du groupe qui peuvent être sélectionnés. Dans cette contribution, ils proposent d'utiliser OCL comme formalisme pour définir les contraintes dans les modèles de *features* avec les cardinalités. Enfin, un type d'attribut peut être spécifié pour une *feature* donnée. Ainsi, une valeur primitive de cette *feature* peut être définie pendant la configuration.

2.2.2 La documentation orthogonale

La documentation orthogonale consiste à représenter la variabilité de la ligne de produits à l'aide d'un modèle dédié. En d'autres termes, la documentation de la variabilité de la ligne de produits est séparée de la documentation des artefacts de développement

logiciel.

Ainsi, la variabilité de la ligne de produits est traitée comme un artefact de première classe de la ligne de produits. En la reliant avec les artefacts logiciels, la réalisation de la variabilité à l'intérieur des artefacts logiciels est documentée. La figure 2.4 esquisse un exemple de documentation orthogonale de la variabilité de ligne de produits et ses relations avec les artefacts de développement logiciels.

2.2.2.1 Orthogonal Variability Model

Orthogonal Variability Model (OVM) est un modèle à plat qui définit graphiquement la variabilité d'une ligne de produits logiciels. Ce formalisme introduit par Klaus Pohl [PBVDL05] permet de capturer les propriétés variables de tous les produits de la même famille. L'idée principale de l'approche OVM est de consolider l'information de la variabilité à partir de plusieurs modèles d'exigences, pour donner lieu à un seul modèle indépendant avec une vue consistante de la variabilité de la ligne de produits. L'objectif est de répondre à certaines questions concernant la variabilité telles que : quels sont les éléments qui varient ? Comment varient-ils dans une ligne de produits logiciel ?

La variabilité définit par OVM peut être reliée au code, à des modèles de conception, ou encore à des modèles de test (voir figure 2.4) [PBVDL05].

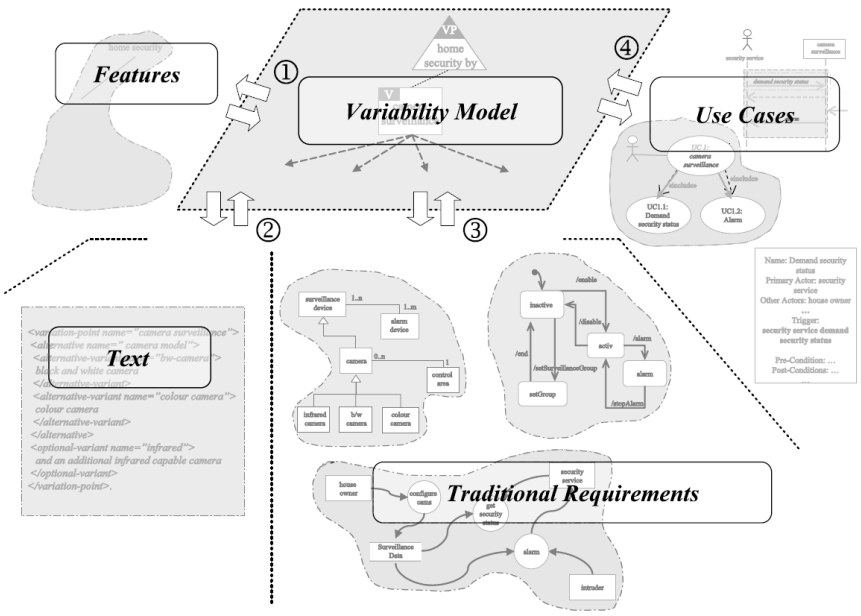


FIGURE *2.4 – Les relations entre OVM et les artefacts de développement*

2.2.2.2 Les éléments de bases d'OVM

Le modèle OVM représente les fonctionnalités variables d'une ligne de produits par des points de variation. Chaque instance d'un point de variation correspond à une variante. Si on se réfère au tableau 1.1 de l'exemple de tableaux de bord, le compteur de vitesse est un point de variation, qui peut avoir comme variantes, un compteur analogique et un compteur numérique.

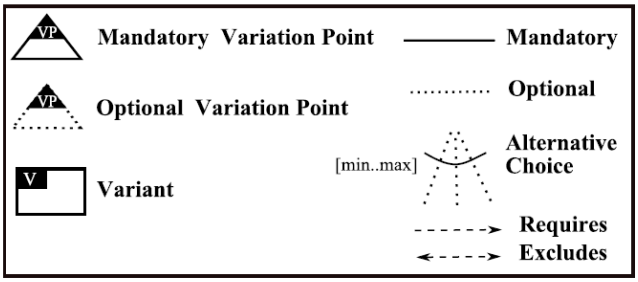


FIGURE *2.5 – La notation graphique de OVM*

Une relation de dépendance de la variabilité est une relation entre un point de variation et une variante. Un point de variation doit avoir au moins une instance et une variante doit être associée à un point de variation. Trois types de relations sont possibles : *obligatoire*, *optionnelle* et *choix alternatif* (voir figure 2.5).

- La relation de type *obligatoire*, exige la présence d'une variante dans la réalisation de la ligne de produits. Par exemple, un tableau de bord doit avoir obligatoirement un voyant de niveau carburant.
- La relation de type *optionnelle* exprime le choix d'associer ou non une variante à un point de variation dans une instance d'une ligne de produits. Par exemple, la navigation est une option qui n'est pas obligatoire dans un tableau de bord de voiture.
- La relation de type *choix alternatif* permet de regrouper les options, ce groupe d'options est associé à une cardinalité qui définit un nombre minimal n et un nombre maximal m de variantes optionnelles liées à un point de variation dans une instance ligne de produits. Cette relation est représentée par un arc de cercle couvrant l'ensemble des relations optionnelles reliant une variante avec un point de variation. Par exemple le compte-tour moteur doit absolument avoir une unité, soit tr/min pour l'Europe, soit rpm pour les États-Unis.

Les contraintes

OVM utilise aussi la notion de contrainte pour raffiner les relations entre les différents éléments du modèle de variabilité. La notion de contrainte permet de réduire au maximum le nombre de combinaisons possibles et d'exprimer les exigences et les

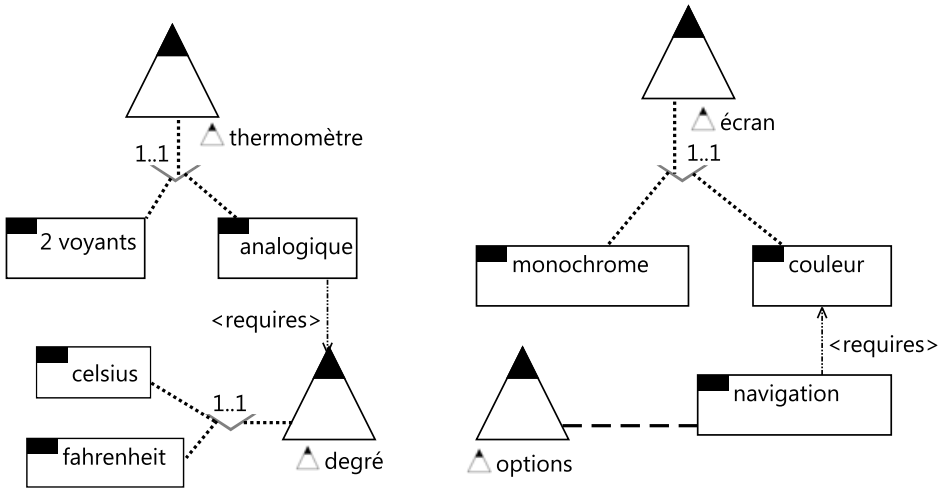


FIGURE *2.6 – Exemple du modèle OVM*

contraintes d'un système facilement.

Les contraintes peuvent être utilisées aussi pour maîtriser la différence fonctionnelle entre chaque configuration possible posée par la notion de la variabilité. La génération d'une configuration peut être automatique grâce aux règles d'inclusion ou d'exclusion et grâce aux relations entre les variantes et les points de variation qui permettent de définir un produit logiciel valide. Il existe deux types de contraintes, *requires* et *excludes*, qui sont organisées en trois classes de contraintes :

Variant_to_Variant (V_V) : Il s'agit de restreindre la liaison pour certaines sélections de variante et permettre la spécification de l'ensemble des fonctionnalités qui interagissent entre eux. Cependant, la relation de dépendance entre les points de variation est implicite.

VP (*Variation point*)_to_VP : Cette classe exprime deux types de relations entre deux points de variation, soit la nécessité de prendre en compte un autre point de variation, soit l'obligation d'exclure la prise en compte d'un autre point de variation pour réaliser le point de variation en question.

V_to_VP : Cette classe décrit la relation entre une variante et un point de variation, deux cas de figure sont possibles : une variante exige la prise en compte d'un point de variation pour être réalisée ou l'inverse, dans ce cas le point de variation avec ces dépendances ne doivent pas faire partie de la même configuration que la variante en question.

Les classes ci-dessus permettent de spécifier les liaisons possibles entre les variantes, ou bien entre les variantes et les points de variation afin de mettre en évidence les contraintes.

Le modèle OVM de la figure 2.6 représente un extrait de l'exemple illustratif de la ligne de produits de tableaux de bord. Ces éléments sont :

- Un écran qui peut être soit de type monochrome soit de type couleur.
- Deux types de thermomètre soit avec 2 voyants soit de type analogique avec affichage de température en Celsius ou en Fahrenheit.
- Les équipements optionnels comme la navigation.

2.2.2.3 Syntaxe abstraite d'OVM

Le modèle OVM fournit une vue transversale de la variabilité dans l'ensemble des artefacts de développement logiciel de la ligne de produits. Nous l'utilisons dans le cadre de nos travaux, pour la définition de la variabilité de la ligne de produits. Cependant, nous avons besoin de mieux connaitre sa structure, de réutiliser et d'étendre sa syntaxe abstraite selon nos besoins identifiés.

Pohl et a. [PBVDL05] ont introduit un méta-modèle illustré par la figure 2.7, qui décrit la structure du modèle OVM. Cette description de la structure d'OVM a permis d'explorer le formalisme et de le faire évoluer. Sur la base du méta-modèle d'OVM, Roos-Frantz et al. [RFBRC09] ont développé une méthode de transformation automatique d'un modèle de *features* en un modèle OVM, dans le but de représenter la variabilité sous différentes vues et permettre l'exploration de nouvelles approches.

Metzger et al. [MPH+07] ont montré qu'un modèle OVM peut être relié à un modèle de *features* pour permettre la navigabilité entre les deux artefacts, afin que les techniques automatisées développées dans le cadre de la modélisation des *features* puissent être réutilisées. Metzger a introduit une formalisation mathématique pour OVM, qui décrit les éléments de base en dehors du méta-modèle d'OVM. Nous la présentons ci-dessous :

Définition 2.1 *Soit un modèle OVM représenté par le tuple* $(VP, V, VG, Parent, Min, Max, Opt, Req, Excl)$*, où :*
- $VP(\neq \emptyset)$*, un ensemble de points de variation.*
- $V(\neq \emptyset)$*, un ensemble de variantes ;* $VP \cap V = \emptyset$*.*
- $VG(\neq \emptyset) \subset \mathrm{P(V)}$ *est l'ensemble de groupe de variantes* VG *partions de* V*.*
- $Parent : V \cup VG \rightarrow VP$ *retourne le parent de VP sous lequel un* V *(respectivement.* VG*) apparait.*
- $Min : VG \rightarrow \mathbb{N}$ *et* $Max : VG \rightarrow \mathbb{N} \cup \{*\}$ *retourne la cardinalité d'un* VG *donné.*
- $Opt : VP \rightarrow \mathbb{B}$ *dénote l'optionalité de* VP*.*
- $Req \subseteq (V \times V) \cup (V \times VP) \cup (VP \times VP)$ *représente un ensemble de relations de type requires entre* V_V*,* V_VP *et* VP_VP*. Excl représente la même chose, un ensemble de relations de type excludes.*

Dans la suite, nous étendons et réutilisons la syntaxe abstraite d'OVM pour notre contribution.

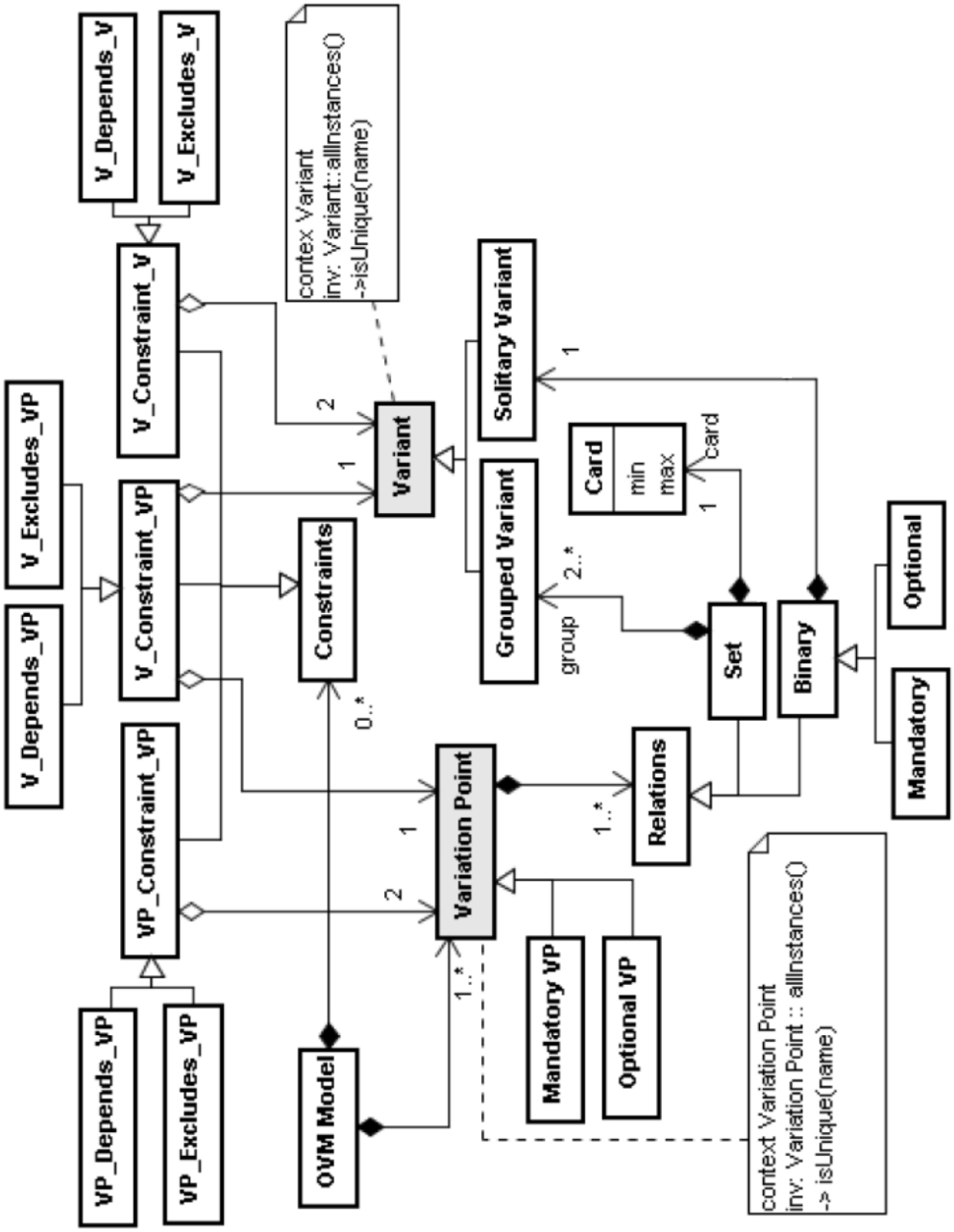


FIGURE *2.7 – Le méta-modèle d'OVM [DFJ54]*

Common Variability Language

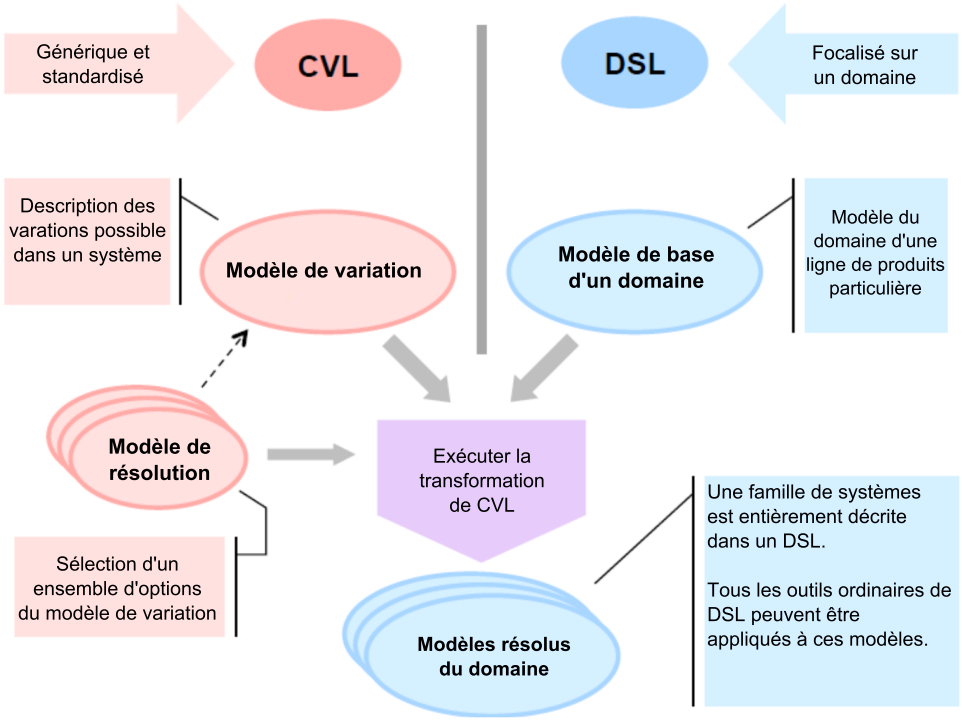


FIGURE *2.8 – L'application de CVL sur un DSL*

Common Variability Language (CVL) est un langage concret d'OVM [MP14]. Haugen et al. [HMPO+08] introduit CVL en tant qu'un langage de domaine indépendant pour la spécification et la résolution de la variabilité sur une instance de n'importe quel langage défini, en utilisant un méta-modèle basé sur MOF, tel que DSL ou UML. Son objectif est de faire un langage générique pour exprimer la variabilité à l'aide d'un mécanisme standardisé.

Comme illustré sur la figure 2.8, un modèle CVL se compose d'un modèle de variation et de plusieurs modèles de résolution appliqués à un modèle de base. Le modèle de variation définit la manière dont les éléments du modèle de base peuvent varier, sachant que plusieurs modèles de variation peuvent être définis pour un modèle de base. En outre, plusieurs modèles de résolution peuvent être associés à un modèle de variation unique, tandis que chaque modèle de résolution décrit une variante du modèle de base produite par l'exécution de la description de CVL.

2.2.2.4 Les langages textuels pour la variabilité

Une grande partie des langages pour gérer la variabilité se basent sur des éditeurs graphiques. La plupart sont des prototypes non maintenus et dans certains cas, leur syntaxe graphique constitue un fardeau pour les industriels quand il s'agit de grands modèles de variabilités. Une enquête réalisée auprès de l'industrie rapporte qu'environ un quart des industriels déclare que leurs modèles de variabilités dépassent les 10.000 unités [BRN+13]. Des chercheurs en ingénierie des lignes de produits logiciels comme Acher et al. [ACLF13](FAMILIAR) et Classen [CBH11](TVL) ont proposé des langages textuels de variabilité basés sur des éditeurs textuels ; ces dernier se justifient par leur capacité de s'adapter et de s'étendre pour inclure des nouvelles informations aux *features* et pour remplacer les formalismes graphiques quand leur syntaxe graphique ne tient pas compte des attributs ou des contraintes complexes et quand la variabilité est de taille importante.

FAMILIAR(for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) est un langage de script introduit par Acher et al. [ACLF13, ACLF11] qui vise à définir, combiner, analyser et manipuler des modèles de *features*. Les outils de FAMILIER basés sur Eclipse se composent d'un éditeur Xtext et des solveurs SAT standard utilisés en interne pour exécuter des opérations *Familiar* afin de raisonner sur les modèles de *features*.

TVL (Text-based Variability Language) a été introduit par Classen et al. [CBH11, BCFH10] pour servir de langage de référence pour la spécification des modèles de *features*. Il s'agit d'un formalisme textuel, plutôt que schématique, qui vise à être évolutif, concis, modulaire et exhaustif. Le modèle de *features* est représenté textuellement comme un arbre de *features* imbriquées, chacun avec une collection d'attributs booléens ou entiers. Il est formellement défini par une grammaire LALR, une sémantique formelle et il est fourni avec une implémentation Java disponible en ligne [CBH11].

2.2.3 Discussion

Ces différentes approches proposées pour gérer la variabilité dans l'ingénierie des lignes de produits peuvent répondre aux besoins de différents domaines. La modélisation intégrée de la variabilité au sein des artefacts logiciels augmente leur complexité, en raison de la documentation additionnelle de la variabilité de ligne de produits dans ces artefacts. En outre, la variabilité est définie de manière redondante dans différents artefacts de développement. Par conséquence, la compréhension et le traçage de la variabilité de la ligne de produits entre les différents artefacts deviennent difficiles. En outre, les dépendances entre la variabilité définie dans les modèles de base ne sont généralement pas documentées de manière explicite. Enfin, il est difficile, sinon impossible, de maintenir la variabilité définie dans les différents artefacts de base du domaine.

Cependant, séparer la variabilité des artefacts de base permet d'éviter ces trois inconvénients de la modélisation intégrée de la variabilité. Dans un modèle de variabilité orthogonal seule la variabilité d'une ligne de produits est définie. Les points communs de la ligne de produits sont documentés dans les modèles de base - une différence

importante avec le modèle de *features* (*approche traditionnelle*), qui définit à la fois, les points communs et la variabilité. La différenciation explicite entre un point de variation et une variante constitue une deuxième différence essentielle avec le modèle de *features*, qui ne fournit pas de concepts explicites de modélisation pour les points de variation. Enfin, la définition de la variabilité dans un modèle orthogonal est libre des problèmes de réalisation. Par conséquent, la modélisation orthogonale conduit à des modèles moins complexes par rapport à l'approche traditionnelle. La modélisation orthogonale de la variabilité doit être indépendante des artefacts de développement logiciel, définis dans les modèles de base comme illustrés dans la figure 2.4. Établir et maintenir des liens de traçabilité entre les modèles de variabilité et les modèles de base n'est pas anodin. Récemment, plusieurs solutions plaident en faveur d'établir des correspondances entres les *features* et les modèles de bases, comme pour CVL (voir section 2.2.2.3), qui introduisent les correspondances entre les points de variation, variantes et les modèles de base conformes-MOF. Une autre solution présentée par Classen et al. [CCS+13], qui vise à paramétrer les modèles de base pour indiquer les liens entres ses éléments et les *features*.

Les solutions pour relier la variabilité définie dans un modèle orthogonal avec les artefacts logiciels définis dans les modèles de base sont encore à leur début [MP14]. Nous devons adopter des approches intelligentes pour rendre facile d'établir et entretenir les relations entre les modèles de la variabilité et les modèles de base. Entre autres, ces approches devraient favoriser la cohérence entre les différents objets.

En outre, la modélisation de la variabilité orthogonale conduit à des modèles moins complexes par rapport à la modélisation de la variabilité intégré. Cependant, étant donné la taille et la complexité croissante des lignes de produits rencontrées dans l'industrie, les approches existantes de modélisation orthogonale de la variabilité atteindront leurs limites dans le traitement de ces lignes de produits. Dans un récent sondage, 25% des participants de l'industrie ont déclaré l'existence des lignes de produits qui comprennent plus de 10.000 *features* [BRN+13]. Enfin, peu de tentatives pour manipuler les modèles de variabilité à grande échelle, comprennent l'utilisation de langages textuels et la définition des couches d'abstraction pour la variabilité de la ligne de produits.

2.3 Résumé

La modélisation de la variabilité est mieux élaborée aujourd'hui et offre une grande variété de formalismes qui peuvent être adaptés à plusieurs contextes, cela a permis d'introduire des nouvelles approches et techniques pour pouvoir analyser la variabilité et gérer le problème de l'explosion combinatoire. Les travaux autour de la formalisation de la variabilité et des méthodes d'analyse associées [MPH+07], [RFBRC09], nous ont permis de profiter de leurs retours d'expérience et de choisir OVM dans le cadre des travaux de cette thèse. Une des questions encore ouvertes dans l'ingénierie des lignes de produits est de savoir comment garder la cohérence entre les modèles de variabilités et les modèles de bases du domaine.

Chapitre 3

Le test logiciel

Vérifier le bon fonctionnement du système dans différentes conditions d'utilisation est essentiel. Le test logiciel est employé pour détecter des anomalies. La vérification et la validation sont deux approches utilisées pour le test logiciel. Le processus de vérification se base sur la construction des modèles pour capturer certaines propriétés du système à tester, afin d'appliquer les techniques d'analyses pour vérifier les propriétés capturées et détecter les défauts. Le processus de validation logiciel consiste à choisir une ou plusieurs stratégies de tests, cette stratégie est utilisée pour définir les critères de sélection des cas de test à générer.

Dans ce chapitre, nous énumérons d'abord plusieurs définitions et termes liés au test logiciel en général, puis nous livrons une brève étude sur les techniques de test existantes. Enfin, nous présentons le *Model-based Testing*, une méthodologie de test qui joue un rôle majeur dans la contribution de cette thèse.

3.1 Termes et définitions

Les méthodes d'assurance qualité peuvent être classées en trois catégories : vérification, validation et test. La vérification et la validation sont deux termes de test fréquemment utilisés mais souvent confondus.

Définition 3.1 ***(Vérification)*** *:*
Un procédé d'évaluation d'un système ou d'un composant de système, pour déterminer si les produits d'une phase d'élaboration donnée remplissent les conditions imposées au début de cette phase [Sep90].

Définition 3.2 ***(Validation)*** *:*
Un procédé d'évaluation d'un système ou d'un composant de système au cours ou à la fin du processus de développement, afin de déterminer s'il satisfait les exigences spécifiées [Sep90].

Définition 3.3 ***(Test)*** *:*

Une activité dans laquelle un système ou composant est exécuté dans des conditions spécifiques, les résultats sont observés ou enregistrés et une évaluation est faite pour certains aspects du système ou d'un composant [Sep90].

De manière générale, le test est un processus qui exécute un programme ou un système dans le but de trouver des défauts [WM10]. Les défauts ou bugs dans un composant logiciel produisent des erreurs. Le test est utilisé aussi pour identifier les défaillances du système, produites lorsque le service fourni n'est plus conforme aux spécifications.

Définition 3.4 ***(Défaut, Bug)*** *:*
Une faille dans un composant ou d'un système qui entraîner une défaillance du composant ou système pendant l'exécution, par exemple, une déclaration inexacte ou la définition de données [SLS11].

Définition 3.5 ***(Erreur)*** *:*
Une action humaine qui produit un résultat incorrect [Sep90].

Définition 3.6 ***(Défaillance)*** *:*
Un écart réel entre le résultat ou le comportement obtenu et le comportement attendu du système ou d'un composant [SLS11].

Nous tenons à préciser les notions suivantes, qui sont utilisées à plusieurs reprises dans cette thèse.

Définition 3.7 ***(Cas de test)*** *: Un ensemble de valeurs d'entrée, les conditions préalables d'exécution, les résultats attendus et les postconditions d'exécution développées pour un objectif particulier ou une condition de test, de manière à exercer un chemin de programme particulier ou pour vérifier la conformité d'une disposition spécifique [Sep90].*

Définition 3.8 ***(Suite de test)*** *:*
Un ensemble de plusieurs cas de test pour un composant ou système sous test, où la postcondition d'un cas de test est souvent utilisée comme une condition préalable au suivant [SLS11].

Définition 3.9 ***(Script de test)*** *:*
Généralement utilisé pour désigner une description de la procédure de test, en particulier un système automatisé [SLS11].

Définition 3.10 ***(Résultat attendu)*** *:*
Le comportement prévu par le cahier des charges, ou par autre source, d'un composant ou d'un système dans des conditions spécifiées [SLS11].

3.2 Techniques de tests

Les tests fonctionnels se réfèrent aux activités vérifiant une action ou une fonction spécifique d'un système. En général, les exigences fonctionnelles sont utilisées comme référence, bien que certaines méthodologies de développement travaillent à partir de cas d'utilisation ou de l'expérience des utilisateurs. Les tests fonctionnels ont tendance à répondre à des questions du genre, est-ce que l'utilisateur peut faire ceci ? Ou cette fonction spécifique se comporte-t-elle correctement ?

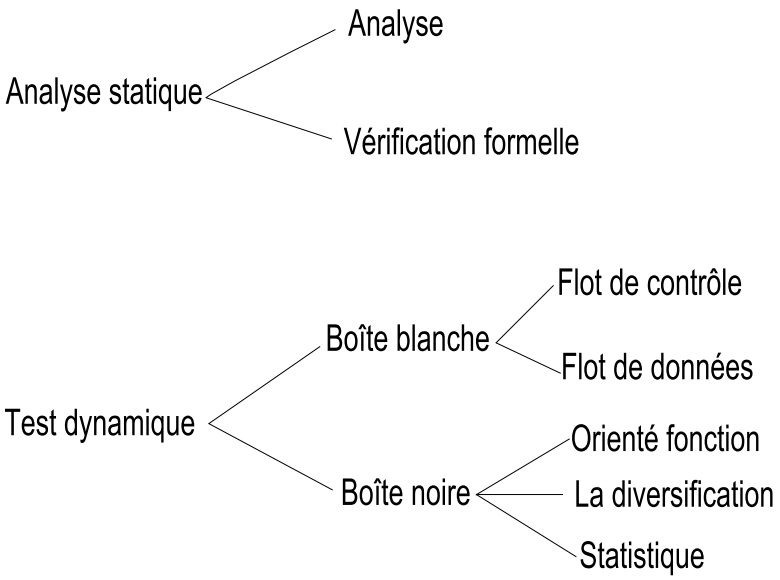


FIGURE *3.1 – Vue d'ensemble des techniques de tests*

La figure 3.1 présente une vue d'ensemble des techniques d'assurance qualité logicielle existante. Dans cette section, nous présentons la différence entre deux types de tests fonctionnels, l'analyse statique et le test dynamique.

3.2.1 L'analyse statique

Le terme analyse statique se réfère à l'analyse qui opère au moment de la compilation et se rapproche de l'ensemble des valeurs ou des comportements qui découlent dynamiquement à l'exécution du programme [Nie10]. L'analyse statique peut être appliquée pour analyser le flot des données, mais aussi l'analyse d'alias, programme de *slicing* et pour effectuer des analyses basées sur les contraintes [Muc97, Wei81, Nie10]. Il peut également dériver des mesures ou métriques, afin de mesurer et prouver la qualité de l'objet sous test. Pour contourner le problème de l'indétermination de fin de programmes, l'analyse statique utilise l'approximation comme une technique clé pour éviter de tomber dans le piège des boucles et de la récursivité [Nie10].

À l'origine, l'analyse statique a été utilisée pour optimiser les compilateurs [Muc97,

Nie10] et pour déboguer le code des programmes [Wei81]. Récemment, l'analyse statique a été utilisée pour vérifier des programmes, dans le but de découvrir des bugs, elle est aussi appelée *test statique* [Nie10]. Par exemple, l'analyse statique est en mesure de trouver des accès à des régions ou des variables de mémoire non initialisées. Certains outils d'analyse statique opèrent sur le code source (par exemple, *LINT* pour C [Dar91]), d'autres sur le byte-code Java (par exemple, *Findbugs* [HP04]) et aussi sur les modèles de *features* (*PACOGEN*) [HBG11].

Les outils de l'analyse et test statique sont soit intégrés dans les compilateurs tels que *CLANG* soit mis en œuvre sous la forme d'outils dédiés tels que *FIND BUGS* [HP04] et *PACOGEN* [HBG11]. Similaire aux techniques de *modèle checking*, l'analyse statique fonctionne automatiquement et ne nécessite pas l'intervention de l'utilisateur.

L'utilisation du test statique ne permet pas de trouver tous les bugs. Cependant, certains bugs apparaissent uniquement lorsque le programme est exécuté, c'est-à-dire au moment de l'exécution et ne peuvent pas être reconnus avant. Par exemple, si la valeur du dénominateur d'une division est mémorisée dans une variable, la valeur zéro peut être affectée à cette variable. Cela conduit à une défaillance pendant l'exécution. En analyse statique, ce bug ne peut pas être facilement trouvé, sauf lorsque cette variable reçoit la valeur zéro par une constante ayant zéro comme valeur. Autrement, tous les chemins possibles de l'opération menée sont analysés. En cas de défaut, l'opération peut être signalée comme potentiellement dangereuse.

3.2.2 Test dynamique

Le test dynamique est considéré comme un processus basé sur les cas de test pour détecter les erreurs dans un programme dont la présence est supposée. Un cas de test réussi est celui qui favorise les progrès dans cette direction, en amenant le programme à l'échec. Le test est utilisé éventuellement pour établir un certain degré de confiance, pour qu'un programme fasse ce qu'il est censé faire et non l'inverse. Cependant, afin de déterminer s'il est possible de tester un programme pour trouver l'ensemble de ses erreurs, il faut établir des stratégies de test.

Le test dynamique est une solution à un problème commun dans les programmes informatiques. Bien que le système soit en cours d'exécution, une panne peut survenir à tout moment, sans que l'utilisateur sache l'origine du problème. Nous ne savons pas si une défaillance que nous ne percevons pas immédiatement causera des problèmes, mais elle peut causer toujours un dysfonctionnement du système. Les défaillances peuvent être des données corrompues, un ralentissement général du système, ou un écran bleu, il y a beaucoup de symptômes éventuels possibles. Quelles sont les causes de ces défaillances ? Il y a probablement un nombre incalculable de défaillances différentes qui pourraient être causées par de petites erreurs. Par exemple, le problème de fuites de mémoire où un développeur oublie de désallouer un petit bout de la mémoire dans une fonction qui est exécutée des centaines de fois par minute. Chaque fuite est petite, mais la somme totale peut causer un crash du programme lorsque le système est à court de RAM.

Le système sous test doit être exécutable. Il est muni de données d'entrée avant qu'il

ne soit exécuté. Dans les étapes de test antérieures dans le cycle de développement (tests unitaires et tests d'intégration), l'objet sous test ne peut pas être exécuté tout seul, mais doit être intégré dans un banc de test pour obtenir un programme exécutable (voir la figure 3.2).

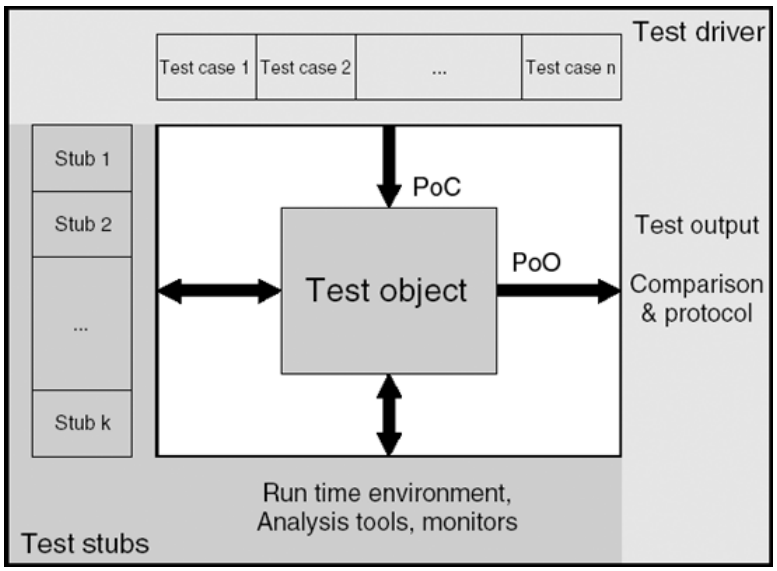


FIGURE *3.2 – Banc de test*

Il existe plusieurs techniques de validation pour spécifier les cas de test et pour définir les critères d'arrêt des tests. Ces techniques sont divisées en tests de boîte noire et boîte blanche.

3.2.2.1 Test boîte noire

Le test boîte noire dit test fonctionnel, est une approche qui consiste à visualiser le programme comme étant une boîte noire. Le but est d'être complètement indifférent au comportement et à la structure interne du programme. Pour se concentrer sur la recherche des conditions dans lesquelles le programme ne se comporte pas conformément à sa spécification. Dans cette approche, les données de test sont issues exclusivement des spécifications, c'est-à-dire sans tirer parti de la connaissance de la structure interne du programme [MSB11, Lew04].

Cependant, pour utiliser cette approche afin de trouver toutes les erreurs dans le programme, le critère est le test exhaustif des entrées, faisant usage de toutes les entrées possibles sous forme de cas de test. Sauf que les tests d'entrée exhaustifs sont impossibles, car ne nous pouvons pas tester un système visant à garantir qu'il est sans bugs, sans oublier le coût important des tests. Un avantage majeur de l'approche boite noire est que les tests sont axés sur ce que le système est censé faire, ce qui est naturel et compréhensible par tout le monde. Plusieurs techniques de test boîte noire existent,

quelques-unes d'entre elles sont présentées dans cette section.

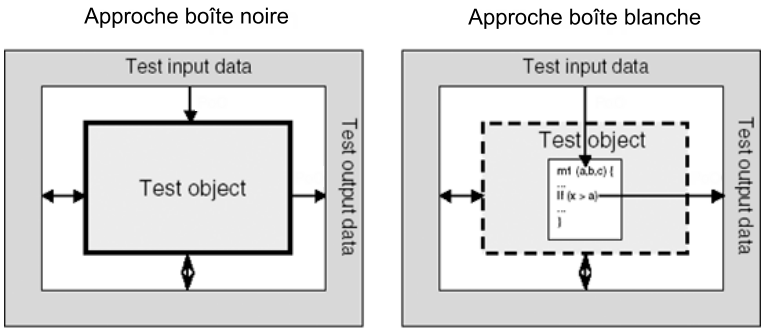


FIGURE *3.3 – Tests boîte noire et boîte blanche*

3.2.2.2 Test boîte blanche

Le test boîte blanche appelé aussi test structurel est une stratégie de test basée sur un graphe de contrôle (un programme), qui permet d'examiner la structure interne du système. Cette stratégie découle les données de test à partir d'un examen de la logique du code de système. L'objectif à ce stade est d'établir pour cette stratégie, une analogie aux tests d'entrée exhaustifs (flot de données) dans l'approche boîte noire. En outre, provoquer chaque instruction dans le programme (chemin de contrôle) à exécuter au moins une fois pourrait sembler être la solution, mais il n'est pas difficile de montrer que cela est très insuffisant.

En général, l'analogue est considéré en tant que chemin de test exhaustif. Néanmoins, si on exécute, avec des cas de test, tous les chemins possibles de flot de contrôle à travers le programme, on peut dire que le programme a été complètement testé. Un avantage de tests boîte blanche est la grande possibilité de détecter les anomalies dans la structure interne du programme, la logique, les erreurs et les sottises délibérées de la part d'un développeur [MSB11, Lew04]. Par contre, l'application de ces tests se limite aux tests unitaires en général, car la création des cas de test basés sur la structure du code source devient vite complexe devant un nombre important de lignes de code du système sous test.

3.3 Model-based Testing

Le *model-based testing* (MBT) est une variante des techniques de test qui se base sur des modèles de comportement explicites, décrivant les comportements attendus du système sous test (SUT), ou le comportement de son environnement, construits à partir

des exigences fonctionnelles. Le MBT est une approche en évolution qui vise à générer automatiquement à partir de l'un de ces modèles, des cas de test à jouer sur le SUT [UL07]. En outre, le MBT se positionne dans le cycle en V de développement, entre la phase de spécification et la phase de validation (voir figure 3.4). Cependant, l'idée derrière la modélisation explicite du comportement attendu du SUT et éventuellement les comportements de son environnement est de contribuer à atténuer les problèmes de gestion des tests, c'est-à-dire avoir des cas de test bien documentés, facile à reproduire et à maintenir. Traditionnellement la gestion manuelle des cas de test n'est pas évidente [UPL12].

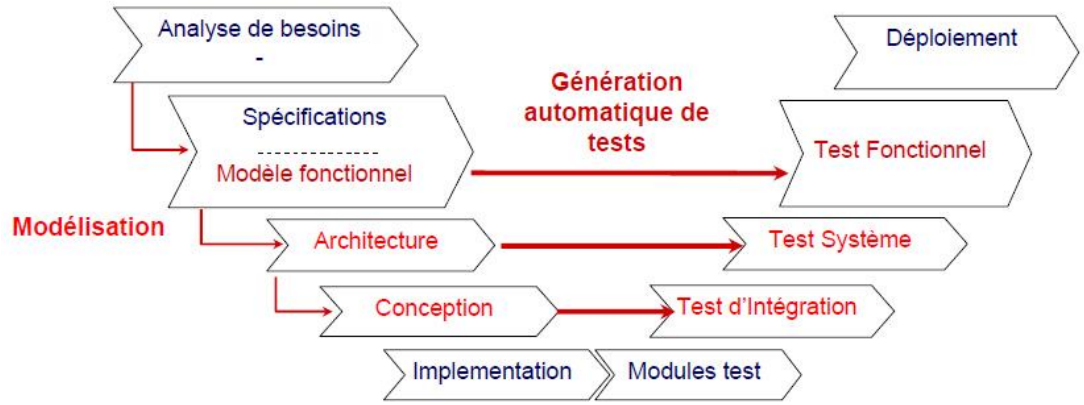


FIGURE *3.4 – Cycle en V*

3.3.1 Avantages économiques du MBT

L'intérêt d'adopter le *model-based testing* dans les projets est d'améliorer la détection des bugs du SUT, réduire le coût et le temps de la phase de tests, améliorer la qualité logicielle, la traçabilité et l'évolution des exigences.

Le *model-based testing* peut amener à réduire le temps et les efforts consacrés à tester si le temps nécessaire pour écrire et maintenir le modèle ainsi que le temps consacré à diriger la génération de test est inférieure au temps de la conception et le maintien d'une suite de test manuellement. Dans ce cas le processus du *model-based testing* est rentable, car la génération des scripts de test est automatisée, et il rend plus facile la gestion de l'évolution des exigences, en ne modifiant que le modèle et en régénérant les cas de test, plutôt qu'en maintenant la suite de test en-soi. Cela permet de réduire considérablement le coût de la maintenance de test.

Avant d'adopter le MBT, il est souhaitable d'avoir un processus de test assez mature et une certaine expérience avec l'exécution automatique de test. Le MBT a été utilisé avec succès sur de nombreux projets industriels et la plupart des études [DJK+99, FHP02, BBN04, BLLP04, SMJU10] ont montré qu'il était un moyen efficace de détection des bugs du système sous test, ainsi qu'une technique rentable. D'autre part,

utiliser le *model-based testing* pour tester des lignes de produits peut être aussi efficace dans ce contexte, pour but de bénéficier de la maturité de cette approche et de son succès auprès des industriels.

3.3.2 Processus MBT

Le MBT adopte un processus assez générique illustré par la figure 3.5.

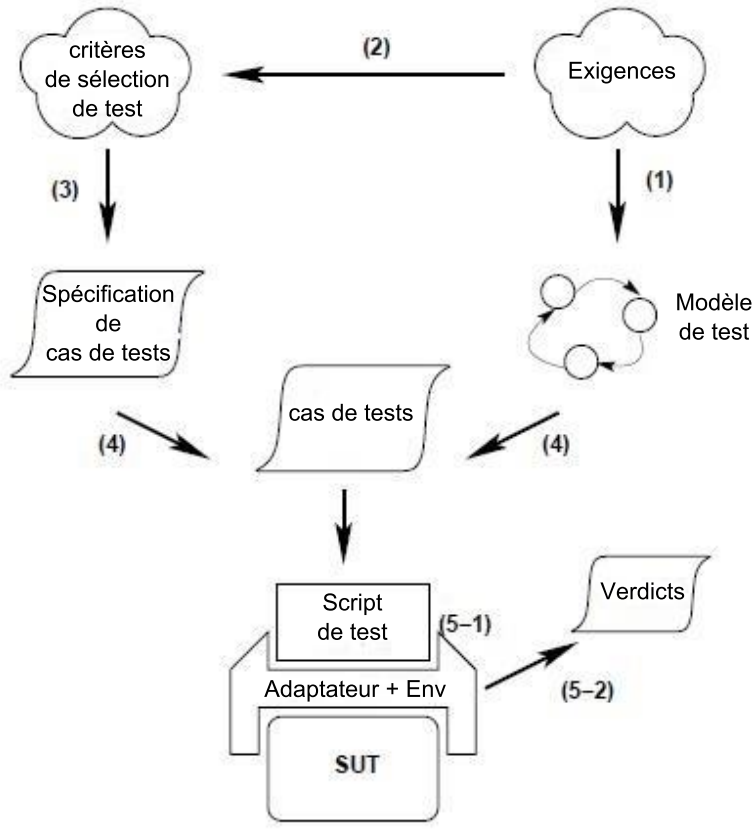


FIGURE *3.5 – Processus de model-based testing*

La première étape (1) du MBT consiste à construire un modèle abstrait appelé modèle de test, à partir des exigences fonctionnelles ou un document de spécification d'un système logiciel. La validation du modèle signifie que les exigences du système sous test sont examinées minutieusement afin de vérifier la consistance, la complétude et ce qui souvent expose les exigences aux erreurs. D'autre part, le modèle doit être suffisamment précis pour servir à la génération automatique des cas de test efficaces. Cela veut dire que la génération des tests doit être complète en terme d'actions, données d'entrées et les résultats attendus, pour avoir une véritable valeur ajoutée.

La deuxième étape (2) du processus global consiste à choisir les critères de sélection des cas de test, pour effectuer une génération automatique. Les méthodes de tests largement utilisées en industrie, comme les directives d'ISTQB [BM11], fournissant un processus de validation générique et personnalisable, qui vise à définir une politique de tests claire et objective pour la validation des SUT. En général, ces méthodes formalisent cette politique dans un document plan de test, qui définit la portée et les différentes techniques de tests utilisées dans les phases de validation d'un projet de développement (test d'intégration, test unitaire, test de système, etc.).

Les critères de sélection de tests peuvent concerner une fonctionnalité spécifique appelée *requirement-based selection critera*, ou la structure du modèle de test, par exemple des critères liés à la couverture des états, la couverture des transitions ou des techniques de couverture de données (*pairwise*, valeurs limites). Les critères de sélection peuvent aussi être de caractère aléatoire, comme les propriétés d'environnement. Dans certains cas, ils peuvent être reliés à un ensemble bien défini de défauts. Par la suite, la troisième étape (3) transforme les critères de sélection en des spécifications de cas de test qui se chargent de les formaliser pour les rendre opérationnels. Une spécification de cas de test est une description de haut niveau d'un cas de test souhaité.

La quatrième étape (4) concerne de la génération des cas de test, après la construction du modèle de test et la définition des spécifications des cas de test. L'algorithme de génération automatique peut être confronté à deux situations lors de la dérivation des cas de test. Première situation, les critères de sélection ne sont pas satisfaits, il n'y a donc pas de génération possible. La deuxième situation, plusieurs cas de test sont générés qui peuvent satisfaire les critères de sélection. L'algorithme de génération sélectionne un seul cas de test parmi la sélection opérée pour chaque critère.

La cinquième étape (5) identifie deux moyens d'exécuter une suite de test. L'exécution des tests peut être manuelle à l'aide d'une personne physique, ou automatique via un environnement dédié (banc de test), qui fournit des facilités pour exécuter automatiquement les cas de test et pour enregistrer les verdicts. Cependant, l'exécution débute par la concrétisation des entrées de tests et par l'envoi de ces données concrètes au système sous test afin de le stimuler et capturer par la suite les résultats attendus, pour les comparer avec les résultats attendus (verdicts de test).

Un cas de test peut être réalisé aussi sous forme du code exécutable appelé script de test, capable d'exécuter le cas de test avec ces entrées sur un banc de test et de calculer le verdict.

En résumé, le processus du *model-based testing* se base sur la réalisation d'un modèle de test (1) la définition d'une stratégie de test (2,3) pour générer des cas de test (4) qui peuvent être complétés par les vérifications à réaliser (si celles-ci ne sont pas incluses dans le modèle), accompagnées par une traduction éventuelle de cas de test pour une exécution automatique (5-1) et achevées par une exécution et un verdict sur les résultats issus des tests [UL07].

3.3.3 Aspects modélisation

Plusieurs formalismes ont été utilisés pour décrire un modèle, les machines à états, les systèmes de transitions étiquetées, le diagramme d'activité UML [OG09], les réseaux de Petri, les chaînes de Markov (pour les modèles probabilistes) [LGT03], etc. En outre, chaque modèle offre des possibilités différentes concernant (par exemple) la prise en compte et la résolution du non-déterminisme (conditions, probabilités, etc.), la manipulation des entrées, des sorties et des variables. Tous ces modèles sont complémentaires et offrent toutes les possibilités requises pour la modélisation et la génération de cas de test automatiques [UL07]. Pour la correction du modèle, plusieurs solutions sont possibles, comme la simulation, l'exploration, l'animation, le *modèle checking*, analyse statique et les preuves [UPL12].

Nous nous intéressons dans ce travail aux types de modèles assimilés à des états - transitions. Cette technique modélise les états du SUT, les transitions entre ces états, les actions à l'origine des transitions et les actions pouvant résulter d'une transition. Pendant la conception des cas de test, les exigences fonctionnelles fournissent l'information état - transitions, parfois cette information peut être aussi tirée des artefacts de conception qui prennent en charge la notation état-transition, comme les diagrammes d'activité dans UML [BR97], le modèle d'usage [LG05].

Les cas de test sont conçus pour opérer les transitions entre les états et spécifier :
- L'état initial du système sous test,
- Les entrées du SUT,
- Les résultats attendus du SUT,
- L'état final du SUT.

Aussi bien qu'être utile pour le test boîte noire, le processus peut dériver des tests fonctionnels qui couvrent les problèmes soit négligés soit mal spécifiés dans le cahier des charges.

Un certain nombre de méthodes graphiques (standardisé ou propriétaire) peuvent être utilisées pour soutenir cette technique qui consistent à visualiser les états et leur transition.

3.4 Outils de Model-based Testing

Un certain nombre d'outils de *model-based testing* existent. Nous identifions trois différentes catégories : outils commerciaux, propriétaires et académiques. Nous avons choisi de présenter MaTeLo, JSXM, Qtronic et Microsoft SpecExplorer. Ils ont été choisis en fonction de leur maturité.

3.4.1 MaTeLo

MaTeLo est l'acronyme de *Markov Test Logic* développé par ALL4TEC. C'est un outil commercial de génération automatique de tests fonctionnels et de validation, basé sur l'approche de l'ingénierie dirigée par les modèles. La construction des modèles de

test, la génération automatique des cas de test et des données de test ainsi que l'analyse de la campagne de test sont les trois fonctionnalités majeures proposées par l'outil.

MaTeLo a son propre modeleur basé sur les chaînes de Markov comme paradigme de modélisation pour représenter le comportement attendu du système sous test. La sélection des cas de test est orientée par les probabilités associées aux transitions du modèle de test. MaTeLo génère des cas de test avec des données d'entrée et de sortie. Les cas de test générés peuvent être exportés en script sous plusieurs formats selon le banc de test connecté.

3.4.2 JSXM

JSXM [DBI12] est un outil académique, développé par l'université de *Sheffield* sous Java, qui permet la spécification de modèles JSXM, leur animation et la génération automatique des cas de test.

Les modèles JSXM sont un type particulier de machines finies d'états étendues, appelé Flux X-Machines (SXMS). SXMS permet de décrire à la fois le contrôle et les données d'un système. Le langage de modélisation JSXM est un langage basé sur XML avec Java Code en ligne. L'entrée et les symboles de sortie sont également décrits dans un code XML.

L'animation du modèle consiste à exécuter le modèle en fournissant un flux d'entrée et en observant le flot de sortie résultant. L'animation interactive ou par lots, sont les deux types d'animation pris en charge par l'outil, elles permettent au concepteur de valider la spécification, c'est-à-dire vérifier que la fonctionnalité sous test est correctement modélisée. Une fois que le modèle est validé, il peut être utilisé pour l'algorithme de génération automatique des cas de test. Les cas de test qui sont générés par JSXM sont disponibles en format XML et ils sont indépendants de la technologie ou le langage de programmation de l'application. Ces cas de test général peuvent ensuite être transformés par l'outil JSXM à des cas de test aux langages de programmation du système sous test.

3.4.3 Qtronic

Qtronic [Hui07] est un outil commercial de génération de test basé sur un modèle développé par Conformiq. Le modèle de conception peut être exprimé comme un ensemble de fichiers textes et/ ou des modèles graphiques. La notation textuelle est définie par la Qtronic Modeling Language (QML). QML signifie un diagramme *Statecharts* étendu avec du code Java ou C#. Le langage d'action d'un diagramme d'états UML tels que des événements, des actions, des conditions de garde et d'autres contraintes spécifiques sur la transition sont décrites en utilisant QML.

L'outil utilise essentiellement le concept de l'exécution symbolique du modèle pour la génération de cas de test. L'outil peut traduire les cas de test générés à n'importe quel format exécutable. Qtronic prend en charge les modèles *multi-thread* concurrents, et support également le test des systèmes non-déterministes en mode en ligne.

3.4.4 SpecExplorer

Microsoft SpecExplorer [VCG+08] est un outil de test basé sur un modèle développé à l'origine chez *Microsoft Research* et maintenant intégré dans la plate-forme de développement *Visual Studio*. Il étend l'environnement de programmation pour modéliser le comportement des logiciels, analyser et visualiser graphiquement le comportement modélisé et la génération automatique des suites de test.

SpecExplorer prend en entrée un jeu d'ensembles de modèles *.NET* créés à partir du modèle de programme et un ensemble de scripts de coordination, pour configurer l'exploration du modèle et de test ainsi que la composition des scénarios. Les scénarios sont utilisés pour limiter le comportement général du programme du modèle à une tranche qui est testable. *SpecExplorer* propose plusieurs stratégies pour gérer l'exploration du modèle, y compris la couverture des valeurs de paramètres des données et l'espace d'états du modèle, ainsi que les critères structurels du modèle tels que la couverture de toutes les transitions. Les cas de test sont générés comme une collection de fichiers *C#* qui peuvent être consommés par n'importe quel framework *.NET* de test.

3.5 MaTeLo

Dans ce travail, nous utilisons le modèle de test de l'outil MaTeLo. D'abord, ce choix est lié au contexte de la thèse et aussi à la problématique de test des lignes de produits à laquelle sont confrontés les utilisateurs de l'outil. De plus, MaTeLo nous offre une plateforme de test et de développement ainsi que des projets pilotes et des partenaires avec qui nous avons eu des collaborations pour évaluer notre approche.

3.5.1 Modèle d'usage de MaTeLo

Le modèle de test MaTeLo est appelé *modèle d'usage*. Ce modèle d'usage est axé sur la modélisation de l'utilisation du système que l'on considère comme une boîte noire. Le modèle d'usage est un diagramme état - transition probabiliste, pouvant être assimilé à des chaînes de Markov étendues [Whi92, LM02, LG05]. Le modèle d'usage est réalisé manuellement sous MaTeLo Editor.

Le modèle d'usage est un modèle hiérarchique, c'est-à-dire qu'il peut être composé de plusieurs chaînes de Markov étendues. Une chaîne est une partie de modèle d'usage référencée par un état d'une autre chaîne au niveau supérieur. Une instance de chaîne est une chaîne référencée plusieurs fois dans le modèle d'usage par d'autres états de d'autres chaînes au niveau supérieur. Chaque transition porte une probabilité $p_{ss'}(\mathfrak{F})$, qui correspond à la probabilité de choisir l'état s' quand le processus se trouve dans l'état s pour le profil $\mathfrak{F}$. Il est possible d'associer plusieurs profils de probabilités au modèle d'usage. Le profil permet également de définir la distribution au niveau de la stimulation associée à la transition. Les probabilités ne sont pas en fonction du temps et ne varient pas pendant la génération [LG05].

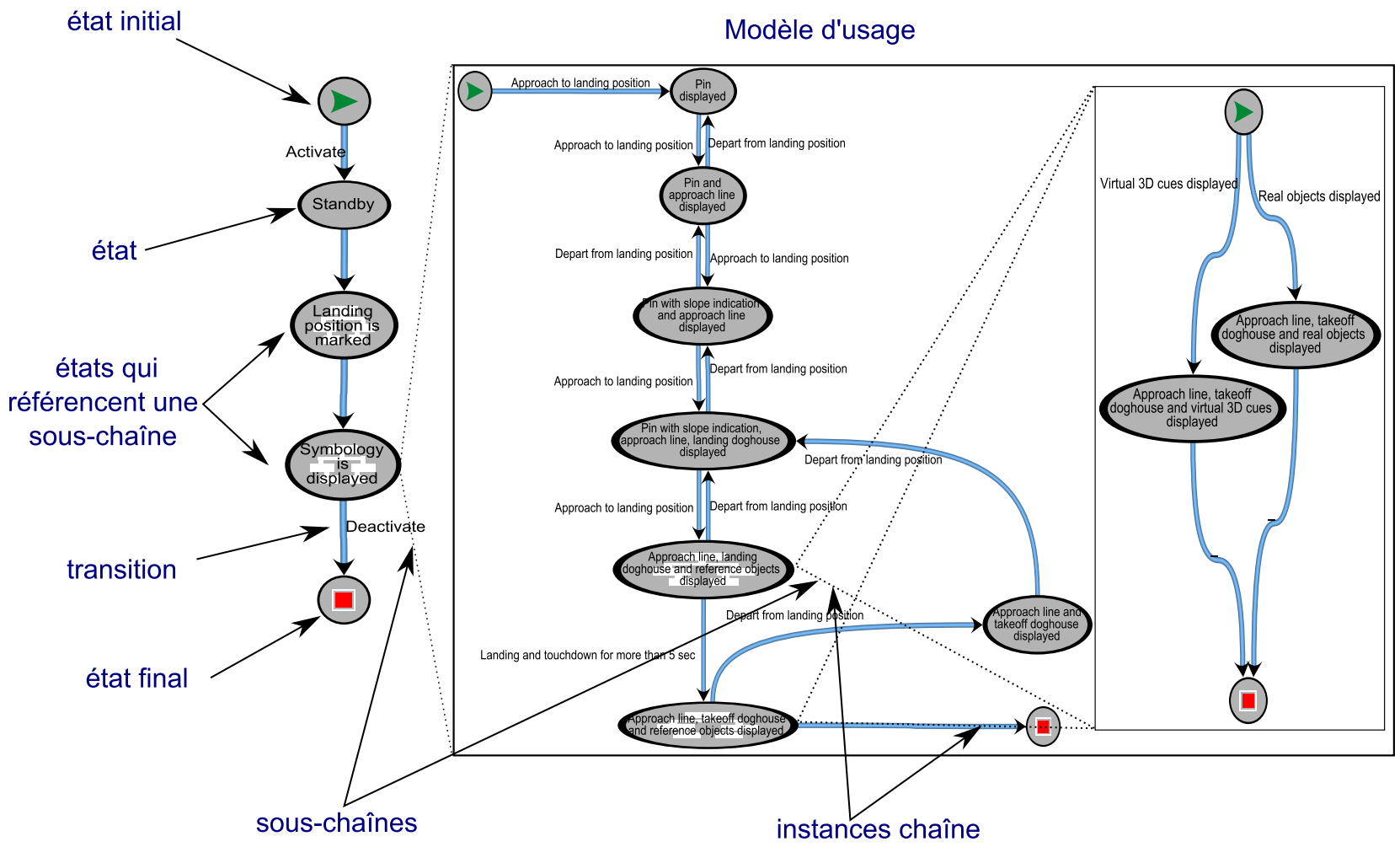


FIGURE *3.6 – Exemple du modèle d'usage de MaTeLo*

Un cas de test commence toujours par l'état initial s_0 et finit par l'état final s_n. La génération de cas de test consiste à définir une stratégie de test, en choisissant un profil d'usage (profil d'utilisateur) et un algorithme de génération. Par la suite, la génération des cas de test consiste à définir un chemin selon le profil et selon l'algorithme de génération de test (critères de sélection des tests).

Une exigence peut être associée aux transitions du modèle d'usage. Chaque exigence à un rôle, par défaut positionné à *nécessaire* au niveau de la transition annotée par celle-ci. Elle peut avoir aussi comme rôle *suffisant*, ou *nécessaire et suffisant*. Une exigence est dite couverte par un cas de test, quand l'ensemble des transitions nécessaires ou une transition suffisante sont présentes dans le chemin du cas de test. Le lien entre les exigences et les transitions est réalisé manuellement dans l'outil MaTeLo.

3.5.2 Définition de la formalisation du modèle d'usage

Hélène Le guen [LG05] a proposé une formalisation mathématique du modèle d'usage de MaTeLo, que nous représentons ci-dessous. Soit un graphe de test noté $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ et représenté par le tuple (S, s_0, s_n, T, V, F, R), où :

- S : un ensemble fini d'états ;
- s_0 (Invoke) $\in S$: unique état de début dans la chaîne.
- s_n (End) $\in S$: unique état de fin dans la chaîne.
- V : un ensemble des entrées (stimulis), sorties (résultats attendus) et variables globales.
- F : un ensemble de fonction de *"fonction de transfert définie"* : f (V1) = V2 avec V1, V2 $\in$ V. V1 : un ensemble spécifique de variables globales et / ou une sélection des vecteurs d'entrée, V2 : un ensemble de vecteurs de sortie ;
- T : un ensemble de transitions probabilistes de la forme $s \xrightarrow{p_{ss'}, R_{ss'}, f_{ss'}} s'$, avec $s \in S$ / s_n, $s' \in S$ / s_0, $R_{ss'}$ est le sous-ensemble des exigences associé à la transition $t_{ss'}$, $p_{ss'} \in [0, 1]$ est la probabilité de choisir s' à partir de s et $f_{ss'} \in$ F est les données d'entrées et sorties associées à la transitions $t_{ss'}$. La somme des probabilités associées aux transitions sortantes de l'état s doit être égal à 1.
- R : l'ensemble des exigences du système sous test (SUT).

Pour réaliser un modèle d'usage valide (voir figure 3.6), il faut respecter les règles suivantes :

- L'état initial s_0 ne doit pas avoir une transition entrante.
- L'état final s_n ne doit pas avoir une transition sortante.
- Tous les états à l'exception de l'état final s_n doivent avoir au minimum une transition sortante.
- Tous les états à l'exception de l'état initial s_0 doivent avoir au minimum une transition entrante.

3.5.3 Réalisation du modèle d'usage sous MaTeLo

MaTeLo offre une perspective pour réaliser un modèle d'usage. Le modèle est constitué d'un ensemble d'états et transitions. Les transitions sont associées aux actions à exécuter sur un SUT. Un pas de test dans un chemin de test est composée d'une entrée (Stimulation) et d'un ensemble de résultats attendus. Chaque partie du modèle peut être liée aux exigences (voir la figure 3.7).

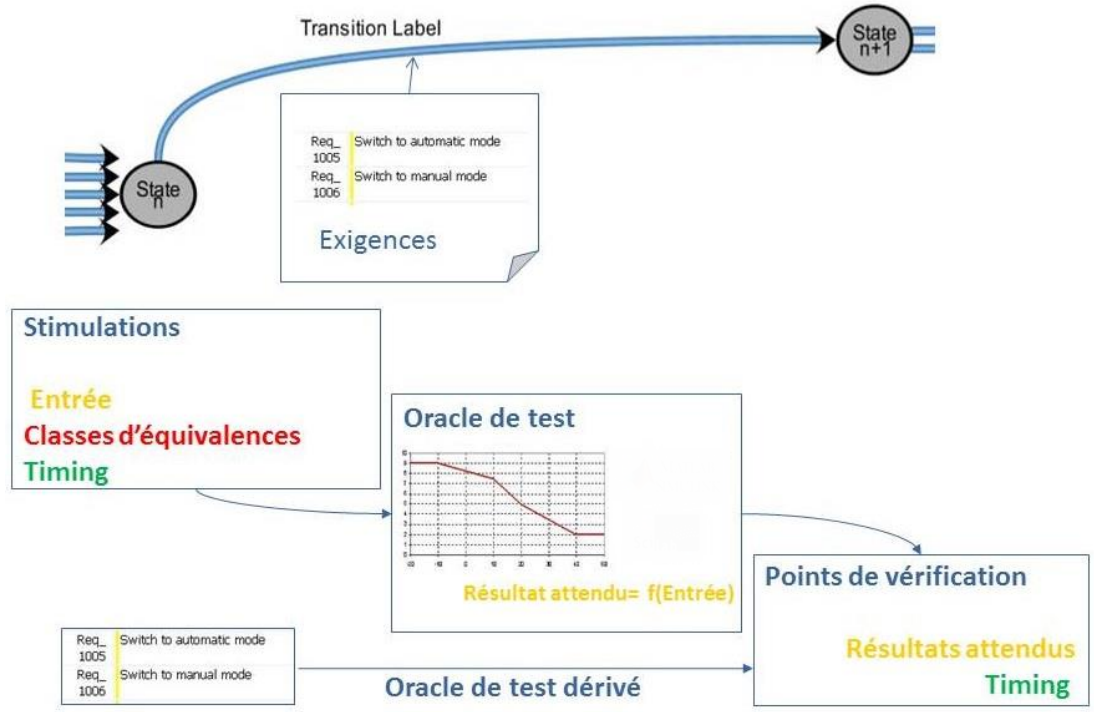


FIGURE *3.7 – Une transition du modèle d'usage*

Les transitions et les données d'entrées qui lui sont associées peuvent être pondérées par des probabilités d'usage comme illustré par la figure 3.8.

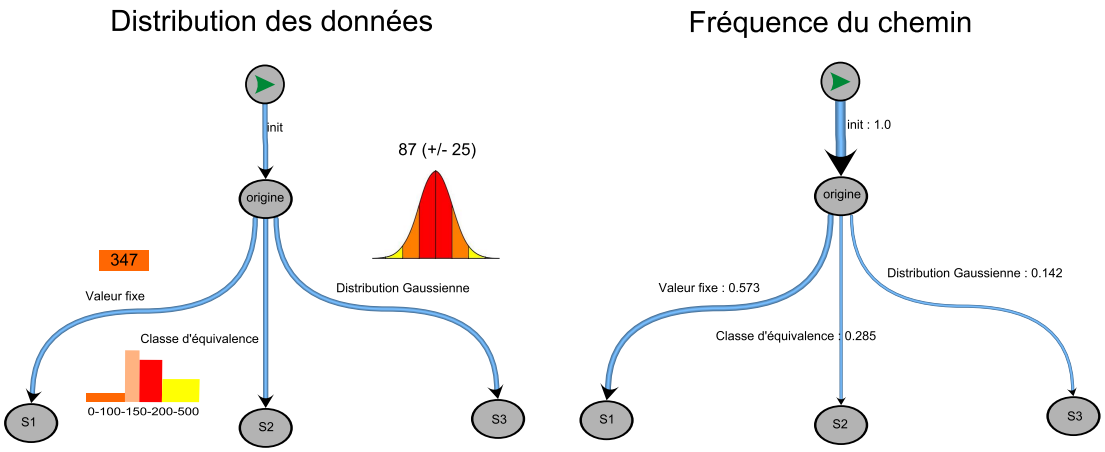


FIGURE *3.8 – Les profils d'usage*

3.5.4 La génération des cas de test avec MaTeLo

Nous rappelons que la génération des cas de test n'est pas un problème traité par cette thèse et que nous réutilisons les approches et les outils existants. MaTeLo génère automatiquement un ensemble fini de cas de test (nominal, intuitif, non-intuitif) en conformité avec le profil d'usage du système sous test. Ces cas de test sont compatibles avec plusieurs bancs de tests. La génération des cas de test est configurée à l'aide d'une stratégie de génération qu'il faut définir sous MaTeLo. Il est possible de créer et de mémoriser un grand nombre de stratégies. Pour être en mesure de générer, les paramètres suivants doivent être définis :

- Champ d'application de la génération, c'est-à-dire le choix des chaînes avec la possibilité de sélectionner une partie ou la totalité du modèle d'usage qui doit être pris en compte pour la génération.
- Le profil d'usage.
- L'algorithme de génération de la suite de tests et ses paramètres.

Algorithme de génération

Au niveau de l'outil MaTeLo, plusieurs algorithmes de génération de cas de test sont définis, mais cette liste n'est pas exhaustive, car il est possible de définir des nouveaux algorithmes selon le besoin, à l'aide des possibilités offertes par la plateforme d'Eclipse RCP sous laquelle MaTeLo est développé.

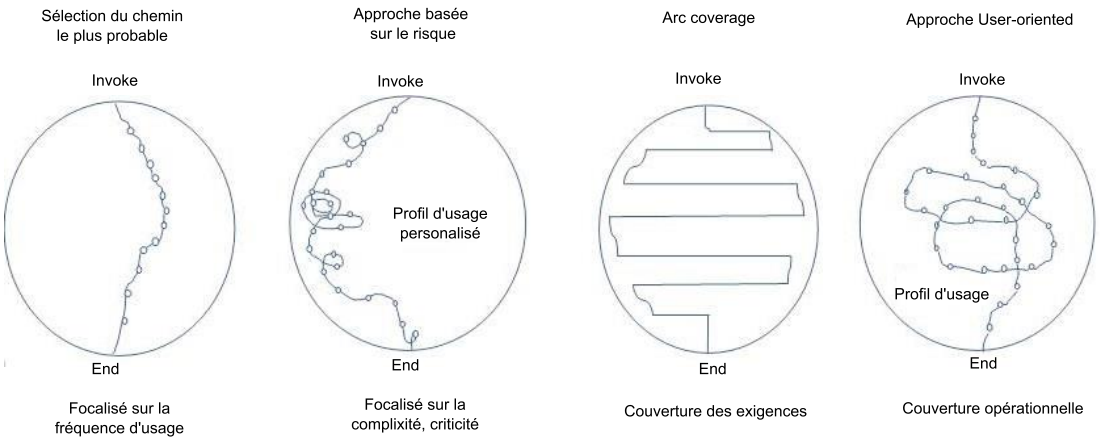


FIGURE *3.9 – Les approches de MaTeLo pour la génération automatique des cas de test*

La figure 3.9 présente les principales approches de génération de tests définies dans MaTeLo.

- Le chemin le plus probable : l'algorithme choisit la transition la plus probable pendant le parcours du modèle.
- Approche basée sur le risque : la sélection des transitions à partir de l'état de début jusqu'à l'état fin est basée sur le profil d'usage personnalisé.
- Arc coverage : cet algorithme couvre l'ensemble du modèle d'usage : il passe par tous les états et transitions en un minimum de séquences de test.
- User-oriented : cette approche consiste à couvrir un ensemble fini de transitions à partir de l'état début jusqu'à l'état fin du modèle d'usage. Les transitions qui suivent sont choisies au hasard, selon le dispositif probabiliste défini par le profil en cours.

En somme, les cas de test générés peuvent être soit sous format textuel comme dans la figure 3.11 qui montre un cas de test pour une exécution manuelle soit sous format script pour une exécution automatique.

Le script de test peut être compatible avec plusieurs bancs de tests comme schématisé par la figure 3.10.

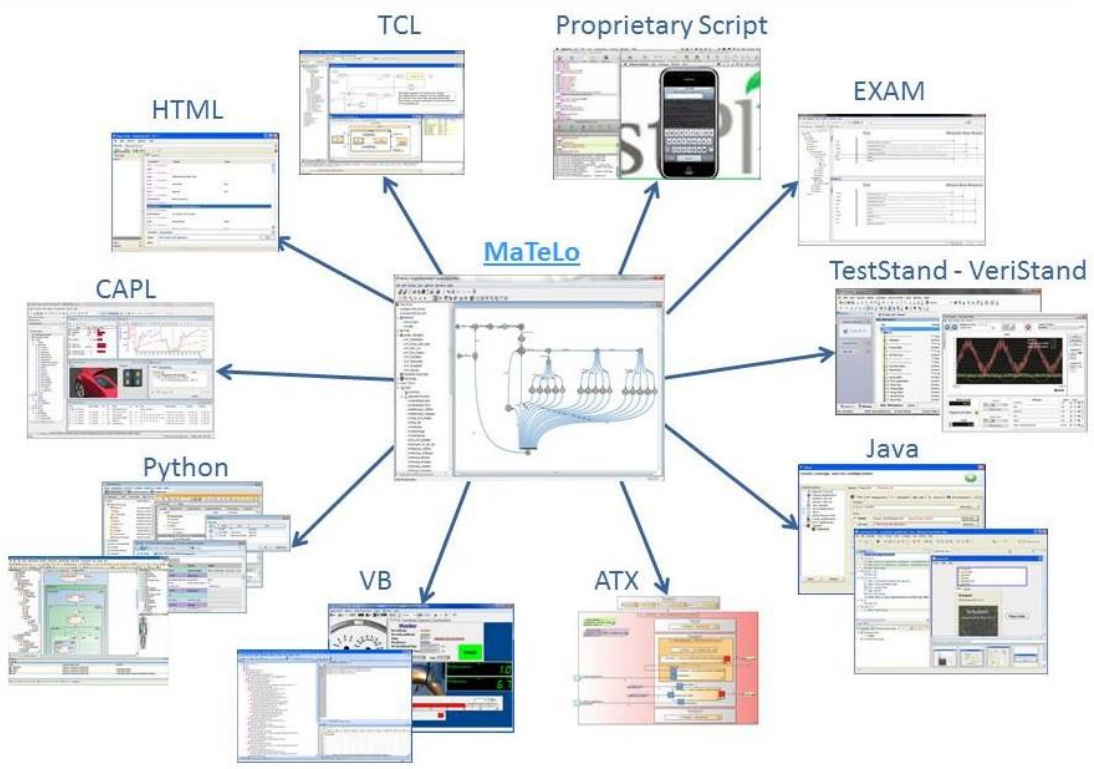


FIGURE *3.10 – L'écosystème de MaTeLo*

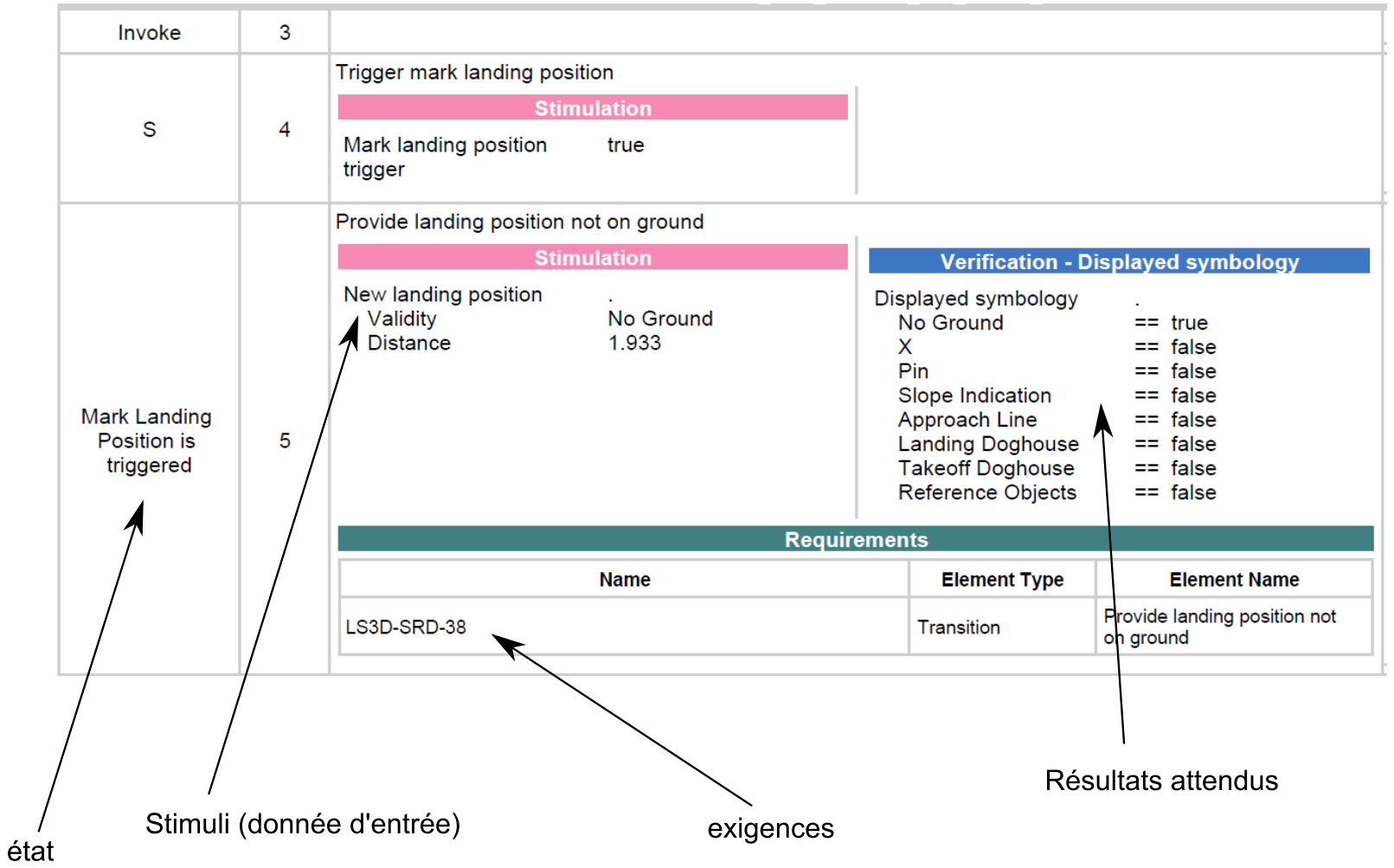


FIGURE *3.11 – Un cas de test manuel généré par MaTeLo*

3.6 Résumé

Les approches de tests peuvent être soit de type statique soit de type dynamique. Ces techniques visent à améliorer le rendement sur l'investissement de tests, en maximisant le nombre de bugs trouvés par un nombre fini de cas de test. Cela implique, notamment, d'être capable de scruter l'intérieur du programme et de faire certains raisonnements, des hypothèses, et cela, sans besoin d'exécuter le système pour jouer les cas de test.

Les stratégies de tests logiciels peuvent être de type boîte noire ou boîte blanche et dans certains cas une combinaison des deux approches. L'approche boîte blanche crée les cas de test en examinant la structure interne du système, tandis que les techniques de la boîte noire examinent les exigences logicielles sans tenir compte de la structure interne du système. L'approche boîte grise est une combinaison entre boîte blanche et les techniques de la boîte noire. Les techniques de tests logiciels peuvent être appliquées pour faire des tests unitaires, tests d'intégration, tests de sous-système, tests d'approbation pour l'assurance qualité logicielle et aussi pour le test de système, en particulier le *model-based testing*, une technique utilisée en industrie qui propose un processus à suivre : construction du modèle de test, définition des critères de sélection des tests, génération et exécution des cas de test. Le modèle de test est utilisé comme base pour dériver les cas de test, afin de valider les exigences fonctionnelles et vérifier leurs consistances. Dans le cadre de cette thèse nous nous intéressons au *model-based testing* pour la validation des lignes de produits.

Chapitre 4

Le test des lignes de produits

L'ingénierie des lignes de produits encourage à introduire la variabilité plutôt dans les phases de développement de la ligne de produit. Le but est d'élaborer un développement performant de la diversité des applications partageant une architecture commune. Cependant, le test figure parmi les priorités dans le cycle de vie de la ligne de produits.

Garantir le bon fonctionnement des artefacts durant le processus de l'ingénierie du domaine est essentiel pour la réussite de l'ingénierie des lignes de produits. Une erreur non détectée dans un artefact définit dans l'ingénierie de domaine peut affecter toutes les applications de la ligne de produits dans laquelle cet artefact est réutilisé. La plupart des techniques de test actuelles sont effectives pour des systèmes uniques et ne peuvent pas être directement appliquées aux lignes de produits [MP14].

De très nombreux travaux de recherches ont porté sur les stratégies et les techniques de test adaptées pour la variabilité, y compris la vérification formelle, l'analyse statique et le test dynamique. La cohérence du modèle de la variabilité est une condition préalable pour la plupart des techniques de vérifications et validations de l'ingénierie du domaine. D'autres stratégies de test visent à sélectionner un sous-ensemble de configurations à tester, et cela, à l'aide de certaines techniques comme *pairwise* [POS+12] ou *T-wise* [PSK+10].

Dans cette section, nous présentons un ensemble de travaux de recherche qui concerne la validation et la vérification des lignes de produits qui entrent dans le cadre de nos travaux de thèse. Nous avons choisi de les classer comme suit :

- L'analyse de la variabilité.
- Sélection d'un ensemble de configurations.
- Techniques de tests pour les lignes de produits.
- *Model-based Testing* pour les lignes de produits.

De plus, nous présentons un tableau comparatif des approches qui ont étendu le processus du model-based testing pour le test des lignes de produits.

4.1 Approches d'analyse de la variabilité

L'analyse statique repose sur le traitement des sources du logiciel, dans le cadre des lignes de produits, cela peut concerner les *features*, qui sont représentées à l'aide d'un modèle de variabilité. L'analyse de la variabilité vise à vérifier et à s'assurer si certaines propriétés d'un modèle de variabilité sont respectées. Il peut être utilisé en complément avec le test dynamique, ou comme un outil d'aide au développement ou de prise de décision par les industriels pour définir des nouveaux produits. C'est-à-dire, vérifier qu'on peut avoir au moins une configuration valide, absence de *features* mortes, générer un ensemble de configuration valides. Ce type de vérification permet de vérifier pendant le processus de l'ingénierie du domaine que l'ensemble des applications dérivées sont fonctionnelles.

Les techniques d'analyse existantes permettent d'analyser les produits individuellement, dans le cadre des lignes de produits, ce type de traitement peut être réalisé sur un nombre de produits limités et non à une ligne de produits avec un nombre important de variantes. Plusieurs stratégies d'analyses dédiées à la variabilité ont vu le jour pour traiter le problème de l'explosion combinatoire et permettre d'appliquer les techniques d'analyse statique sur les lignes de produits. Notamment, d'appliquer une stratégie classique, qui consiste à prélever un nombre réduit de produits, généralement sur la base de certains critères de couverture, ou à la base des contraintes et les propriétés de la ligne de produits [OMR10, PSK+10].

L'analyse de l'ingénierie de domaine a été traité par Metzger et al. [Met07] et Lauenroth et al. [LMP10] qui ont abordé trois stratégies d'assurance de la qualité de lignes de produits (par exemple, la vérification des modèles), à savoir la stratégie de vérification des points communs de la ligne de produits, la stratégie d'échantillonnage, similaire à la stratégie basée sur l'analyse des *features*, elle ne peut révéler que certains bugs dans une ligne de produits donnée et la stratégie exhaustive (similaire aux approches d'analyses par produit). L'idée de la stratégie de vérification des points communs est d'encourager la validation des artefacts de base pendant la phase de test du domaine [Met07], afin que les tests d'application soient focalisés sur les parties spécifiques de l'application non couvertes par les tests de domaine. Dans le cas normal, les produits complets ne sont pas obtenus durant l'ingénierie de domaine, car celle-ci se concentre sur le développement des modèles de base. Par conséquent, dans la plupart des cas, les tests du domaine peuvent être effectués d'une manière limitée [TAK+14].

Thüm et al. [TAK+14] ont examiné 123 stratégies de vérification proposées pour les lignes de produits. Ils proposent une classification de ces techniques d'analyse en trois principales stratégies : *product-based*, *feature-based* et *family-based analyses*. Ces techniques peuvent être même combinées pour donner lieu à d'autres nouvelles approches. Elles indiquent comment l'analyse traite la variabilité du logiciel. Thüm catégorise les approches par rapport à l'implémentation et la stratégie de spécification, il identifie également les stratégies de spécifications qui ont été appliquées dans le domaine de la vérification de ligne de produits.

D'autres approches améliorent cette stratégie, en mettant l'accent sur les interactions entre les *features* qui peuvent être source de bugs dans les produits, soit par une

mauvaise combinaison soit par le non-respect des relations et des contraintes définies au niveau de la ligne de produits [NL11, ZZM13]. Par conséquent, ces approches récupèrent un ensemble minimal de produits répondant à un critère de couverture donnée et seulement ces produits sont analysés.

Parmi ces études, nous pouvons citer l'extraction automatique d'un ensemble de configurations valides à l'aide de solveurs dédiés à la variabilité, comme FAMA introduit par Benavides et al. [BTRC05], FAMILIAR proposé par Acher et al. [ACLF13] permettant de détecter des erreurs de configurations grâce à l'analyse automatique de la variabilité. Malgré ces solutions, réduire l'espace des combinaisons possibles reste un challenge pour le test des lignes de produits. Hervieu et al. [HBG11] ont proposé PACOGEN, une approche qui permet de sélectionner un sous-ensemble de configurations pertinentes pour le test selon les critères de couvertures *pairewise*. Néanmoins, ces méthodes ne s'intéressent pas à la génération de cas de test pour les lignes de produits.

Kim et al. [KBBK10] proposent une technique d'analyse pour une famille de produits basée sur la programmation orientée *features*. Ils appliquent leur technique d'analyse sur une famille de produits, afin de réduire l'ensemble des produits possibles, pour lequel les propriétés de sécurité doivent être suivies pendant l'exécution. Les propriétés de sécurité sont définies dans AspectJ. Dans une première étape, l'analyse statique réalisée sur la famille exclut les configurations qui ne peuvent pas violer les propriétés de sûreté. Le résultat de l'analyse statique est un modèle de variabilité spécialisé pour représenter des produits qui sont examinés dans la deuxième étape.

Perrouin et al. [PSK+10] proposent une approche associée à une plateforme supportant la génération automatique de cas de test pour les lignes de produits logiciels. Ce travail est motivé par les problèmes d'évolution et la possibilité d'utilisation. En ce qui concerne le premier problème, ils ont combiné les tests combinatoires d'interaction, comme un moyen systématique pour échantillonner un ensemble réduit de cas de test, avec deux stratégies *divide-and compose*. Ces stratégies portent sur les limitations d'évolution des solveurs SAT utilisés pour générer les cas de test qui satisfont toutes les contraintes capturées dans un modèle de *features*. Grâce à ces stratégies, ils sont en mesure de générer automatiquement des suites de cas de test pour une ligne de produits de taille moyenne tels que *AspectOPTIMA*. Ces stratégies sont évaluées par des métriques calculées et discutées sur la base des facteurs qui influencent la génération des cas de test.

Oster el al. [OMR10] introduisent une méthodologie pour appliquer les techniques *pairwise* aux lignes de produits. Ceci est fait en convertissant le modèle de *feature* en un problème de résolution de contraintes binaires (CSP). Pour ensuite, générer un ensemble de produits valides contenant toutes les paires valides de *features* en respectant les critères de couverture *pairwise*.

Dans un récent travail de recherche, Galindo et al. [GAA+14] présentent VANE (*Variability-based Testing*), une nouvelle approche de test pour dériver des variantes de séquence vidéo pour le test des logiciels du traitement vidéo. VANE encode dans un modèle de variabilité ce qui peut varier à l'intérieur d'une séquence vidéo. En se basant sur les critères de couvertures *T-wise*, VANE synthétise des variantes de séquences vidéo correspondant aux configurations, tout en optimisant les fonctions dépendantes

des attributs.

Bien que l'échantillonnage semble être réalisé dans les règles par rapport à l'analyse de base, il peut rester encore des bugs non identifiés dans les produits de la sélection. Tandis que la plupart des techniques de tests combinatoires tels que *pair-wise* ou *t-wise* sont souvent proposées pour vérifier des systèmes uniques [NL11], ils ont aussi été appliqués récemment pour les lignes de produits, afin d'optimiser la vérification des types [JWEG07, LvRK$^+$13], le test statique [LvRK$^+$13, OMR10, PSK$^+$10, HBG12], modèle *checking* [OMR10, PSK$^+$10, HBG12] [Plath and Ryan 2001 ; Apel et al. 2013c].

Dans un contexte de validation des modèles de tests de lignes de produits, Classen et al [CHSL11] proposent un modèle *checking* pour la ligne de produits. Cette méthode de vérification est conçue pour vérifier un modèle états - transitions d'une ligne de produits entière, en reliant le modèle de *feature* avec le modèle de transitions (FTS), une représentation mathématique commune du comportement du système. Le lien est établi entre les *features* et les transitions du FTS. Cette méthode permet de vérifier le modèle complet.

4.2 Techniques de test pour les ligne de produits

McGregor [McG01] souligne la nécessité d'un processus de test bien conçu et examine les relations complexes entre les plateformes et les produits dans son rapport technique. Il discute dans ce travail d'une structure pour des méthodes et artefacts de tests en alignement avec la structure des produits de la ligne de produits construits.

Kolb et al. [KM] discutent de l'importance et la complexité de tester une ligne de produits logiciels. Ils mettent le doigt sur le besoin de lignes directrices et les techniques exhaustive et efficaces pour tester systématiquement les lignes de produits. Ils favorisent également l'idée de créer des cas de test génériques.

Tevalinna et al. [TTK04] abordent le problème de l'applicabilité des méthodes de tests existantes aux lignes de produits, en particulier les tests de régression et les techniques de tests des programmes partiels, car ils considèrent qu'une ligne de produits peut être vue comme un programme incomplet. Ils mettent l'accent sur l'absence de techniques efficaces pour les tests de ligne de produits et présentent le test en tant que processus orienté uniquement produit. Ils discutent également de quatre stratégies différentes pour modéliser les tests de lignes de produits : le test par produit, les tests incrémentals de lignes de produits, instanciation des artefacts réutilisables et la répartition des responsabilités [TTK04].

Pohl et al. [PBVDL05] présentent et discutent quatre stratégies définies pour les tests de ligne de produits. Ils discutent des stratégies intégrant des tests à différents niveaux, y compris les tests unitaires, tests d'intégration et les tests de la ligne de produits. La stratégie utilisée consiste à effectuer les tests à tous les niveaux et les exécuter pour tous les produits de la ligne de produits. En revanche, pour la stratégie appliquée au niveau de l'ingénierie d'application, seulement les produits livrés sont testés. Enfin, les artefacts de bases liés aux points communs et la variabilité de la ligne de produits sont testés en ingénierie de domaine, par la suite tous les produits sont

testés séparément.

Les cas de test de ligne de produits créés durant l'ingénierie de domaine sont généralement dérivés à partir des exigences fonctionnelles et généralisés à partir des cas de test existants [McG01]. Les cas de test provenant des cas d'utilisation pendant l'ingénierie de domaine devraient être étendus à l'ingénierie d'application. Selon Nebut et al. [NFLTJ04] les cas de test provenant de l'ingénierie de domaine peuvent être étendus à l'ingénierie d'application pour la réutilisation. Ils proposent un algorithme pour générer automatiquement des cas de test spécifiques à un produit de la ligne de produits en se basant sur des exigences exprimées en UML sous forme de cas d'utilisations [MP07]. Ils évaluent leur approche sur une petite étude de cas.

Knauber et al. [KS] examinent comment combiner la programmation orientée aspect et les tests unitaires dans le but d'atteindre la traçabilité entre une instance de la ligne de produits et ses tests. Hartmann et al. [HVR04] proposent d'utiliser les diagrammes d'activités comme modèle de test, qui intègre aussi la variabilité, mais, les cas de test sont dérivés seulement pendant l'ingénierie d'application. Il s'agit d'une approche de test basée sur les modèles (*model-based testing*) pour la dérivation automatique des cas de test, uniquement à partir de cas d'utilisation dans l'ingénierie d'application, qui ne considère pas la réutilisation des cas de test.

Metzeger et al. [MP14] soulignent le problème d'interactions des exigences fonctionnelles avec les produits et les artefacts de la ligne de produits, où ils mettent l'accent sur le manque de technique intelligente pour gérer ce type d'interactions complexes présent dans les lignes de produits. Comment réaliser une interaction continue entre les exigences, les artefacts du domaine et la variabilité ? Comment évaluer l'impact de l'évolution des exigences sur le développpement de la ligne de produits et en particulier les artefacts de tests ?

Engström et al. [ER11] ont réalisé une étude sur 64 travaux de recherche qui concernent les tests de ligne de produits, il semble que 40% de ces travaux sont consacrés aux systèmes et aux tests d'acceptation. L'objectif le plus fréquent est la génération automatique de cas de test à partir des exigences. Les exigences peuvent être basées sur un modèle, les cas d'utilisation [RMP06], les spécifications formelles [Mis06] ou un langage naturel.

Il existe peu d'outils disponibles, spécialement pour le test des lignes de produits, couvrant le processus de test en entier [dMSNRdCM+11]. Les quelques outils disponibles opèrent sur différentes phases du cycle de vie du développement, pour soutenir les activités de validation nécessaires et pour fournir un ensemble de produits utilisables (phase de spécifications, phase de tests et maintenance) [CN02].

La sélection des outils disponible pour la validation des lignes de produits, qui peuvent soutenir les objectifs de l'entreprise et les besoins techniques des développeurs est très difficile [LKL12], à cause des deux processus de développement impliqués. On doit respecter, certains critères doivent être respectés pendant le choix des outils de lignes de produits. L'exemple des critères présentés, inclut les fonctionnalités proposées par l'outil, le coût, la stabilité des fournisseurs, la formation, l'interopérabilité avec d'autres outils et le support [McG01]. Ces critères sont d'actualités encore aujourd'hui.

Bertolino et al. [BG04] ont développé PLUTO (Product Lines Use case Test Op-

timization), une méthodologie pour dériver des cas de test à partir des exigences de la ligne de produits décrites comme des cas d'utilisations [BG03]. La méthodologie est inspirée de la technique de partition de catégorie, élargie avec la capacité de gérer la variabilité dans les lignes de produits, afin d'instancier des cas de test de produits spécifiques membres de la ligne de produits sous test. L'approche est basée sur les exigences fonctionnelles exprimées avec un langage naturel. Cependant, la dérivation des cas de test est réalisée en partie manuellement [BG04]. L'objectif de PLUTO est de proposer un processus simple de test de lignes de produits, basée sur les cas d'utilisation, pour but de dériver une série de plans de test, et une suite de cas de test spécifique à chaque application personnalisée de la ligne de produits.

Condron [Con04] propose une approche d'automatisation des tests de ligne de produits pour l'ingénierie de domaine, en combinant des *frameworks* de l'automatisation des tests à partir de divers endroits dans la ligne de produits pour lesquelles le test est nécessaire. McGregor et al. [MSM04] proposent et évaluent une approche de conception logiciels d'automatisation de tests, qui est basée sur la correspondance entre la variabilité dans les produits et le test logiciel.

Ridene et Barbier. [RB11] introduisent DSML, un langage dédié au test des dispositifs portables. Ils présentent MATeLe ; un outil qui répertorie un large éventail de scénarios de tests dans le *Cloud*, des liens entre les caractéristiques du dispositif portable ; aussi les cas de test sont tissés, afin d'extraire selon la demande les cas de test équivalents à l'application mobile sous test.

Ganesan et al. [GKK+07] proposent un modèle économique pour le test des lignes de produits, après un constat sur la pratique de test exercée aujourd'hui. Grâce à une étude empirique qui compare deux stratégies de test différentes, appliquées sur un cas d'étude industriel, ils ont constaté qu'en utilisant un modèle économique basé sur une stratégie de test axée sur la réutilisation, le bénéfice sur le coût, en moyenne, peut atteindre 13 % avec 87 % de certitude, par rapport à une stratégie de test classique adaptée pour un produit et non pour une famille de produits. Le rapport-bénéfice est obtenu à l'aide des résultats de la simulation avec la méthode *Monte-Carlo*. Cette méthodologie pourra être utilisée pour évaluer et comparer d'autres méthodes de test, de la ligne de produits.

4.3 Combinaison des techniques d'analyse statique avec les méthodes de test dynamique

Une stratégie typique consiste à échantillonner un nombre réduit de produits, généralement sur la base de certains critères de couverture, ou d'autres propriétés possibles de la ligne de produits entière [OMR10, PSK+10, NL11]. Cette technique de test statique a été combinée avec le test dynamique pour optimiser l'activité de tests de la variabilité. Plusieurs travaux ont adopté cette technique, nous en présentons quelques-uns.

Afin de réduire l'effort de test, McGregor [McG01] propose une conception de test combinatoire, où la technique *pair-wise* est utilisée systématiquement pour sélectionner

une combinaison de variantes, pour être testée à la place des toutes les combinaisons possibles. Il examine ailleurs [McG08] la nécessité d'accroître les connaissances sur les défauts susceptibles d'apparaître dans une instance de ligne de produits et présente un modèle de défaut. Des modèles de défauts peuvent être utilisés en tant que base pour la conception de cas de test et comme support pour estimer l'effort de test nécessaire pour détecter un certain type de défauts. Dans cette optique, Cohen et al. [CDS06] proposent une technique de tests d'interactions raccordée aux critères de couverture combinatoires.

Kim et al. [KBK11] présentent une approche globale pour générer des cas de test généraux. Pour chaque cas de test, on dérive un ensemble de produits couvert par ce dernier. Ils étendent le flot de contrôle et le flot de données d'analyse avec des informations de la variabilité de la ligne de produits, afin de tracer les effets des *features*.

Cabral et al. [CCR10] ont réalisé une étude sur les éléments de la variabilité qui peuvent impacter d'une façon négative le test d'une ligne de produits et accroître le nombre de configurations possibles à tester, en mettant l'accent sur les options et les points de variation dans une ligne de produits. Dans ce travail, ils proposent une approche de test basée sur un graphe nommé *FIG*. Cette méthode consiste à sélectionner à partir des relations entre les éléments de la variabilité définis au niveau du graphe, les produits et les *features* valides pour le test, afin de les transformer par la suite en cas de test qui correspondent au produit sous test.

4.4 Model-based Testing pour les lignes de produits

Le *model-based testing* n'est pas adapté pour tester une ligne de produits, car le processus et le modèle de test sont effectifs que pour des systèmes individuels. Pour tester une ligne de produits avec l'état actuel du MBT, il est possible de réaliser un modèle de test pour chaque produit, par contre, dans le cadre de l'ingénierie des lignes de produits ce type d'approche n'est pas favorable. Tous les avantages du MBT sont perdus et les points communs entre les produits ne sont pas exploités de la même manière la réutilisation ne sera pas instaurée. Une telle pratique en industrie ne peut que couter cher

Plusieurs travaux de recherche ont proposé d'étendre le MBT pour introduire la variabilité et pour bénéficier de sa puissance afin de concevoir des tests pour une ligne de produits. C'est également l'objectif principal de nos travaux. Dans ce qui suit, les approches liées à la contribution de cette thèse sont résumées et discutées.

4.4.1 Travaux d'extension du MBT pour les lignes de produits

ScenTED

Reuys et al. [RKPR05] présentent *"ScenTED"*, une technique de *model-based testing* pour le test des lignes de produits. Ils proposent d'étendre le MBT et de l'adapter à l'ingénierie des lignes de produits : les artefacts de tests dans l'ingénierie de domaine sont étendus pour représenter la variabilité, en particulier les diagrammes d'activités

et diagrammes de séquence. Le diagramme d'activité est créé manuellement et sert de modèle de test pour la ligne de produits. La variabilité est décrite à l'aide des chemins alternatifs au sein des diagrammes d'activité. Les cas de test de domaine sont générés sur la base des diagrammes d'activité et les critères de couverture de branche sont utilisés pour s'assurer que chaque branche est couverte par au moins un cas de test. Les cas de test pour différents produits peuvent être obtenus facilement à partir des cas de test de domaine. Au moment où un produit est dérivé, la variabilité est résolue et les cas de test de produits spécifiques sont dérivés, à partir des cas de test de domaine. Ainsi, les cas de test de domaine peuvent être réutilisés pour différentes applications.

CADeT

Olimpiew [Oli08] a mis en place CADeT (Customizable Activity Diagrams, Decision Tables, and Test specifications), une approche qui vise le test niveau système. Pour modéliser les exigences en cas d'utilisation, elle se base sur PLUS (*Product Line UML based Software engineering*) [Gom05]. Cette technique, permet pendant la phase d'analyse de domaine de créer manuellement selon l'approche PLUS, les cas d'utilisation textuel à partir des exigences de la ligne de produits. En outre, un modèle de *features* est créé pour représenter les différents cas d'utilisation et fournir une vue d'ensemble sur les points communs et sur la variabilité de la ligne de produits. D'ailleurs, les diagrammes d'activités qui incluent la variabilité sont créés manuellement à partir des cas d'utilisation. Par la suite, les cas de test sont dérivés automatiquement, en fonction des diagrammes d'activités.

Pour projeter les cas de test aux différents cas d'utilisation, une table de décision est utilisée, reliant les *features* du modèle de *features* représentant les différents cas d'utilisation à des cas de test correspondants. En choisissant une combinaison de *features* particulières, les cas de test spécifiques à la variante du produit peuvent être dérivés des cas de test du domaine.

MoSo-PoLiTe

Oster et al [OZML11] présentent une approche qui combine un modèle de *features* avec une technique de tests combinatoires et avec le *model-based testing*. Le modèle de *features* est utilisé par l'approche en tant que l'élément central. Il est créé sur la base des exigences de la ligne de produits pendant l'ingénierie du domaine. La sélection d'une combinaison de *features* selon les dépendances et les contraintes se traduit par des configurations qui peuvent être interprétées comme des sous-arbres du modèle de *features* d'origine. La sélection des configurations est réalisée sous MoSo-PoLiTe, réalisée par un algorithme de sélection spécifique basé sur des tests combinatoires.

Tout d'abord, ils transforment le modèle de *features* en un problème binaire de satisfaction de contraintes (CSP) composé de paramètres, des valeurs et des contraintes.

Ensuite, l'algorithme d'extraction de sous-ensemble génère un ensemble minimal de configurations qui respecte les critères de couvertures T-wise.

En revanche, pour la dérivation des cas de test, le modèle de *features* est lié à un modèle de test représentant le comportement de l'ensemble de la ligne de produits. À l'aide d'une méthode de mise en correspondance, l'approche dérive, pour chaque configuration valide dérivée du modèle de *features*, un de test correspondant au comportement de cette configuration. Ce modèle de test est capable de générer des cas de test pour chaque produit sélectionné.

Featured Transition System

Devroey et al. [DPC+14] proposent une approche de hiérarchisation des produits à tester selon des critères basés sur le comportement réel des produits. Ils s'appuient sur des techniques de test statistique, qui génèrent des cas de test à partir d'un modèle d'usage représenté par une chaîne de Markov à temps discret (DTMC). Ce modèle d'usage représente les scénarios d'utilisation du logiciel sous test ainsi que leur probabilité respective. Le profil d'utilisation permet de déterminer la probabilité des scénarios d'exécution et de les classer en conséquence. Cette technique de structuration est utilisée pour donner la priorité aux produits à tester dans une ligne de produits. Cependant, pour relier la variabilité au modèle d'usage, ils introduisent le FTS (Featured Transition System) un modèle de conception décrivant le comportement de la ligne de produits. Le FTS est utilisé pour obtenir le lien manquant entre le modèle d'usage et la variabilité. L'approche propose de supprimer les comportements des FTS qui n'ont pas été retenus lors de l'analyse du modèle d'usage, afin d'obtenir un FTS élagué qui présente seulement les comportements les plus communs, peuvent être analysés pour permettre la réutilisation des tests entre les produits.

4.4.2 Discussion

Le tableau 4.1 ci-après, présente une comparaison des approches précitées avec notre solution que nous présentons dans cette thèse. Les critères de comparaison sont les suivants :

- Type de modèle de test utilisé,
- Type de formalisme de variabilité utilisé,
- Prise en compte de la traçabilité,
- Comment sont gérés les liens de traçabilité entre les *features* et les artefacts de test,
- Résultat attendu de l'approche :
  - sous-modèle de tests spécifiques à un produit,
  - cas de test spécifiques à un produit.
- Exécution des cas de test,
- Implémentation de l'approche dans un outil,
- Évaluation de l'approche.

	ScenTED	CADet	MoSo-PoLiTe	FTS	MPLM
Modèle de test	Diagramme d'utilisation et d'activité	Diagramme d'activité	Diagramme d'activité	Modèle d'usage	Modèle d'usage
Modèle de variabilité	Modèle de *features*	Modèle de *features*	Modèle de *features*	Modèle de *features*	OVM
Traçabilité	non	oui	oui	oui	oui avec les exigences
Type de liens entre les *features* et le modèle de test	manuel	manuel	manuel	manuel	automatique
Modèle de test spécifique produit	non	non	oui	oui	oui
cas de test spécifiques produit	oui	oui	oui	non	oui
cas de test exécutables	non	non	non	non	oui
Outil de support	oui	oui	oui	non	oui
Expérimentation	oui	non	oui	non	oui

TABLE *4.1 – Tableau comparatif des approches Model-based Product Lines Testing*

Comme nous l'avons montré ci-dessus, ces travaux de recherche proposent d'étendre le *model-based testing* afin d'introduire la variabilité et de n'utiliser qu'un seul et unique modèle de test dans le but de modéliser le comportement attendu d'une ligne de produits. Le formalisme le plus utilisé est le diagramme d'activités d'après le tableau 4.1. Dans le cadre de nos travaux, nous nous intéressons au modèle d'usage. Or, à notre connaissance, la seule proposition pour gérer le formalisme spécifique des modèles d'usage pour la validation des lignes de produits a été élaborée par Devroey et al. [DPC+14], tandis que leur approche n'est pas outillée ni évaluée. Pour instaurer la réutilisation au niveau du processus du *model-based testing*, nous avons besoin de deux propriétés supplémentaires par rapport au processus classique : introduire la variabilité dans le modèle de test et relier la description de la variabilité avec les artefacts de test. Pour cela, nous devons adopter des méthodes avancées pour établir et pour maintenir facilement les liens entre les modèles de la variabilité et les artefacts du test. Ces approches devraient favoriser la cohérence entre les différents objets [MP14], ce que malheureusement, les approches présentées n'incluent pas. Elles sont basées sur une gestion manuelle des liens entre les *features* représentées par un modèle de *features* et les transitions du modèle de test, sans aucune logique. En revanche, notre approche est basée sur les exigences fonctionnelles comme un lien entre les *features* du modèle OVM (le modèle de variabilité choisi pour notre approche) et les transitions du modèle d'usage. Lors de la réalisation du modèle d'usage, les exigences sont déjà reliées au modèle. À l'aide de l'outil MPLM, le lien entre les *features* et les exigences est établi. Ensuite, à l'aide de l'algorithme implémenté, les correspondances entre les *features* et les transitions sont déduites automatiquement, ainsi la dérivation automatique des variantes du modèle d'usage est réalisée. Un inconvénient majeur de l'approche *"Scen-TED"*, se reflète dans le fait qu'il n'est pas possible de mettre en relation des points de variation pour décrire de nouvelles dépendances. CADeT propose différents critères de couverture de *features* pour tester les variantes du système. La couverture de test, du point de vue classique et la définition de données de test dans l'ingénierie de domaine ne sont pas pris en charge. Cependant, les cas de test générés ne sont pas exécutables. Notre approche permet d'obtenir des cas de test exécutables à l'aide du modèle d'usage dérivé automatiquement et à l'aide de la réutilisation du processus de génération de cas de test de MaTeLo. L'approche proposée par Oster et al. [SOS11] nécessite un effort important pour la mise en œuvre. Rien que pour réaliser le modèle de test, il faut créer manuellement un diagramme d'utilisation puis le diagramme d'activité correspondant. La gestion de plusieurs artefacts de test à la fois peut être une phase lourde et difficile à gérer surtout pour les lignes de produits. En outre, MoSo-PoLiTe gère mal la traçabilité, car les liens entre les artefacts de test et le modèle de variabilité ne suivent aucune logique. Ainsi, si une exigence change, il est nécessaire de mettre à jour toutes ses relations manuellement. Comme présenté par Berger et al. [BRN+13], une ligne de produits peut contenir plus de 10 000 *features* dans 25 % des projets industriels recensés.

Bien que, toutes les techniques de réutilisation précitées bénéficient de la réduction de l'effort pour les tests, chaque produit doit être testé individuellement. En outre, ces approches de *model-based testing* manquent d'une vraie technique pour instaurer la tra-

çabilité entre un modèle de variabilité, un modèle de test, et les exigences fonctionnelles. D'ailleurs, ces techniques se basent sur des nouvelles approches pour la génération de cas de test immatures et basés sur le prototypage. En revanche, notre approche est basée sur un processus de génération de cas de test matures, implémentés par l'outil industriel MaTeLo, utilisé depuis plus de 10 ans en industrie.

En résumé, nous présentons dans cette thèse MPLM, le fruit de nos travaux. MPLM est une approche qui propose d'utiliser le *model-based testing* dans un contexte de ligne de produits. L'approche propose de réaliser un unique modèle d'usage global pour une ligne de produits. La variabilité est représentée dans un modèle séparé à l'aide d'OVM. La contribution se base sur la possibilité de relier les deux modèles en utilisant les exigences fonctionnelles, ainsi que d'instaurer une méthodologie intelligente pour gérer la traçabilité entre les artefacts de test et la variabilité. Il est, par ce biais, possible de projeter le modèle de variabilité sur un modèle d'usage assimilé à des chaînes de Markov étendues. En considérant l'ensemble des *features* d'un produit, il est alors possible d'extraire des variantes du modèle d'usage à partir du modèle initial et par la suite de réutiliser le processus traditionnel du MBT en vue de générer automatiquement une suite de cas de test exécutables.

4.5 Résumé

L'ingénierie des lignes de produits a permis aux industriels d'optimiser le processus de développement des produits similaires [GKK+07]. Le but de test dans l'ingénierie des lignes de produits est de vérifier et valider les modèles de base définis durant l'ingénierie de domaine avant de les réutiliser en ingénierie d'application [PBVDL05, LKL12, TAK+14]. En effet, l'intérêt est de concentrer les efforts de la validation autour d'une famille de produits plutôt que de tester chaque produit séparément.

Jusqu'à présent, la recherche en matière de tests des lignes de produits a mis l'accent sur la vérification et les tests formels. Par exemple, l'identification de la source des incohérences dans les modèles de variabilité, sélection d'un ensemble de produits valides pour le test [TAK+14].

Par conséquent, il est difficile de trouver des approches utiles, qui ne nécessitent pas des changements majeurs à l'ensemble du processus d'ingénierie logicielle, par exemple, les modèles formels pour les exigences et la variabilité [ER11, MP14]. Dans la plupart des travaux de recherche autour de la validation des familles de produits mettent en évidence la gestion de la variabilité [TAK+14]. Cependant du point de vue test du système, le principal défi est de savoir comment faire face à un grand nombre de tests requis pour un ensemble de variantes de produits qui sont plus ou moins similaires.

Un autre challenge souligné par Metzger et al. [MP14] est la gestion des interactions entre les exigences fonctionnelles et les modèles de base dont la variabilité. Ils mettent l'accent sur la difficulté de garder en continuité des liens entre les artefacts et les exigences pour suivre l'impact de leur évolution sur le développement de la ligne de produits pendant toutes ses phases (conception, code, tests, etc.). Ils sont surpris de l'absence des travaux de recherches pour traiter la gestion de l'évolution des exigences

dans l'ingénierie des lignes de produits.

Les contributions sont le plus souvent de type méthode. L'ingénierie des lignes de produits en général et les tests en particulier ont besoin de nouvelles approches méthodologiques. Cependant, les méthodes doivent être soutenues par des modèles de bases appropriés à leur fondement théorique, outillé pour une utilisation pratique et des mesures pour leur gestion et leur évaluation [ER11]. En outre, comme dans le développement des systèmes uniques, les activités de vérification et de validation devraient commencer dès le début du développement et devraient accompagner toutes les activités dans le domaine et de l'ingénierie d'application [MP14]. Le *model-based testing* est un bon exemple de type de processus à adopter, en particulier avec l'utilisation du modèle d'usage [LG05, UL07].

La partie suivante, *Contribution*, présente une solution combinant les avantages du modèle d'usage et le modèle OVM pour le test des lignes de produits. Nous présentons les concepts de notre approche, concernant la réalisation du modèle d'usage pour bénéficier de MaTeLo en tant qu'un outil de modélisation des tests et la génération automatique des cas de test. Nous décrivons plus en détail, notre méthodologie de traçabilité entre les *features* et les éléments du modèle de test. En suite, nous présentons l'algorithme de dérivation automatique des variantes du modèle d'usage, afin de permettre la génération automatique des suites de tests. Nous discutons enfin, les grands choix adoptés pour notre méthodologie.

Deuxième partie

Contribution

Cette partie présente en détails notre contribution qui introduit une nouvelle approche pour le test des lignes de produits. Cette approche réalise une combinaison entre le *model-based testing* et l'ingénierie des lignes de produits. Elle est pilotée par les trois concepts, le modèle de variabilité, les exigences fonctionnelles et le *Model-based Testing*.

Notre approche combine ces trois concepts pour le test des lignes de produits. Le *Model-Based Testing* permet de générer automatiquement des cas de test pour un système à partir des exigences fonctionnelles. Dans ce travail, nous utilisons le modèle d'usage de MaTeLo (un modèle de test basé sur les chaînes de Markov) [LG05] pour représenter les scénarios d'utilisation possibles d'une ligne de produits sous test [KS60, LMT04]. Néanmoins, pour concevoir un modèle de test de la ligne de produits, le formalisme de modèle d'usage doit être capable de représenter les points communs et les points de variation de la ligne de produits. Pour répondre à notre besoin, nous proposons une adaptation dans le chapitre 5.2. Cette introduction de la variabilité dans le modèle d'usage permet la dérivation automatique des variantes de modèle d'usage pour un ensemble de produits similaires, en se basant sur le modèle d'usage de la ligne de produits.

La variabilité de la ligne de produits est représentée par OVM [PBVDL05]. Le modèle de variabilité OVM fournit une structure à plat des exigences de la ligne de produits et représente les parties communes et variables d'une ligne de produits. La sélection des *features* selon les dépendances et les contraintes donne lieu à des variantes de configurations. Généralement, les *features* peuvent être liées à des artefacts de développement [PBVDL05] tels que des fragments de code ou des modèles de comportement, à des cas de test ou bien à des artefacts de tests [OZML11, DPC$^+$14]. Par la suite, lors de la sélection d'une configuration à l'aide des *features*, cela implique que la sélection de certains artefacts peut conduire à dériver un produit. Le modèle de variabilité peut être utilisé également pour analyser la variabilité de la ligne de produits, afin de sélectionner le sous-ensemble de configurations selon les critères définis.

Cependant, le modèle de variabilité est séparé des artefacts de tests. Au cours de la réalisation du modèle d'usage, les exigences fonctionnelles sont associées à des transitions du modèle d'usage. Afin de permettre la dérivation des sous-parties du modèle d'usage de la ligne de produits correspondant à un produit, nous associons les exigences fonctionnelles aux *features* du modèle de variabilité OVM. De cette façon, les correspondances entre les *features* et les transitions du modèle d'usage peuvent être déduites automatiquement. En outre, nous proposons une extension de la formalisation d'OVM proposée par Metzger et al. [MPH$^+$07] pour exprimer les liens entre les *features* et les exigences fonctionnelles.

Pour le test de la ligne de produits, nous configurons un ensemble de produits selon le modèle OVM. Nous proposons un algorithme pour dériver automatiquement des variantes du modèle d'usage. La dérivation d'une variante consiste à sélectionner les exigences relatives à chaque *feature* qui compose une configuration. Par la suite, nous sélectionnons toutes les transitions associées à la sélection des exigences identifiée. Enfin, nous élaguons le modèle d'usage de la ligne de produits en fonction des exigences et des transitions identifiées, en produisant ainsi un modèle d'usage valide, équivalent à la spécification de la configuration sélectionnée. Nous réutilisons par la suite le processus

classique du *model-based testing* en se basant sur l'outil MaTeLo pour dériver les cas de test.

Chapitre 5

Extension d'OVM et du modèle d'usage pour le test des lignes de produits

Le modèle OVM est l'élément central dans le processus de MPLM. Nous fournissons des arguments pour expliquer pourquoi nous avons choisi OVM, comme approche pour modéliser la variabilité d'une ligne de produits. En outre, nous introduisons une extension de la définition formelle d'OVM, pour préparer le terrain à une définition précise de la procédure de sélection de sous-ensemble de combinatoires qui peut être soit manuelle ou automatique à l'aide d'une des approches précitées dans la section 4.1. Par suite, d'établir les liens avec les exigences fonctionnelles de la LP et notre modèle de test.

5.1 Le choix d'OVM

Comme mentionné précédemment, OVM est un modèle à plat qui définit graphiquement la variabilité d'une ligne de produits logiciels. Ce formalisme introduit par Pohl et al [PBVDL05] permet de capturer uniquement les propriétés variables de tous les produits de la même famille. L'objectif est de répondre à certaines questions concernant la variabilité telles que :

- Quels sont les éléments qui varient ?
- Comment varient-ils dans une ligne de produits logiciels ?

La variabilité définit par OVM peut être reliée à des modèles de conception, à des artefacts de développement ou encore à des modèles de test [PBVDL05]. Nous nous intéressons en particulier à la liaison avec les modèles d'exigences.

En plus de ce que nous avons présenté comme différence entre la documentation interne et orthogonale au niveau de la section 2.2.3, nous nous sommes basés sur trois arguments pour justifier le choix d'OVM en tant que formalisme pour modéliser la variabilité de la ligne de produits :

- En premier lieu, nous sommes intéressés par les libertés de modélisation offertes par OVM pour représenter la variabilité de la ligne de produits. Être indépendant d'une représentation hiérarchique donne la possibilité d'opérer plus facilement sur ce type de modèles. Par exemple, concaténer plusieurs modèles OVM, ou dans le cas d'une ligne de produits de taille importante, scinder en plusieurs sous-modèles, pour faciliter le traitement et réduire la complexité de la ligne de produits. Notamment, dans l'industrie automobile, la ligne de produits peut être la voiture, mais cette ligne de produits peut contenir aussi plusieurs lignes de produits, comme présentées dans l'exemple illustratif, une ligne de produits de tableaux de bord d'automobile.
- En deuxième lieu, OVM représente uniquement la variabilité de la ligne de produits. Les points communs de la ligne de produits sont représentés au niveau des modèles de bases auxquels les éléments du modèle OVM sont reliés. Dans ce contexte, ces artefacts logiciels sont appelés des modèles de base [PBVDL05]. Nous nous intéressons à ce type de liens, en particulier aux liens entre les *features* du modèle OVM et les modèles d'exigences. De plus, l'objectif est de se focaliser uniquement sur la variabilité, car dans notre contexte nous traitons un modèle d'usage de la ligne de produits. Ce modèle représente les scénarios de test des parties communes et variables de la ligne de produits. La dérivation des variantes du modèle d'usage consiste à identifier les éléments correspondant aux exigences de la configuration sous test.
- En dernier lieu, notre contexte est lié à une problématique réelle de test des lignes de produits en industrie, de même notre partenaire industriel *Airbus Defence and Space* utilise OVM en tant que modèle de variabilité. Par conséquent, être compatible et lié à ce qui se passe en industrie est un avantage pour encourager l'introduction de l'approche dans un projet industriel.

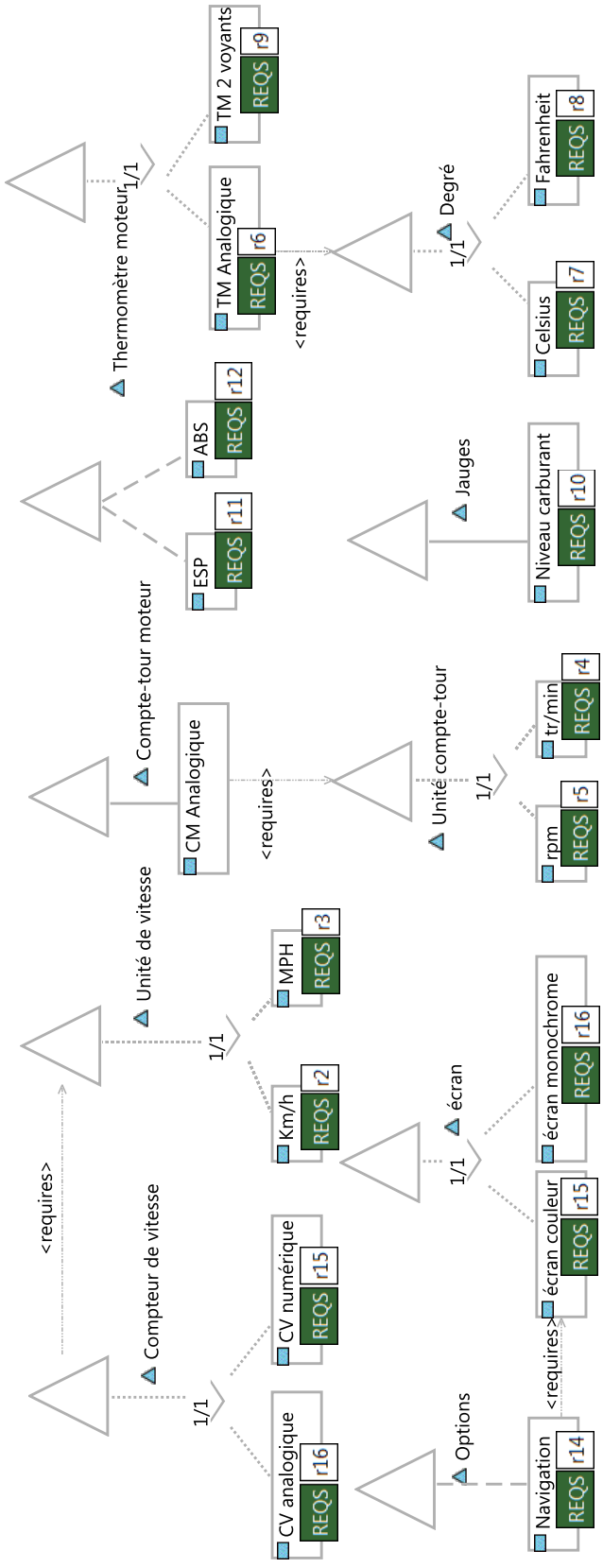


FIGURE 5.1 – *Le modèle OVM du tableau de bord automobile*

Le modèle OVM de la figure 5.1 représente l'exemple illustratif du tableau de bord. Ces éléments sont :

- Le compteur de vitesse peut être soit analogique soit numérique. La vitesse peut être exprimée soit en Km/h soit en mph.
- Un compte-tour moteur analogique utilise une unité de calcul soit en rpm ou en tr/min.
- Deux types de thermomètre : soit avec 2 voyants, soit de type analogique avec affichage de température en Celsius ou en Fahrenheit.
- Différents types d'équipements comme les voyants, les jauges, l'écran et les options.

5.2 Extension de la définition formelle d'OVM

Metzger et al. [MPH+07] ont introduit une formalisation mathématique pour OVM, qui décrit les éléments de base en dehors du métamodèle d'OVM [PBVDL05]. Nous avons étendu cette formalisation pour pouvoir la réutiliser dans notre contribution, afin d'exprimer d'une manière formelle les liens entre les *features* et les exigences de la ligne de produits.

Définition 5.1 *Soit un modèle OVM noté $\mathcal{M}_{\mathcal{V}}$ et représenté par le tuple $(VP, V, VG, Parent, Min, Max, Opt, Req, Excl, \mathcal{P}(V), C, \delta)$, où :*

Définition de base d'OVM

- *$(VP, V, VG, Parent, Min, Max, Opt, Req, Excl)$, équivalent à la formalisation de base d'OVM présentée dans la section 2.2.2.3*

Notre extension

- *vp, est un point de variation, où $vp \in VP$.*
- *v, est une feature[a], où $v \in V$.*
- *$\mathcal{P}(V) = \{V' | V' \subseteq V\}$ l'ensemble des parties de l'ensemble V, $\mathcal{P}(V)$ désigne l'ensemble des sous-ensembles de V, y compris l'ensemble vide et l'ensemble V lui-même.*
- *$C = Req \cup Excl$, ensemble des contraintes, où $c \in Req$ ou $c \in Excl$,*
- *$\delta : V \nrightarrow RG$, la fonction delta associe un sous-ensemble d'exigences à chaque variante (l'ensemble d'arrivée RG est l'ensemble des parties de toutes les exigences, cf. section 3.5.2)*

a. Dans la terminologie d'OVM *v* est désigné comme une "variante". Dans le présent document, nous utilisons *feature* (au lieu de variante) pour désigner un discriminant, caractéristique visible par l'utilisateur d'un système ; Le terme *variante* est utilisé pour désigner une configuration.

Un modèle OVM permet d'identifier un ensemble de combinaisons valides de *features* qui composent un produit d'une famille de produits. Un produit, appelé aussi une configuration, ou une variante, représente une expression de *features* qui respectent les contraintes et les relations définies dans le modèle de variabilité.

Définition 5.2 *Un produit ou une variante est une configuration valide composée d'un ensemble de features qui respecte un ensemble de contraintes défini dans $\mathcal{M}_{\mathcal{V}}$. Un produit noté $\mathcal{P}$ est un tuple de la forme $(V_{\mathcal{P}}, R_{\mathcal{P}}, \mathcal{M}_{\mathcal{T}_{\mathcal{P}}})$ utilisé pour identifier $\mathcal{M}_{\mathcal{T}_{\mathcal{P}}}$ à partir de $\mathcal{M}_{\mathcal{T}}$, où :*
– *$V_{\mathcal{P}}$ un ensemble de features qui composent $\mathcal{P}$,*
– *$R_{\mathcal{P}}$ défini des exigences relatives à $\mathcal{P}$,*
– *$\mathcal{M}_{\mathcal{T}_{\mathcal{P}}}$ une variante du modèle d'usage.*
– *$\mathbf{P} = \{\mathcal{P}_1, ..., \mathcal{P}_n\}$ représente les variantes de la ligne de produits.*

La formalisation proposée permet d'exprimer la composition d'une variante choisie pour dériver le modèle d'usage correspondant, afin de générer automatiquement la suite de tests.

Nous avons choisi d'identifier les éléments de $\mathcal{M}_{\mathcal{V}}$ (voir figure 5.1), comme suit :

L'ensemble VP est composé de :
– vp_1 : Compteur de vitesse ;
– vp_2 : Unité de vitesse ;
– vp_3 : Compte-tour moteur ;
– vp_4 : Voyants ;
– vp_5 : Thermomètre moteur ;
– vp_6 : Options ;
– vp_7 : Écran ;
– vp_8 : Unité compte-tour ;
– vp_9 : Jauges ;
– vp_{10} : Degré.

L'ensemble V est composé de :
– v_1 : CV analogique ;
– v_2 : CV numérique ;
– v_3 : Km/h ;
– v_4 : Mph ;
– v_5 : TM Analogique ;
– v_6 : TM 2 voyants ;
– v_7 : CM analogique ;
– v_8 : rpm ;
– v_9 : tr/min ;
– v_{10} : Navigation ;
– v_{11} : ESP ; v_{12} : ABS ;
– v_{13} : Écran couleur ;
– v_{14} : Écran monochrome ;
– v_{15} : Celsius ;
– v_{16} : Fahrenheit ;
– v_{17} : Niveau carburant.

VG est défini comme suit : $VG = \{\emptyset, \{v_1\}, \{v_2\}, ..., \{v_1, v_2, ..\}, ..., V\}$

Ces configurations doivent respecter les contraintes C définies au niveau du modèle OVM, C est composé de :

– c_1 : v_{10} *Requires* v_{13} ;
– c_2 : vp_1 *Requires* vp_2 ;
– c_3 : v_7 *Requires* vp_8 ;
– c_4 : v_5 *Requires* vp_{10}.

À partir du modèle OVM de l'exemple, nous pouvons extraire différentes configurations possibles (ou produit noté $\mathcal{P}$) et valides. Nous avons sélectionné deux configurations pour le test parmi l'ensemble des possibilités, qui sont :

$\mathcal{P}_1$ **est un produit haut de gamme destiné à l'Europe** : compteur de vitesse numérique en Km/h, compte-tour moteur en tr/min, thermomètre moteur analogique en Celsius, ESP, ABS, jauge niveau carburant, écran couleur, navigation.

$\mathcal{P}_2$ **est un produit haut de gamme destiné aux États-Unis** : compteur de vitesse numérique en mph, compte-tour moteur en rpm, thermomètre moteur analogique en Fahrenheit, ESP, ABS, jauge niveau carburant, écran couleur, navigation.

5.3 Extension de la définition du modèle d'usage

Dans ce chapitre, nous présentons une adaptation du *model-based testing* pour le test d'une famille de produits. La méthode consiste à modéliser la variabilité d'une ligne de produits, configurer manuellement les variantes à tester et à réaliser le modèle de test de la ligne de produits. Nous proposons pour notre approche, de modéliser en un seul modèle d'usage les scénarios de test des parties communes et variables de la ligne de produits. Cette approche est adoptée et utilisée par certains clients d'ALL4TEC. Cela résulte de l'évolution des systèmes sous test (SUT) de ces clients. Les praticiens ont commencé par des systèmes uniques, ces derniers ont évolué vers des systèmes hautement configurables. Le modèle de test a évolué de la même manière.

Les techniques du MBT implémentées par MaTeLo sont en mesure de générer un ensemble fini de cas de test, en sélectionnant les exécutions représentatives du modèle d'usage du SUT, jusqu'à ce que certains critères de couverture soient remplis.

Notre objectif est d'introduire la variabilité dans le processus du *model-based testing*. Pour cela, le modèle doit être capable de représenter le comportement attendu des *features* de la ligne de produits. Cependant, tester une ligne de produits à l'aide de l'approche model-based testing est très différent de l'approche traditionnelle connue. En effet, les *features* sont des entités abstraites qui décrivent certaines caractéristiques de la ligne de produits et déterminent si ces entités sont communes à tous les produits ou sont des entités variables. Donc, le modèle correspondant ne peut représenter un système valide à tester et ne peut être utilisé pour générer automatiquement une suite de tests.

Pour pouvoir réutiliser le processus classique du MBT, nous voulons encourager la génération de cas de test à partir des variantes du modèle d'usage. Ceux-ci sont dérivés automatiquement à partir d'un seul et unique modèle d'usage de la ligne de produits et surtout en respectant les exigences suivantes :

– Associer les exigences aux éléments du modèle d'usage.

– La mise en correspondance des *features* avec les transitions du modèle d'usage.

Les variants du modèle d'usgae dérivées vont utiliser les algorithmes standards de MaTeLo pour la génération automatique de cas de test et s'appuyer sur les critères standards de couverture des exigences.

Le modèle d'usage de la ligne de produits doit être réalisé en ingénierie du domaine pour deux raisons :

– Le modèle d'usage du domaine est utilisé pour faciliter et effectuer une validation anticipée des exigences du domaine de la ligne de produits.
– Un seul modèle d'usage est créé pour le domaine. Ce modèle qui contient la variabilité est adapté ensuite, aux besoins spécifiques des variantes. Ainsi, les cas de test sont créés pour une réutilisation pendant l'ingénierie d'application.

Nous avons privilégié une méthodologie qui se base sur la dérivation automatique des variantes du modèle d'usage qui correspondent aux spécifications des produits à tester, pour deux raisons :

– Les modèles de test et les cas de test sont utilisés pour documenter les activités de validations des systèmes logiciels. Cela est nécessaire lorsque les clients ou les lois et les normes exigent une preuve que les exigences ont été testées avec succès.
– Un modèle de test d'un produit est nécessaire pour générer les cas de test correspondants.

En résumé, cette méthodologie de test permet de générer automatiquement des cas de test pour chaque produit de la ligne de produits à partir des variantes du modèle d'usage de la ligne de produits entière, dérivées automatiquement. Ce processus offre la possibilité de profiter de l'efficacité du *model-based testing* et d'utiliser les critères de couverture standard et les algorithmes de génération de cas de test standard, ainsi donc d'utiliser les outils existant, comme l'outil MaTeLo.

Dans ce qui suit, nous allons présenter la construction d'un modèle d'usage réutilisable. Ensuite, nous détaillons quelques points au niveau du modèle d'usage de MaTeLo. Enfin, nous allons discuter de possibilités et les limites de cette philosophie de modèle d'usage global pour une ligne de produits entière.

5.4 Formalisation du modèle d'usage de la ligne de produits

Nous étendons la **syntaxe du modèle d'usage de MaTeLo** introduite par Le Guen [LG05].

Définition 5.3 *Le modèle d'usage de la ligne de produits noté $\mathcal{M}_{\mathcal{T}}$, est un tuple de la forme $(S, s_0, s_n, T, V, F, R, \mathcal{P}(R), T, \mathcal{P}(T), \gamma)$, où :*
- *$(S, s_0, s_n, T, V, F, R)$, équivalent à $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ décrit dans la section 3.5.2,*
- *$\mathcal{P}(R) = \{R' | R' \subseteq R\}$, $\mathcal{P}(R)$ ensemble des parties de l'ensemble R,*
- *$\mathcal{P}(T) = \{T' | T' \subseteq T\}$, $\mathcal{P}(T)$ ensemble des parties de l'ensemble T.*
- *$\gamma : R \nrightarrow \mathcal{P}(T)$ est une fonction partielle d'étiquetage des exigences avec les transitions.*

Ces relations ne sont vraies que pour le modèle d'usage d'une ligne de produits. Pour réaliser un modèle d'usage valide, il faut respecter les règles de construction présentées dans la section 3.5.2. Dans la suite, nous considérons un unique profil $\mathfrak{F}$ associé au modèle d'usage de la ligne de produits. Le profil $\mathfrak{F}$ est indépendant des configurations de la ligne de produits décrites par le modèle d'usage pour de multiples variantes. Il est utilisé pour avoir une structure valide du modèle d'usage et aider à l'extraction du modèle d'usage spécifique à une variante avec son propre profil.

Définition 5.4 *Sémantique du modèle d'usage de la ligne de produits.*
Chaque $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ dérivé décrit l'ensemble des séquences d'utilisations (voir Section 3.5.2) d'une variante. La sémantique d'un $\mathcal{M}_{\mathcal{T}}$ est donc, l'union des utilisations de toutes les configurations valides :

$$\mathcal{M}_{\mathcal{T}} = \bigcup_{\mathcal{P} \in \mathbf{P}} [\![\mathcal{M}_{\mathcal{T}_\mathcal{P}}]\!]$$

La formalisation proposée décrit la sémantique du modèle d'usage pour une ligne de produits, à partir duquel, nous allons dériver des modèles d'usages spécifiques aux variantes de la ligne de produits.

La figure 5.2 représente le modèle d'usage de l'exemple *tableau de bord* avec ses sous-chaînes. Le modèle représente le comportement attendu des *features* de la ligne de produits, où le comportement dynamique peut être visible au niveau des transitions du modèle ou au niveau des données d'entrée et des résultats attendus, associés à une transition. Par exemple, la fonction ESP peut référencer une sous-chaîne qui représente son comportement attendu. Les cas de test de l'exemple peuvent concerner une fonctionnalité du tableau de bord ou plusieurs fonctionnalités.

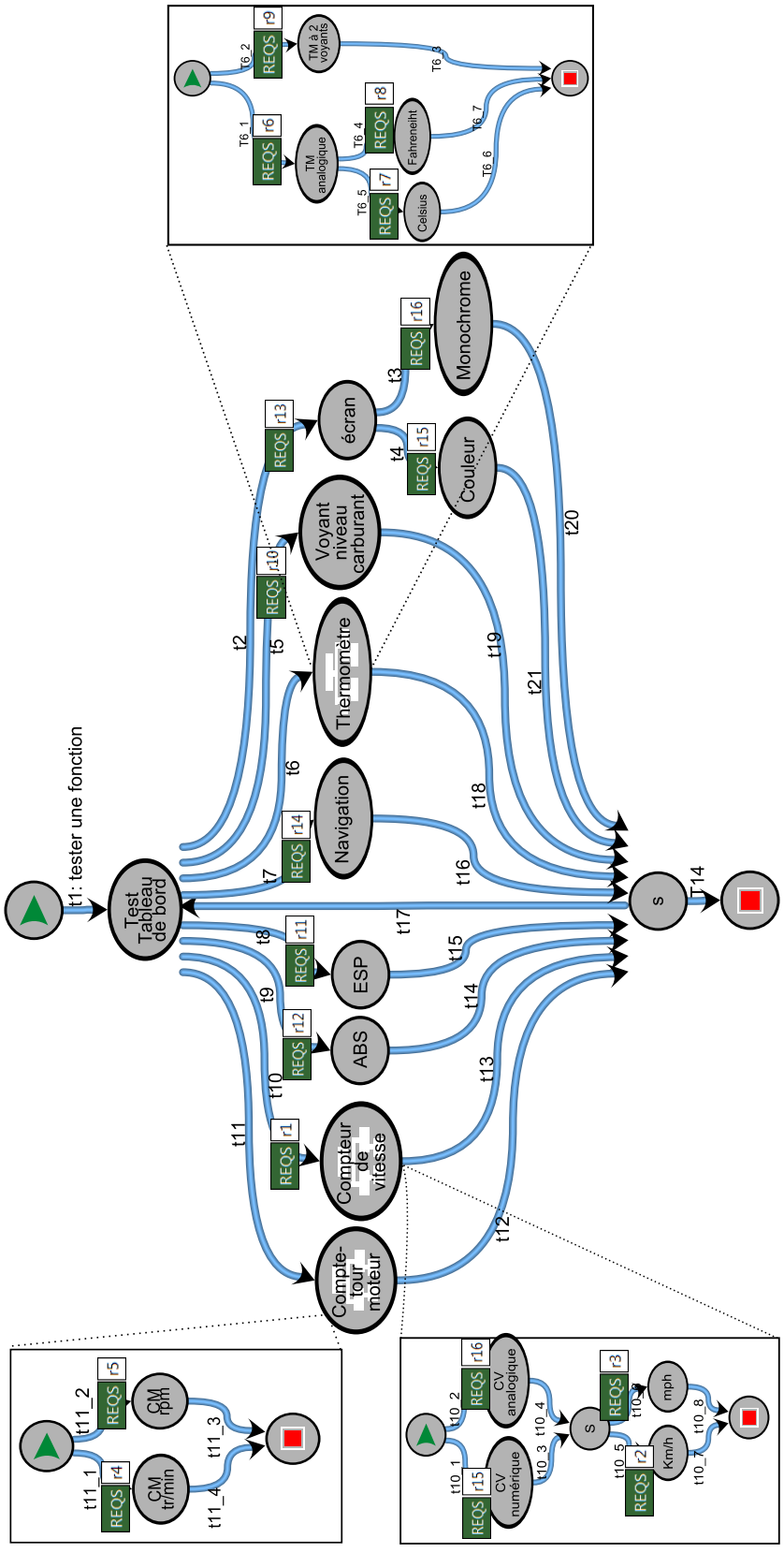


FIGURE *5.2 – Modèle d'usage de la ligne de produits de tableaux de bord automobile avec ses sous-chaînes*

Ce modèle représente les exigences des produits de la catégorie haut de gamme et bas de gamme.

Le modèle d'usage contient plusieurs niveaux hiérarchiques, chaque niveau est une chaîne. L'ensemble T=$\{t_1, ..., t_{6\_7}\}$ est la liste des transitions du $\mathcal{M}_{\mathcal{T}}$ qui sont annotées par la liste des exigences fonctionnelles suivantes :

r_1 : Le compteur de vitesse (CV) lit la vitesse à la sortie de la boîte de vitesse ou directement sur une roue à l'aide d'un capteur.

r_2 : Le CV pour l'Europe doit afficher la vitesse instantanée du véhicule en Km/h.

r_3 : Le CV pour les États-Unis doit afficher la vitesse instantanée du véhicule en mph.

r_4 : Le compte-tour moteur (CM) pour l'Europe doit indiquer en temps réel la fréquence de rotation du moteur en tour par minute. Cette information est obtenue grâce à un capteur sur le vilebrequin.

r_5 : Le CM pour les États-Unis doit indiquer en temps réel la fréquence de rotation du moteur en rpm.

r_6 : L'aiguille du thermomètre moteur (TM) doit toujours être dans une position intermédiaire pour une température normale. Un capteur de température placé sur le bloc moteur ou la culasse fournit la température.

r_7 : Le TM pour l'Europe doit afficher la température instantanée du moteur en C°.

r_8 : Le TM pour les États-Unis doit afficher la température instantanée du moteur en F°.

r_9 : Le TM à base de 2 voyants doit allumer le voyant bleu quand le moteur est froid, le voyant rouge quand il est trop chaud et tous les voyants doivent être éteints quand tout est normal. Un capteur de température placé sur le bloc moteur ou la culasse fournit la température.

r_{10} : La jauge doit indiquer le niveau de carburant dans le réservoir.

r_{11} : L'ESP doit se déclencher lors d'un changement de trajectoire anormal ou un dysfonctionnement.

r_{12} : Le voyant ABS doit s'allumer durant 5 à 6 secondes lorsque la clé de contact est mise sur ON et s'éteindre si tout est normal.

r_{13} : L'écran doit permettre de paramétrer les options du tableau de bord.

r_{14} : La navigation doit être capable de récupérer la position du véhicule et d'assister le conducteur par la voix et des indications visuelles.

r_{15} : Le tableau de bord HG utilise l'affichage numérique et un écran couleur pour la navigation.

r_{16} : Le tableau de bord BG utilise l'affichage analogique et un écran monochrome.

Les caractéristiques du modèle d'usage :

Nombre des exigences : 16,

Taille du modèle de variabilité : 5 points de variation et 15 *features*,

Taille du modèle de test : 21 états, 40 transitions, 3 instances de chaînes,

Nombre de produits configurés : 4,

En résumé, les transitions et ses données associées (stimulation et résultats attendus) représentent les éléments du modèle d'usage ou nous pouvons associer des exigences. Celles-ci sont également liées aux *features* du modèle de variabilité. En effet, les liens établis entre les *features* et les exigences vont nous aider à extraire les variantes du modèle d'usage d'une ligne de produits. Nous décrirons dans le chapitre suivant, comment les liens sont établis entre les *features* et les transitions du modèle d'usage grâce aux exigences de la ligne de produits.

5.5 Résumé

Nous avons étendu la définition formelle d'OVM pour inclure les exigences et décrire leur relation avec les *features*. Nous n'avons pas proposé une méthodologie de sélection de configurations, car il n'est pas le sujet de cette thèse. Nous estimons que notre méthodologie peut être compatible avec d'autres approches existantes, par exemple FAMA-OVM [RFBRC$^+$12]. Cette possibilité est discutée dans la section perspective.

Nous introduisons des adaptations au processus du *model-based testing* pour le test des lignes de produits, car utiliser le MBT dans l'état, cela revient à concevoir pour chaque variante un modèle d'usage correspondant. En plus, répéter ce processus pour toutes les variantes de la ligne de produits est un travail lourd et difficile à gérer, non seulement, cela nous éloigne du bénéfice attendu d'adoption du MBT. En outre, cette démarche n'entre pas dans les principes de l'ingénierie des lignes de produits qui encourage la réutilisation. Nous proposons de réaliser un seul modèle d'usage pour la ligne de produits représentant les scénarios de test des *features* à tester dans une famille de produits, puis de dériver des variantes du modèle d'usage.

Pendant l'ingénierie du domaine, pour équiper les *features* d'un certain comportement, l'association des artefacts appropriés tels que des fragments de code et des modèles de test est nécessaire. Cela est indispensable pour extraire les modèles de test spécifiques aux configurations de la ligne de produits, qui sont ensuite utilisés pour générer des cas de test de configurations spécifiques. Un modèle d'usage d'une variante spécifique décrivant son comportement est désigné en tant qu'un modèle d'usage traditionnel.

Nous décrivons dans le chapitre suivant, la mise en correspondance entre le modèle OVM, le modèle d'usage d'une ligne de produits et les exigences fonctionnelles, ensuite nous décrivons le processus de dérivation automatique des variantes du modèle d'usage.

Chapitre 6

Dérivation des variantes du modèle d'usage

Dans ce chapitre, nous introduisons notre approche de mise en correspondance (*mapping*) entre les *features*, les exigences et les transitions du modèle d'usage. L'objectif est d'identifier les éléments du modèle d'usage qui correspondent aux *features* utilisées pour spécifier les membres de la ligne de produits. Nous présentons également la deuxième partie de notre contribution : la dérivation automatique des variantes du modèle d'usage. Les liaisons entre les *features*, les exigences et les transitions du modèle d'usage de la ligne de produits vont servir pour dériver des variantes du modèle d'usage avec seulement les transitions et les exigences d'une variante sous test.

6.1 Mapping des *features* avec les exigences

Au cours de la réalisation du modèle d'usage, les éléments du modèle sont étiquetés par des exigences. Une exigence peut être associée à plusieurs éléments et à différents endroits dans le modèle d'usage. Par exemple, le tableau 6.2 présente un exemple d'association entre l'exigence r_{16} et les transitions t_3 et t_{10_2}.

Pour concilier le modèle OVM avec le modèle d'usage, nous supposons que les *features* sont également étiquetées par des exigences, il est possible d'associer à une exigence à plusieurs *features*. Ce *mapping* va donner une sémantique aux *features* du modèle OVM.

Lors de l'association exigences - *features*, nous rencontrons trois différents types d'exigences :

- Une exigence correspondant aux points communs dans la ligne de produits, comme r_1. Cette exigence est généralement associée aux transitions qui représentent le point de variation vp_3 (compte-tour moteur) et non à ses *features*. Cette exigence est considérée comme une exigence générique et ne peut être associée à aucune *feature*.
- Une exigence relative à une seule *feature*, comme r_{14} associée à la *feature* navigation.

– Une exigence correspondant à plusieurs *features*, comme le cas de r_{15} associée à la *feature* v_2 (CV analogique) et à la *feature* v_{13} (écran monochrome). Cette exigence est traitée de manière spécifique (cf. section 6.2.3).

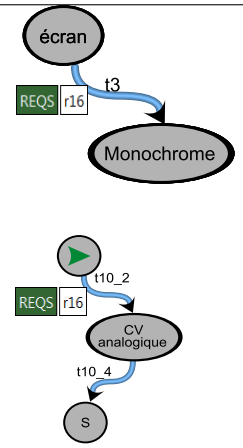
Exigences	Transitions du modèle d'usage de la ligne de produits
REQS r16	

TABLE *6.1 – Mapping des exigences et des transitions du modèle d'usage*

Exigences	*Features*
REQS r16	écran monochrome

TABLE *6.2 – Mapping des exigences et les features du modèle OVM*

Après la configuration et la sélection de l'ensemble de produits pour le test à partir des features du modèle OVM, nous allons pour chaque produit, extraire les exigences auxquelles ses *features* sont associées et identifier les éléments du modèle d'usage correspondant. Cette opération est basée sur les fonctions partielles définies auparavant au niveau de la sémantique de $\mathcal{M}_{\mathcal{V}}$ (cf. section 5.2) et de $\mathcal{M}_{\mathcal{T}}$ (cf. section 5.3). À l'aide de γ et δ, l'ensemble des exigences qui correspond à chaque variante v_i est identifié, afin de récupérer par la suite pour chaque exigence r_i l'ensemble des transitions annotées par celle-ci. Grâce aux transitions sélectionnées nous pouvons extraire à partir de $\mathcal{M}_{\mathcal{T}}$ une variante du modèle d'usage $\mathcal{M}_{\mathcal{T}_{\mathcal{P}}}$ spécifique au produit $\mathcal{P}$ qui est étiqueté par les exigences du produit $\mathcal{P}$ et qui contient les transitions correspondantes.

La figure 6.2 représente l'idée principale de notre méthodologie de relier le modèle de variabilité OVM avec le modèle d'usage. Ainsi donc, lors de la configuration d'un produit, pour chaque *feature* qui compose ce produit à tester, un ensemble d'exigences correspondantes est identifié et les éléments du modèle de test étiquetés par cet ensemble sont identifiés et sélectionnés.

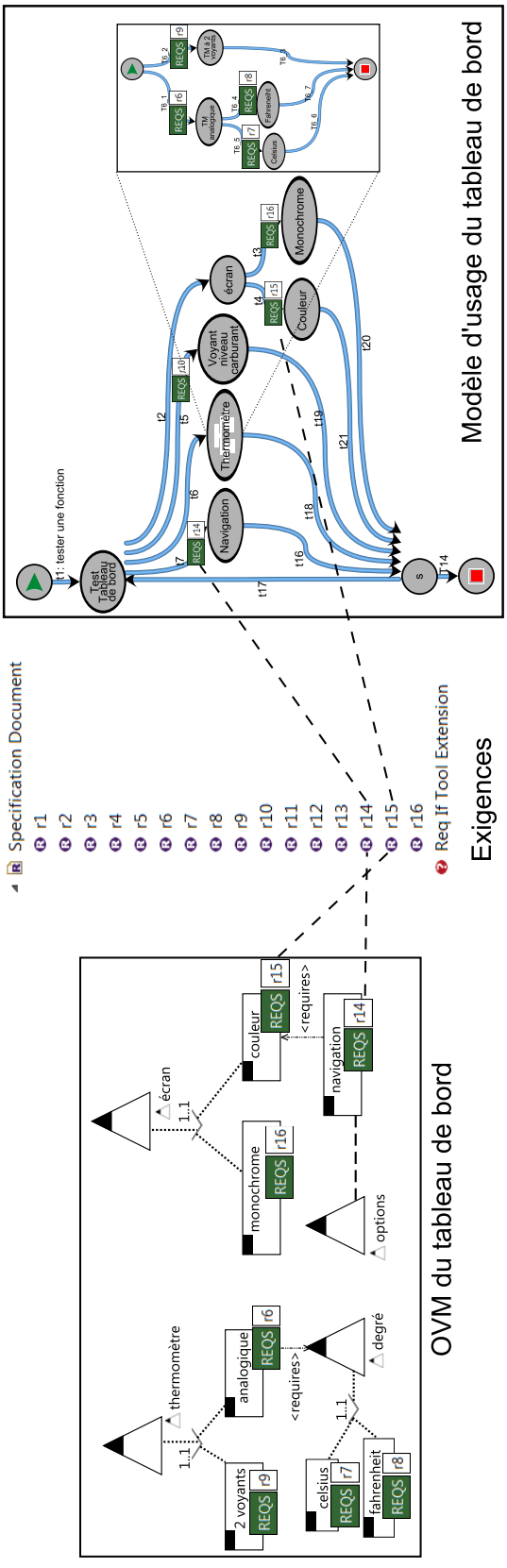


FIGURE *6.1 – Processus de réconciliation du modèle de variabilité $\mathcal{M}_{\mathcal{V}}$ avec le modèle d'usage de la ligne de produits $\mathcal{M}_{\mathcal{T}}$*

6.2 Processus de dérivation des variantes du modèle d'usage

Dans notre approche, nous considérons quatre étapes pour atteindre la dérivation automatique des variantes du modèle d'usage :

Step A Identifier et sélectionner les *features* qui composent la configuration sous test.

Step B Extraire et classifier selon les *features* choisies les exigences à conserver et à supprimer.

Step C Identifier et sélectionner selon les exigences correspondantes, les transitions à conserver et identifier les cas incohérents.

Step D Dériver des variantes du modèle d'usage à partir du modèle d'usage de la ligne de produits en supprimant les transitions non mappées directement à la configuration de la variante sélectionnée pour le test.

Step E Ajuster les probabilités associées aux profils d'usage du modèle d'usage généré, tel que défini dans la section 3.5.

6.2.1 Illustration

A titre d'illustration, nous allons décrire l'algorithme d'extraction à l'aide du produit haut de gamme $\mathcal{P}_1$ (cf. section 5.2). L'algorithme de dérivation prend en entrée le modèle d'usage $\mathcal{M}_\mathcal{T}$, le modèle OVM $\mathcal{M}_\mathcal{V}$ et dans notre cas, nous avons choisi de l'appliquer sur le produit $\mathcal{P}_1$. Pour rappel $\mathcal{P}_1$ est composé de :

$V_{\mathcal{P}_1}$=$\{v_2, v_3, v_5, v_7, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{15}, v_{17}\}$,

$C_{\mathcal{P}_1}$=$\{c_1, c_2, c_3, c_4\}$,

$R_{\mathcal{P}_1} = \{r_2, r_4, r_6, r_7, r_{10}, r_{11}, r_{12}, r_{14}, r_{15}\}$.

L'association entre les *features* du produit $\mathcal{P}_1$ et les exigences qui les couvrent est réalisée manuellement dans un outil dédié que nous présentons dans le chapitre 7.1.

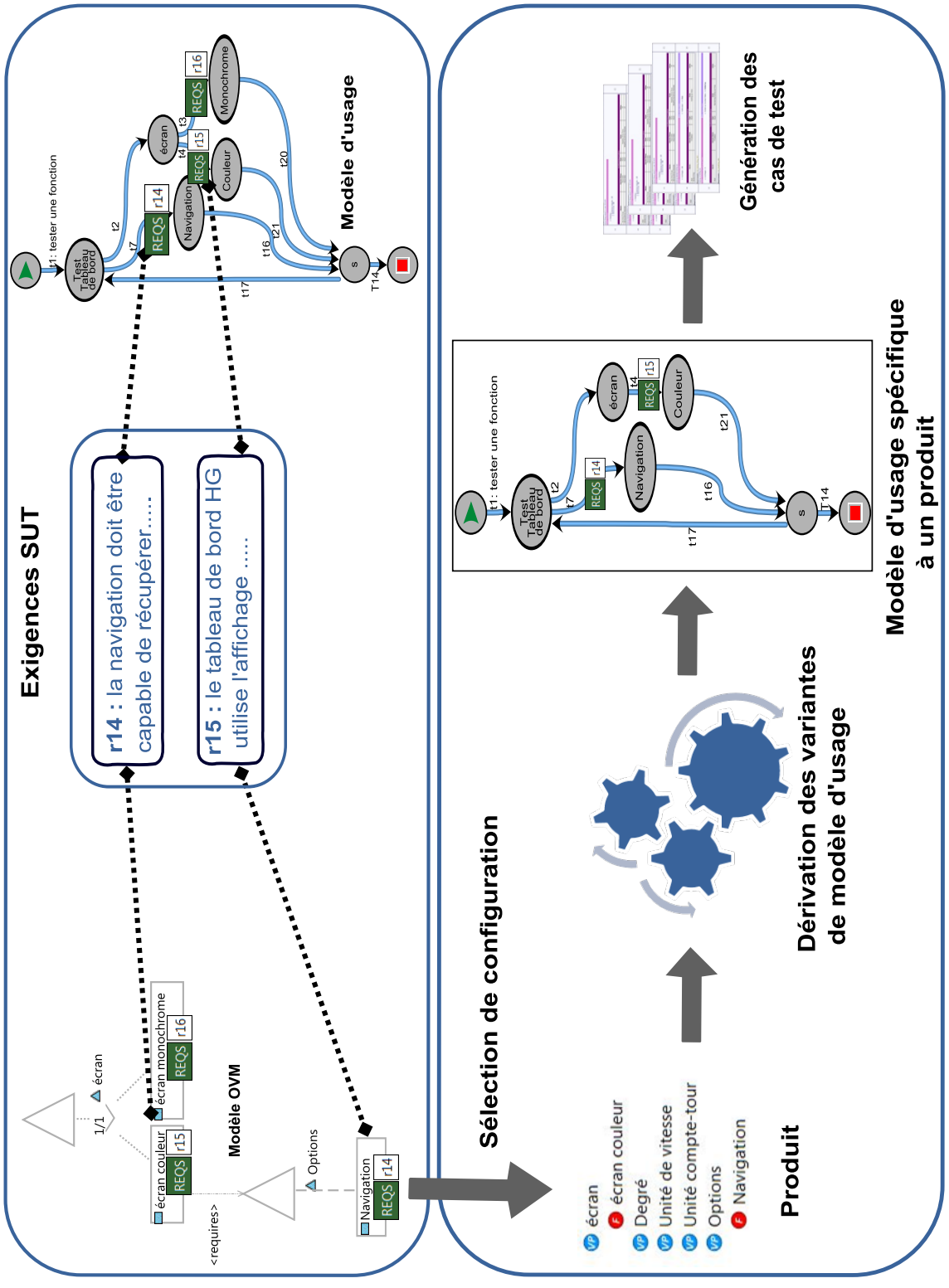


FIGURE 6.2 – *Le processus global du concept de MPLM*

Dans l'exemple illustré par le premier bloc de la figure 6.2 nous avons choisi une sous partie du modèle OVM. Deux points de variation ont été sélectionnés, les options du tableau de bord et le type d'écrans. L'ensemble des *features* qui correspond aux points de variation sélectionnés est $V' = \{v_{10}, v_{13}, v_{14}\}$, couverts par $R' = \{r_{14}, r_{15}, r_{16}\}$. Le comportement lié à V' est représenté durant la génération automatique des cas de test par l'ensemble des transitions $T' = \{t_7, t_{10\_1}, t_{10\_2}\}$. Par exemple pour obtenir l'ensemble des transitions qui correspond à la *feature* v_{14}, en appliquant $\delta(v_{14})$ nous obtenons l'ensemble des exigences correspondant $\{r_{16}\}$. De même avec $\gamma(r_{16})$ nous obtenons l'ensemble des transitions $\{t_{10\_2}\}$ qui correspond à l'exigence r_{16} associée à v_{14}. Pour le reste des associations exigences - *features* (voir figure 5.1).

6.2.2 Step A - Identification des *features* pour le test

La première étape consiste à identifier les *features* qui ne doivent pas être prises en compte pour la variante du modèle d'usage dérivée. Cela suppose, qu'il faut sélectionner toutes les *features* du modèle OVM à l'exception des *features* sélectionnées composant la variante sous test. Cela revient à identifier $V_{\mathcal{P}}$, l'ensemble des *features* de $\mathcal{P}$. À partir de $V_{\mathcal{P}}$ nous pouvons identifier les *features* qui ne concernent pas $\mathcal{P}$.

Soit $V_{\bar{\mathcal{P}}}$ l'ensemble des *features* qui ne sont pas associées à $\mathcal{P}$.

$$V_{\bar{\mathcal{P}}} = V \setminus V_{\mathcal{P}} \tag{6.1}$$

Grâce à la fonction de traçabilité δ définie dans la section 5.2, nous pouvons extraire les exigences correspondantes pour les deux $V_{\mathcal{P}}$ et $V_{\bar{\mathcal{P}}}$ et ainsi identifier les éléments du modèle d'usage correspondant à $\mathcal{P}$.

6.2.3 Step B - Classification des ensembles d'exigences

Cette étape élague l'ensemble R, pour déterminer les exigences qui couvrent la variante $\mathcal{P}$ et les exigences qui ne la couvrent pas. R est composé de R_V un ensemble d'exigences lié à la variabilité de la ligne de produits et $R_{\bar{V}}$ un ensemble d'exigences associé aux points communs.

$$R = R_V \cup R_{\bar{V}} \tag{6.2}$$

Dans l'exemple, $R_{\bar{V}}$ correspond à $\{r_1, r_{13}\}$ et à la partie variable
$R_V = \{r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, r_{14}, r_{15}, r_{16}\}$.
L'étape suivante consiste à affiner R_V, pour identifier les exigences à conserver ou non et les exigences communes aux deux ensembles.

$$R_V = R_{V\_\mathcal{P}} \cup R_{V\_\bar{\mathcal{P}}} \tag{6.3}$$

Où $R_{V\_\mathcal{P}}$ est l'ensemble des exigences qui étiquette $V_{\mathcal{P}}$ tandis que $R_{V\_\bar{\mathcal{P}}}$ est l'ensemble des exigences qui étiquette $V_{\bar{\mathcal{P}}}$.

Néanmoins, certaines exigences peuvent appartenir à $R_{V\_\mathcal{P}}$ et en même temps à $R_{V\_\bar{\mathcal{P}}}$. R_{Inter} représente les exigences communes entre les variantes, où :

$$R_{Inter} = R_{V\_\mathcal{P}} \cap R_{V\_\bar{\mathcal{P}}} \tag{6.4}$$

Dans l'exemple, à partir des produits définis dans la section 5.2. R_{Inter} correspond à $\{r_6, r_{10}, r_{11}, r_{12}, r_{14}, r_{15}\}$.

L'identification de R_{Inter}, va servir à affiner $R_{V\_\mathcal{P}}$ et $R_{V\_\bar{\mathcal{P}}}$, afin de ne pas supprimer les éléments du modèle d'usage qui doivent être conservés pour le produit $\mathcal{P}$ et aider à la détection des cas incohérents qui peuvent figurer dans le modèle d'usage de la ligne de produits.

$$R_{\|V\_\mathcal{P}\|} = R_{V\_\mathcal{P}} \setminus R_{Inter} \tag{6.5}$$

$R_{\|V\_\mathcal{P}\|}$, l'ensemble des exigences qui couvre uniquement les *features* spécifiques du produit sélectionné pour le test.

$$R_{\|V\_\bar{\mathcal{P}}\|} = R_{V\_\bar{\mathcal{P}}} \setminus R_{Inter} \tag{6.6}$$

$R_{\|V\_\bar{\mathcal{P}}\|}$, l'ensemble des exigences qui n'est pas associé à $\mathcal{P}$. Cela permet d'identifier les éléments du modèle d'usage à supprimer de manière certaine et qui sont associés à cet ensemble d'exigences.

Dans l'exemple, pour $\mathcal{P}_1$, $R_{\|V\_\mathcal{P}\|}$ est égal à $\{r_2, r_4, r_7\}$ et $R_{\|V\_\bar{\mathcal{P}}\|}$ est égal à $\{r_3, r_5, r_8, r_9, r_{16}\}$.

Les opérations d'affinage pour classifier l'ensemble des exigences telles que définies ci-dessus, sont réalisées à l'aide de l'algorithme 1. L'objectif est d'identifier R_{Inter}, ainsi que $R_{\|V\_\mathcal{P}\|}$ et $R_{\|V\_\bar{\mathcal{P}}\|}$.

Entrées: $V_{\mathcal{P}}$, $V_{\bar{\mathcal{P}}}$
Sorties: $R_{\|V\_\mathcal{P}\|}$, $R_{\|V\_\bar{\mathcal{P}}\|}$, R_{Inter}
début
  /* *Construction de* $R_{V\_\mathcal{P}}$ */
  pour chaque v ***de*** $V_{\mathcal{P}}$ **faire**
    $R' \longleftarrow \delta(v)$;
    si $R' \nsubseteq R_{V\_\mathcal{P}}$ **alors**
      $R_{V\_\mathcal{P}} \longleftarrow R_{V\_\mathcal{P}} \cup R'$;
    fin
  fin
  /* *Construction de* $R_{V\_\bar{\mathcal{P}}}$ */
  pour chaque v ***de*** $V_{\bar{\mathcal{P}}}$ **faire**
    $R' \longleftarrow \delta(v)$;
    si $R' \nsubseteq R_{V\_\bar{\mathcal{P}}}$ **alors**
      $R_{V\_\bar{\mathcal{P}}} \longleftarrow R_{V\_\bar{\mathcal{P}}} \cup R'$;
    fin
  fin
  /* *Construction de* R_{Inter} */
  $R_{Inter} \longleftarrow R_{V\_\mathcal{P}} \cap R_{V\_\bar{\mathcal{P}}}$;
  $R_{\|V\_\mathcal{P}\|} \longleftarrow R_{V\_\mathcal{P}} \setminus R_{Inter}$;
  $R_{\|V\_\bar{\mathcal{P}}\|} \longleftarrow R_{V\_\bar{\mathcal{P}}} \setminus R_{Inter}$;
fin

Algorithme 1: *Classification des ensembles d'exigences*

Analyse de la complexité de l'algorithme 1

Le problème consiste à identifier les exigences liées à chaque *feature* qui compose le produit. Nous avons choisi une recherche linéaire : utiliser la fonction $\delta(v)$ pour récupérer R' associé à v'. Pour cette méthode, le meilleur cas et le pire cas n'existent pas car, la configuration d'un produit est connue. D'ailleurs, l'utilisation d'une méthode linéaire aide à optimiser cette opération. Ainsi, l'ordre de grandeur de cette méthode est de $O(nm)$, où n est le nombre de *features* du produit $\mathcal{P}$ et m le nombre d'exigences.

6.2.4 Step C - Classification des transitions du $\mathcal{M}_{\mathcal{T}}$

Nous rappelons qu'une transition de modèle d'usage peut être étiquetée par plusieurs exigences, de même qu'une exigence peut être associée à plusieurs éléments du modèle d'usage. L'algorithme 2 sélectionne trois types de classes de transitions :

Sélection des transitions à supprimer T_S :

Cette étape permet d'identifier les transitions à supprimer de manière définitive. Elle consiste à identifier pour $R_{\|V\_\bar{\mathcal{P}}\|}$ les transitions du modèle d'usage liées à cet ensemble.

En d'autres termes, chaque transition étiquetée par le sous-ensemble d'exigences R', qui remplit la condition suivante, doit être supprimée :

$$R' \subseteq R_{\|V\_\bar{\mathcal{P}}\|} \tag{6.7}$$

Dans le cas où toutes les exigences de la ligne de produits appartiennent à $R_{\|V\_\bar{\mathcal{P}}\|}$, c'est-à-dire que le produit sous test n'est pas associé aux exigences qui annotent le modèle d'usage de la ligne de produits, la génération n'est pas possible, car l'algorithme va supprimer toutes les transitions. Pour rappel, $R_{\|V\_\bar{\mathcal{P}}\|}$ est égal à $\{r_3, r_5, r_8, r_9, r_{16}\}$, donc $T_S = \{t_3, t_{6\_2}, t_{6\_4}, t_{10\_2}, t_{10\_6}, t_{11\_2}\}$.

Sélection des transitions à conserver T_G :

L'identification des transitions à conserver T_G, consiste à identifier les transitions associées à $R_{V\_\mathcal{P}}$ et à les ajouter dans T_G. En d'autres termes, chaque transition étiquetée par le sous-ensemble d'exigences R', qui remplit la condition suivante, doit être conservée :

$$R' \subseteq R_{V\_\mathcal{P}} \tag{6.8}$$

Pour rappel, $R_{V\_\mathcal{P}}$ est l'union de $R_{\|V\_\mathcal{P}\|}$ et R_{Inter}. Pour $\mathcal{P}_1$, $R_{\|V\_\mathcal{P}\|}$ est égal à $\{r_2, r_4, r_7\}$ et R_{Inter} correspond à $\{r_6, r_{10}, r_{11}, r_{12}, r_{14}, r_{15}\}$.

T_G est égal à $\{t_4, t_5, t_{6\_1}, t_{6\_5}, t_7, t_8, t_9, t_{10\_1}, t_{10\_5}, t_{11\_1}\}$.

Si toutes les transitions du modèle d'usage sont sélectionnées dans T_G, cela signifie que le modèle d'usage de la ligne de produits décrit uniquement l'usage d'un système, et il n'est en aucun cas un modèle d'usage d'une ligne de produits. Dans ce cas, toutes les transitions seront conservées.

Détection des cas incohérents $T_{incoherent}$:

Il consiste à identifier toutes les transitions incohérentes, c'est-à-dire identifier les transitions annotées à la fois par un ensemble d'exigences à conserver $R_{V\_\mathcal{P}}$ et un ensemble d'exigences à enlever $R_{\|V\_\bar{\mathcal{P}}\|}$. Ces cas doivent être détectés lors de la sélection et de la classification des transitions. Les cas incohérents sont toutes les incohérences survenues pendant la construction du modèle d'usage de la ligne de produits.

Le modèle d'usage contient aussi un autre genre de transitions, dit *transition de construction*, qui ne sont pas étiquetées par des exigences, mais nécessaires pour compléter le modèle d'usage. Certaines de ces transitions peuvent également contenir des données d'entrée et les résultats attendus. Dans certains cas, ces transitions peuvent être retirées si nécessaire afin de maintenir la validité de la forme de la variante du modèle d'usage dérivée, sinon elles seront conservées.

Les ensembles $R_{\|V\_\mathcal{P}\|}$ et $R_{\|V\_\bar{\mathcal{P}}\|}$ identifiés par algorithme 1, servent comme entrée pour l'algorithme 2, afin d'extraire et de classifier les transitions du modèle d'usage $\mathcal{M}_\mathcal{T}$, qui correspondent à l'ensemble des exigences du produit $\mathcal{P}$. Nous utilisons $R_{V\_\mathcal{P}}$, car il représente l'ensemble des exigences à conserver.

Le tableau 6.3 récapitule les règles de classification de chaque transition vis-à-vis de l'ensemble des exigences auquel elle est associée.

$R_{\|V\_\mathcal{P}\|}$	R_{Inter}	$R_{\|V\_\bar{\mathcal{P}}\|}$	
x			$t \in T_G$
x	x		$t \in T_G$
x		x	$t \in T_{incoherent}$
	x		$t \in T_G$
x	x	x	$t \in T_{incoherent}$
	x	x	$t \in T_{incoherent}$
		x	$t \in T_S$

TABLE *6.3 – Tableau récapitulatif des règles de classification des transitions*

Entrées: $R_{V\_\mathcal{P}}$, $R_{\|V\_\bar{\mathcal{P}}\|}$
Sorties: T_S, T_G, $T_{incoherent}$
début
  / Sélection des transitions à conserver */*
  pour chaque r ***de*** $R_{V\_\mathcal{P}}$ **faire**
    $T' \longleftarrow \gamma(r)$;
    si $(T' \nsubseteq T_G)$ **alors**
      $T_G \longleftarrow T_G \cup T'$;
    fin
  fin
  / Sélection des transitions à supprimer */*
  pour chaque r ***de*** $R_{\|V\_\bar{\mathcal{P}}\|}$ **faire**
    $T' \longleftarrow \gamma(r)$;
    si $(T' \nsubseteq T_S)$ **alors**
      $T_S \longleftarrow T_S \cup T'$;
    fin
  fin
  / Détection des cas incohérents */*
  pour chaque t ***de*** T_G **faire**
    si $(t \in T_S) \wedge (t \notin T_{incoherent})$ **alors**
      $T_{incoherent} \rightarrow ajouter(t)$;
    fin
  fin
fin

Algorithme 2: *Classification des transitions*

Analyse de la complexité de l'algorithme 2

L'algorithme 2, dans le cas de $\mathcal{P}$ consiste à trouver l'ensemble des transitions à conserver, à supprimer et en même temps à identifier les cas incohérents. Nous avons choisi une recherche linéaire : utiliser la fonction $\gamma(r)$ pour récupérer l'ensemble T' annoté par l'exigence r. Pour cette méthode, le pire cas est un modèle d'usage avec un grand nombre de transitions, car le nombre des transitions est important, ainsi que le nombre des exigences qui peuvent être associées à ce modèle. L'ordre de grandeur est de $O(mt)$, où m est le nombre des exigences du SUT et t le nombre de transitions.

6.2.5 Step D - Extraction de $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$

Après l'élagage des transitions, l'algorithme 3 procède à la dérivation d'une variante du modèle d'usage. Cette phase consiste à supprimer toutes les transitions et les états qui ne correspondent pas à la variante $\mathcal{P}$ sous test. Dans la pratique, le modèle généré peut être dans l'un des états suivants :

1. Un modèle états - transitions avec transitions manquantes,
2. Un modèle états - transitions complet avec des probabilités de profil incorrectes.

Nous présentons ces cas dans les sections suivantes. Le résultat attendu de l'algorithme de dérivation est un modèle d'usage d'états - transitions, assimilé à des chaînes de Markov étendues. Le profil d'usage associé à ce modèle doit avoir une distribution de probabilités correcte pour chaque transition sortant d'un état (voir step E).

Élagage des transitions dans le modèle d'usage de la ligne de produits :

Tout d'abord, nous initialisons le modèle d'usgae à dériver $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ à $\mathcal{M}_\mathcal{T}$, le modèle d'usage de la ligne de produits. Ensuite, toutes les transitions de l'ensemble T_S de $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ sont supprimées. À cette étape, une certaine incohérence peut apparaître dans le modèle d'usage, comme les branches cassées. À ce moment-là, nous pouvons rencontrer des cas incohérents tels que la suppression des transitions qui doivent être conservées pendant la correction du modèle d'usage. Cependant, pour avoir un modèle valide, l'algorithme 3 détecte les branches cassées à enlever. La figure 6.4 illustre le modèle dérivé $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ avec des branches cassées.

Modèle états - transitions avec transitions manquantes

Après la suppression de l'ensemble T_S, certaines incohérences apparaissent au niveau du modèle d'usage, comme des branches cassées. Pour avoir un modèle valide, l'algorithme 3 procède à la détection des branches cassées pour les supprimer. La figure 6.3, illustre le $\mathcal{M}_{\mathcal{T}_\mathcal{P}1}$ extrait avec des branches cassées.

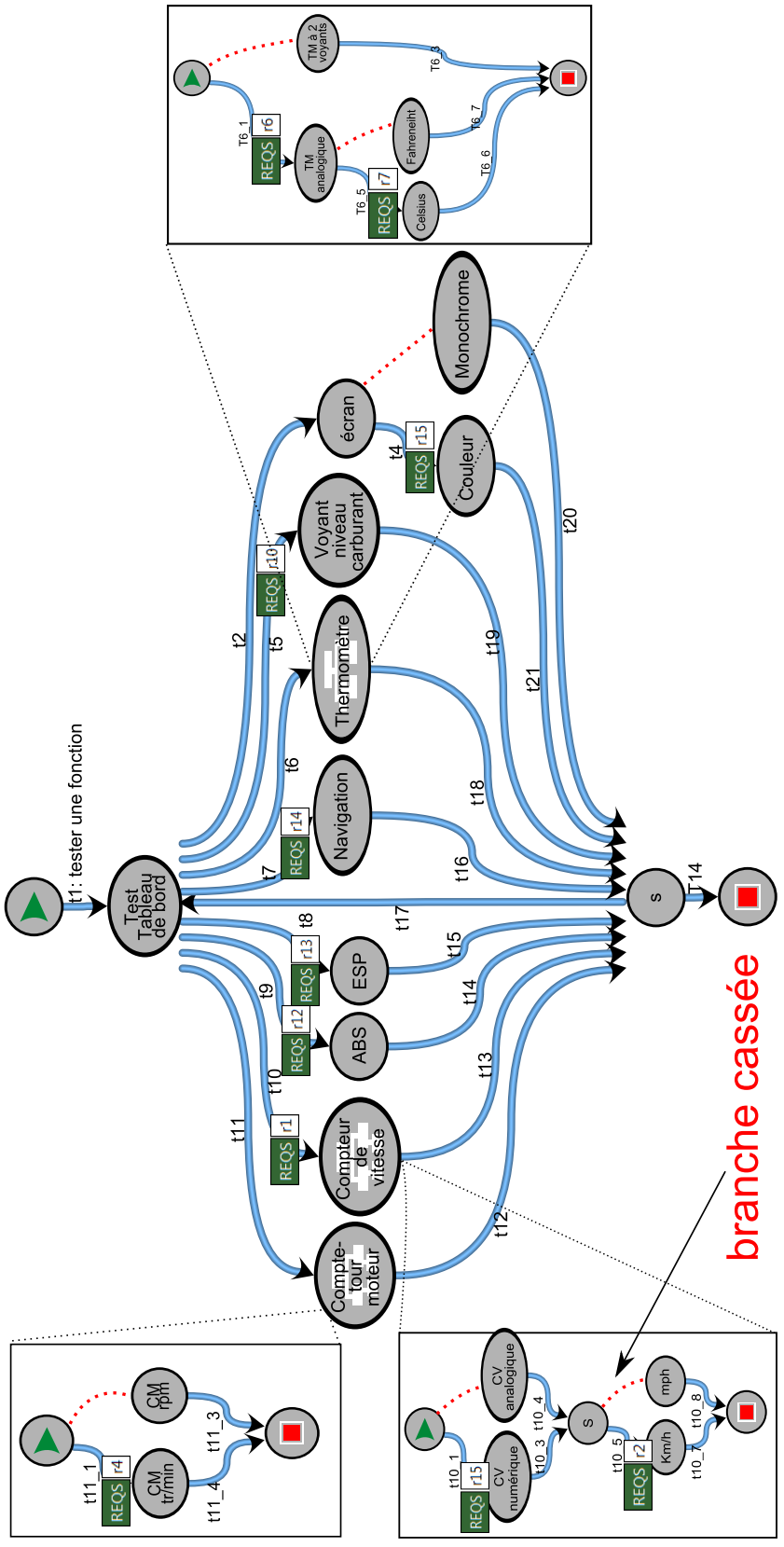


FIGURE *6.3 – Modèle d'usage du produit $\mathcal{P}_1$ avec des branches cassées*

Les transitions $\{t_{20}, t_{11\_3}, t_{10\_4}, t_{10\_8}, t_{6\_7}, t_{6\_3}\}$ doivent être supprimées pour obtenir un modèle équivalent à la variante choisie. Cet ensemble de transitions, appartenant aux branches cassées, peut contenir des transitions correspondant à la variante sous test. De ce fait, l'algorithme les identifie en tant que cas incohérents. Ces cas peuvent être le résultat d'une mauvaise construction du modèle d'usage de la ligne de produits.

En somme, pour ces cas particuliers, nous avons choisi de les supprimer et de les signaler au testeur pendant la génération ; un journal d'exécution est prévu pour aider le testeur à identifier l'ensemble des transitions supprimées du modèle d'usage de la ligne de produits.

Lors de la suppression des transitions T_S, deux problèmes peuvent survenir. Le premier concerne certains états inaccessibles à partir de l'état initial s_0 et le second se présente lorsqu'il n'est pas possible d'atteindre l'état final s_n. La résolution de ces problèmes est assurée par l'identification des branches cassées, qui consiste à détecter tous les chemins incomplets.

Le processus d'identification s'intéresse à la probabilité de visiter les états de la chaîne à partir de l'état initial s_0 qui doit être strictement positif, ainsi qu'à la probabilité de visiter l'état final s_n à partir de chaque état de la chaîne. Cette probabilité de visite est calculée sur une chaîne avec un état absorbant. Le processus ajoute un nouvel état d'absorption dans la chaîne avec une probabilité égal à 1, pour aller de l'état final s_n à l'état absorbant s_{n+1}. Cette construction est nécessaire, car nous voulons considérer la probabilité d'atteindre l'état final (par exemple, dans le cas où une transition entrante de l'état final a été supprimée par l'étape précédente de l'algorithme). Pour calculer la probabilité de visiter les états avant l'absorption, il faut supprimer la ligne et la colonne liée à cet état, afin d'obtenir la matrice stochastique $\mathbf{Q}$, qui a la forme suivante :

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q'} & \mathbf{q}_n \\ \mathbf{0} & 0 \end{pmatrix} \tag{6.9}$$

Où la sous-matrice $\mathbf{Q'}$ est les lignes et les colonnes liées à des états sauf l'état final s_n et $\mathbf{q}_n$ est le vecteur des probabilités pour aller des états $s_0, ..., s_{n-1}$ à l'état final s_n.

- $q_{ij} = p_{ij}$, si la transition $t_{ij} \in T_G$
- 0 si $t_{ij} \in T_S$
- 0 si $i = n$, si l'état final n'est pas accessible par un autre état
- 0 si $j = 0$, s'il n'est pas possible d'atteindre un état à partir de l'état initial.

Soit la matrice $\mathbf{B} = (\mathbf{I} - \mathbf{Q})^{-1}$ [Cin75]. $\mathbf{B}$ permet de calculer la probabilité de visiter s_j lorsque le processus est en s_i.

$$f_{ij} = b_{ij}/b_{jj} \tag{6.10}$$

L'état s_i et toutes ses transitions entrantes et sortantes sont retirés du modèle $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ dans deux situations particulières :

- si $f_{0i} = 0$, i.e. s_i est inaccessible,
- si $f_{in} = 0$, i.e. il n'est pas possible d'atteindre s_n lorsque le processus est à s_i.

6.2.6 Step E - Ajustement des probabilités

L'étape qui suit la purification du modèle d'usage des branches cassées est la mise à jour du profil d'usage associé. La figure 6.4 présente le modèle obtenu après la suppression des branches inaccessibles.

Modèle états - transitions complet avec des probabilités incorrectes

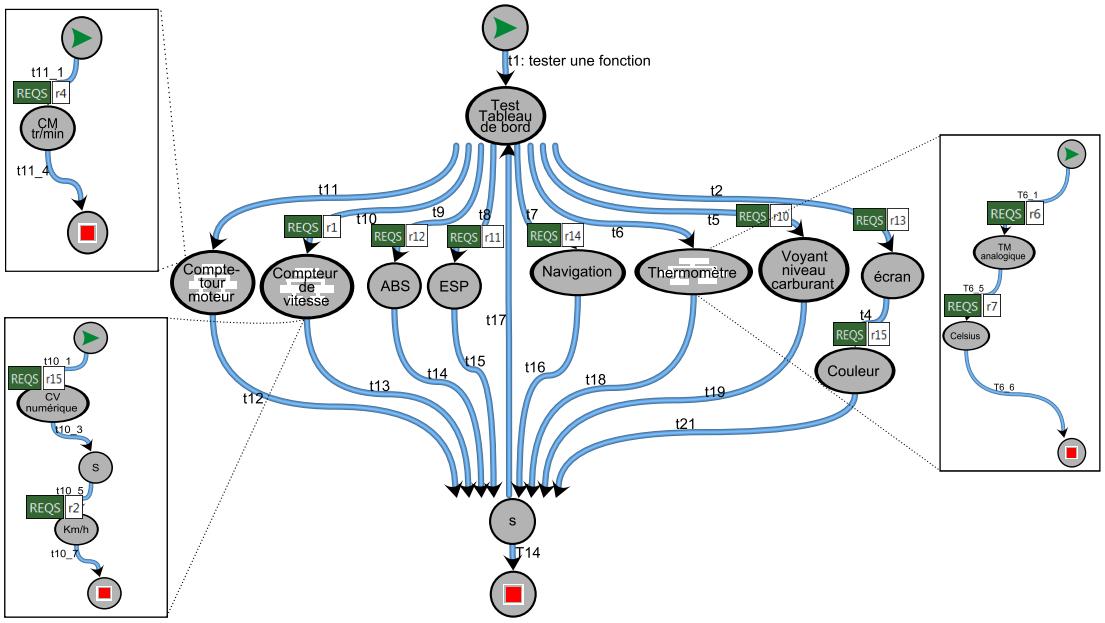


FIGURE *6.4 – Modèle d'usage de $\mathcal{P}_1$ correct avec ses sous-chaînes*

Le modèle d'usage extrait avec l'ensemble de ses sous-chaînes est complet au niveau de sa structure. Cependant, il nécessite une autre opération de mise à jour pour les probabilités associées au profil d'usage. Nous avons choisi une mise à jour proportionnelle,

qui consiste à distribuer les probabilités des transitions supprimées sur les transitions adjacentes. Cela revient à appliquer l'équation suivante :

$$\sum_{j \in S} p_{ij}(\mathfrak{F}) = \sum_{j \in S} p_{ij} / \sum_{k \in S} p_{ik} = 1 \tag{6.11}$$

L'algorithme 3 illustre les différentes étapes présentées ci-dessous :

Entrées: $\mathcal{M}_{\mathcal{T}}$, T_S, T_G, $T_{incoherent}$
Sorties: $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$
début
  $\mathcal{M}_{\mathcal{T}_\mathcal{P}} \longleftarrow \mathcal{M}_{\mathcal{T}}$;
  /* *Suppression des* T_S *du* $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ */
  supprimer_transitions($T_S, \mathcal{M}_{\mathcal{T}_\mathcal{P}}$) ;
  /* *Détection des états non atteignables dans* $\mathcal{M}_{\mathcal{T}}$ */
  $n \longleftarrow nb\_etat$;
  $\mathbf{Q}[n,n] \longleftarrow$ transformer_Modele2Matrice($\mathcal{M}_{\mathcal{T}_\mathcal{P}}$) ;
  $\mathbf{B} \longleftarrow (\mathbf{I} - \mathbf{Q})^{-1}$;
  pour $i \leftarrow 0$ **to** n **faire**
    pour $j \leftarrow 0$ **to** n **faire**
      $f_{ij} \longleftarrow b_{ij}/b_{jj}$;
      si $f_{ij} = 0$ **alors**
        $s \longleftarrow$ trouver_etat(i,j) ;
        $S \rightarrow ajouter(s)$;
      fin
    fin
  fin
  pour chaque $s \in S$ **faire**
    /* *Récupérer toutes les* t *sortantes et entrantes de l'état* s */
    $T \longleftarrow s \rightarrow transitions[*]$;
    pour chaque t ***de*** T **faire**
      si $t \in T_G$ **alors**
        afficher_problèmes(t);
      fin
    fin
    supprimer_transitions($T, \mathcal{M}_{\mathcal{T}_\mathcal{P}}$) ;
  fin
  /* *Suppression de tous les* s *inaccessibles* */
  Supprimer_etats($S, \mathcal{M}_{\mathcal{T}_\mathcal{P}}$) ;
  /* *Correction des probabilités de chaque* t *modifiée dans* $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ *pour avoir une chaîne de Markov valide* */
  ajuster_probabilités($\mathcal{M}_{\mathcal{T}_\mathcal{P}}$) ;
fin

Algorithme 3: *Extraction du modèle d'usage produit*

6.2.7 Analyse de la complexité de l'algorithme 3 et fiabilité

Complexité :

Pour extraire une variante du modèle d'usage $\mathcal{P}$, l'étape D est basée sur les données fournies par l'étape C qui implique l'étape A et l'étape B. L'étape préliminaire de l'algorithme 3 consiste à rechercher tous les états non atteignables de l'état s_0 à l'état s_n. Nous avons choisi une approche matricielle, pour énumérer de manière efficace tous les états non atteignables. Pour cette approche, le pire cas est un modèle d'usage avec un grand nombre de sous-chaînes et un nombre important d'états $(n \times n) \times m$. L'ordre de grandeur est donc $O(n^2m)$, où n est le nombre des états de $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ et m est le nombre d'instances des sous-chaînes. La deuxième partie de l'algorithme épure le $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ de toutes les transitions des états non atteignables. Enfin, l'algorithme effectue une vérification de cohérence du modèle d'usage. Ici, l'ordre de grandeur est de $O(n^2m)$.

Validation de l'algorithme 3 :

La génération des cas de test est considérée comme un parcours aléatoire sur une chaîne de Markov jusqu'à que l'état final S_n soit atteint. Nous sommes intéressés par la probabilité de visite : cette probabilité doit être strictement positive de s_0 jusqu'à tous les autres états du modèle, l'état final s_n doit être accessible à partir de chaque état de la chaîne. Cette probabilité de visite est calculée sur une chaîne avec l'état absorbant, c'est-à-dire la probabilité de visiter un état avant l'absorption.

La matrice inverse $\mathbf{B} = (\mathbf{I} - \mathbf{Q})^{-1}$ existe toujours, car $\mathbf{Q}$ (voir définition 6.9) est une sous-matrice stochastique, c'est-à-dire que tous les éléments sont compris entre 0 et 1 et la somme de chaque ligne est inférieure ou égal à 1. $\mathbf{B}$ s'exprime sous la forme suivante :

$$\mathbf{B} = \begin{pmatrix} \mathbf{B} & \mathbf{b} \\ \mathbf{0} & 1 \end{pmatrix} \tag{6.12}$$

Où $\mathbf{b}$ est composé de valeurs égales à 0 ou 1.

En effet, comme nous n'avons considéré qu'un seul état absorbant, soit le processus est absorbé par cet état, soit il n'est pas possible de l'atteindre. Par conséquent, les états à éliminer sont identifiés. Si nous supprimons toutes les lignes et colonnes, où $b_{0i} = 0$ et $b_{in} = 0$, les probabilités de visite peuvent être calculées sans difficulté et le modèle est dit «nettoyé», c'est-à-dire sans branches cassées. Finalement, nous ajoutons la propriété que la somme des probabilités associées aux transitions sortantes de chaque états est égal à 1.

$$\sum_{j \in S} p_{ij}(\mathfrak{F}) = \sum_{j \in S} p_{ij} / \sum_{k \in S} p_{ik} = 1 \tag{6.13}$$

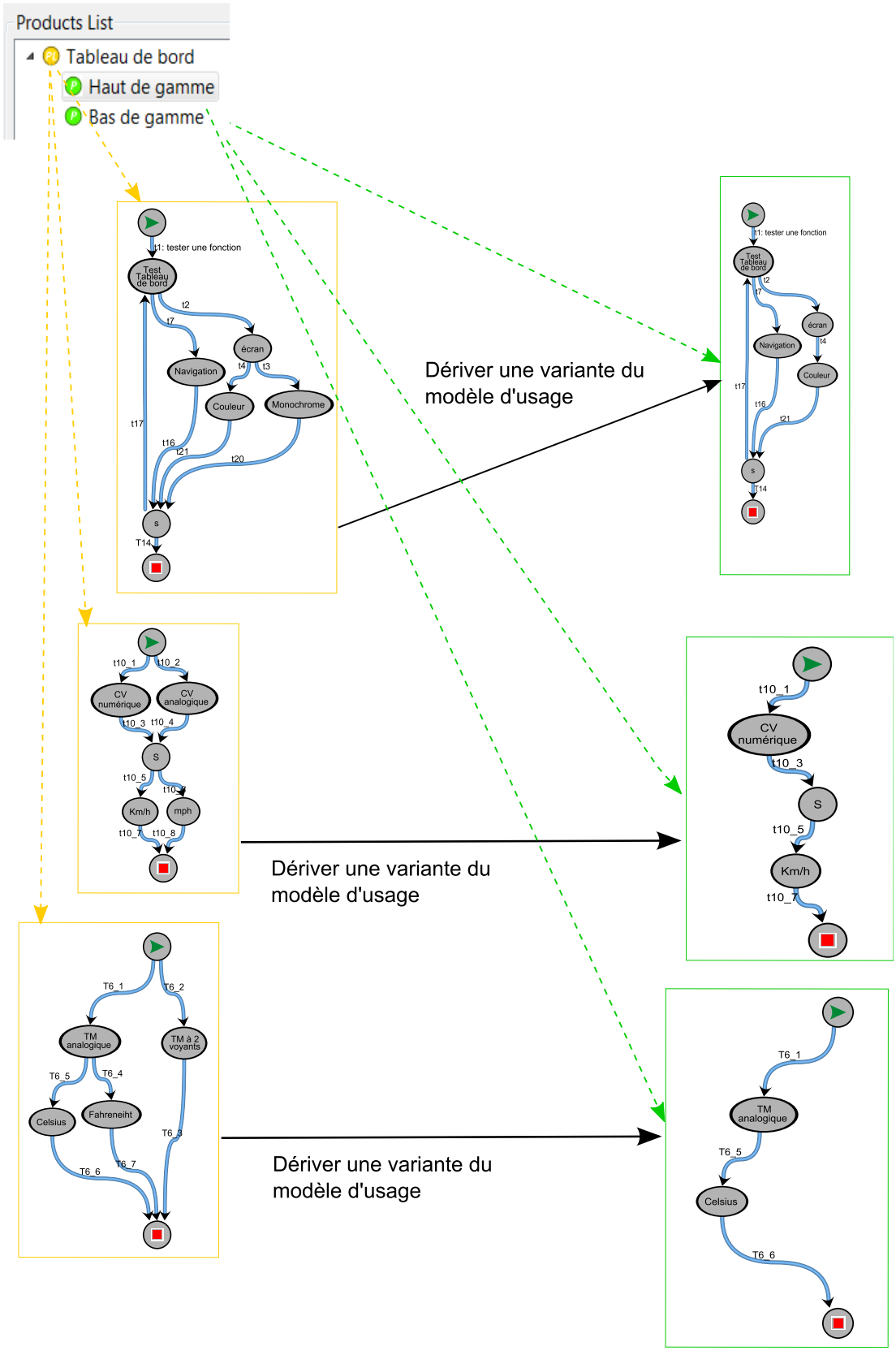


FIGURE *6.5 – Dérivation des variantes du modèle d'usage*

En somme, nous pouvons conclure que le modèle d'usage est une chaîne de Markov étendue, où des cas de test peuvent être générés par un parcours aléatoire.

6.3 Discussion

Notre objectif est de montrer qu'en mélangeant les techniques du *model-based testing* avec les principes de l'ingénierie des lignes de produits, nous pouvons réduire les coûts de test et l'effort de modélisation pour tester plusieurs variantes du système dans une optique de ligne de produits. Nous observons que notre approche n'a aucune incidence sur les solutions MBT existantes, comme MaTeLo, puisque, nous étendons l'approche effective pour les systèmes uniques à une solution de ligne de produits appropriée. La variabilité est gérée par un modèle de variabilité externe. Les changements au niveau de la ligne de produits peuvent également ne pas avoir d'impact sur le modèle d'usage de la ligne de produits, et par conséquent, ils ne peuvent pas impacter les modèles d'usage existants.

Dans cette section, nous présentons et discutons les réflexions sur les choix effectués dans notre approche globale.

Réalisation du modèle d'usage de la ligne de produits

Plusieurs travaux se sont intéressés au processus du *model-based testing* pour le test des lignes de produits, en particulier pour réaliser un seul modèle de test apte à représenter le comportement attendu de la ligne de produits, comme précité dans la section 4.4. Dans la même démarche, Oster et al. [SOS11] utilisent un unique modèle de test pour la ligne de produits et présentent aussi les lacunes quand nous utilisons une telle approche ; nous en citons quelques-unes et nous les complétons :

- La création et le maintien d'un «grand» modèle de test d'une ligne de produits entière est difficile et nécessite une bonne maitrise des techniques de la modélisation.
- Vérifier la cohérence de ce genre de modèle est une tâche compliquée.
- La génération des cas de test n'est pas possible avec un tel modèle, car les tests ne vont pas correspondre à un système, puisque le modèle lui-même ne représente pas un système, mais une architecture commune de la ligne de produits avec ses parties variables.
- La création des profils d'usage personnalisés est difficile, car après la dérivation des variantes du modèle d'usage ces profils seront recalculés automatiquement par l'approche.

Cependant, modéliser en un seul modèle d'usage, toutes les *features* de la ligne de produits à tester offrent aussi des avantages tels que ceux-ci :

- Nous réutilisons les artefacts de test pour chaque nouveau produit de la ligne de produits, plutôt que de partir de zéro.

- Moins d'artefacts de test maintenus pour la famille de produits, puisque la réutilisation est gérée au niveau d'un seul modèle d'usage de la ligne de produits.
- Les variantes du modèle d'usage dérivées à partir du modèle d'usage de la ligne de produits, qui sont examinées et utilisées dans de nombreux produits, dans différents contextes et dans différents scénarios, ce qui conduit à une meilleure qualité.

Mise en place et évaluation de l'approche

Nous observons que l'approche fonctionne mieux si les exigences sont bien détaillées et préparées pour faciliter l'identification des *features* auxquelles doivent être associées, en d'autres termes, les exigences et les modèles d'usage doivent être structurés d'une manière qui facilite l'association des *features* avec les exigences. Nous avons identifié deux autres raisons, qui sont :

Premièrement
: La modélisation de la variabilité est basée sur les exigences fonctionnelles de la ligne de produits, sur laquelle l'ingénieur se base pour identifier ce qui peut varier ou non dans le système. Ainsi, l'usager doit rendre ce lien explicite à l'aide de l'outil. En outre, nous proposons d'utiliser OVM comme approche, car nous souhaitons focaliser uniquement sur la variabilité, afin de pouvoir piloter le modèle de test de la ligne de produits.

Deuxièmement
: Le processus MBT requiert un étiquetage explicite du modèle de test avec des exigences. Cela permet de fournir des statistiques concernant la couverture des exigences par les cas de test générés.

Des projets peuvent comporter plus de 4 000 exigences fonctionnelles. Nous admettons que dans ce cas-là, il est difficile de faire le lien ; mais, comme l'industriel a pu réaliser le modèle d'usage correspondant à la ligne de produits et établir avec succès le lien entre les exigences avec les éléments du modèle équivalent, nous estimons qu'il est aussi possible d'appliquer notre méthodologie.

Nous avons également noté les réactions de nos partenaires qui rapportent les résultats de l'évaluation de notre approche. Les praticiens font état d'une réduction du coût pour le développement des cas de test grâce au *model-based testing* qui permet de générer automatiquement les cas de test. En outre, ils mettent en évidence le caractère invasif minimal de la solution, car les exigences établies et les modèles d'usages sont réutilisés. Nous avons également appris que la méthodologie de gestion de la variabilité influence grandement la façon dont les exigences sont spécifiées et qu'une phase de formation est nécessaire avant de commencer d'utiliser l'approche.

6.4 Résumé

Cette approche présente une combinaison du *model-based testing* et l'ingénierie des lignes de produits, pour but de générer automatiquement les cas de test pour une famille de produits. Nous nous basons sur le modèle OVM pour représenter la variabilité de la ligne de produits ainsi que sur le modèle d'usage pour la validation.

Nous étendons la définition formelle d'OVM pour inclure les exigences et décrire les relations avec les *features*. Nous utilisons cette relation pour relier le modèle de variabilité avec le modèle d'usage. Nous n'avons pas proposé une méthodologie d'analyse automatique de la variabilité de la ligne de produits, car elle n'est pas le sujet de cette thèse. Nous estimons que notre méthodologie peut être compatible avec d'autres approches existantes. Cette possibilité est discutée dans la section perspective.

Par contre, pour valider une ligne de produits avec l'approche *model-based testing*, cela nécessite des adaptations. Nous étendons le formalisme du modèle d'usage pour pouvoir y introduire la variabilité. En outre, nous proposons de modéliser avec un seul modèle d'usage toutes les *features* à tester dans une ligne de produits entière. Cependant, pour établir un lien entre le modèle de variabilité et le modèle d'usage de la ligne de produits, nous proposons une méthodologie de traçabilité entre les *features* et les transitions du modèle d'usage via les exigences fonctionnelles. Ce type de lien donne la possibilité de suivre les évolutions des exigences sans briser les liens établis entre les *features* et les transitions, car ce lien dépend d'une entité abstraite sur laquelle se basent les ingénieurs pour réaliser le modèle de variabilité du domaine et le modèle de test correspondant.

Nous utilisons cette démarche pour dériver automatiquement des variantes du modèle d'usage à partir du modèle de la ligne de produits. L'algorithme de dérivation est basé sur cinq étapes essentielles. La première phase consiste à identifier les *features* qui composent le produit $\mathcal{P}$. La deuxième étape permet de classifier les exigences en sous-ensembles d'exigences qui correspondent au produit à tester et en sous-ensembles d'exigences non associés à ce produit. La troisième étape est l'identification des transitions à conserver et à supprimer du modèle d'usage. La quatrième étape consiste à dériver une variante du modèle d'usage. La dernière étape consiste à corriger les probabilités du modèle généré, afin d'avoir une chaine de Markov valide. Un modèle d'usage d'une variante spécifique décrivant son comportement est désigné en tant qu'un modèle d'usage traditionnel. Celui-ci est utilisé par la suite, par le processus du MBT afin de générer les cas de test.

Après avoir présenté le concept et la théorie de notre méthodologie, nous présentons MPLM, un outil qui implémente cette approche. Pour soutenir son application dans le contexte industriel, le champ d'action dans la partie suivante est de fournir une chaîne d'outils permettant à l'industrie d'appliquer cette approche. En outre, la partie suivante présente les résultats de l'expérimentation de la contribution dans un contexte industriel réel. L'outil est utilisé par Airbus Defense & Space dans le contexte du projet européen MBAT.

Troisième partie

Implémentation et évaluation

Chapitre 7

Implémentation

Dans ce chapitre, nous présentons l'outil MaTeLo Product Line Manager [1] (MPLM), qui implémente les fondements théoriques de notre approche. MPLM est une extension basée sur Eclipse de la suite d'outils MaTeLo, qui prend en charge la dérivation automatique des variantes du modèle d'usage à partir d'un modèle d'usage de la ligne de produits. Nous avons développé l'outil pour permettre son application dans un contexte industriel.

7.1 MaTeLo Product Line Manager

MPLM est une perspective intégrée dans la nouvelle plateforme de l'outil MaTeLo basée sur Eclipse RCP (figure 7.1), avec près de 14 000 lignes de code dans sa version actuelle. MPLM vise à adapter la chaîne d'outils MaTeLo pour supporter le test des lignes de produits. MaTeLo est un outil industriel basé sur le processus du *model-based testing*, qui permet de générer automatiquement des cas de test pour des systèmes uniques en utilisant un modèle d'usage. Avec l'aide de MPLM, l'utilisation des variantes de modèles peuvent être dérivées pour MaTeLo afin de générer des cas de test spécifiques pour les variantes d'une ligne de produits. L'approche de MPLM peut être résumée comme suit :

- Développer un modèle de variabilité avec l'approche *Orthogonal Variability Model* (OVM). OVM fournit une documentation explicite de la variabilité liée à des *features*.

- Développer un modèle d'usage pour une ligne de produits avec MaTeLo. Le modèle d'usage comprend la traçabilité des exigences de la ligne de produits.

- Associer les exigences de la ligne de produits avec les *features* du modèle OVM.

- Configurer différentes variantes en sélectionnant l'ensemble de combinaisons de *features* valides pour le test.

1. `http://people.irisa.fr/Hamza.Samih/mplm`

– Dériver des variantes du modèle d'usage couvrant les *features* pertinentes et les exigences de produits.

– Générer des cas de test avec MaTeLo pour chaque variante de la ligne de produits, en se basant sur les variantes de modèles d'usage dérivées.

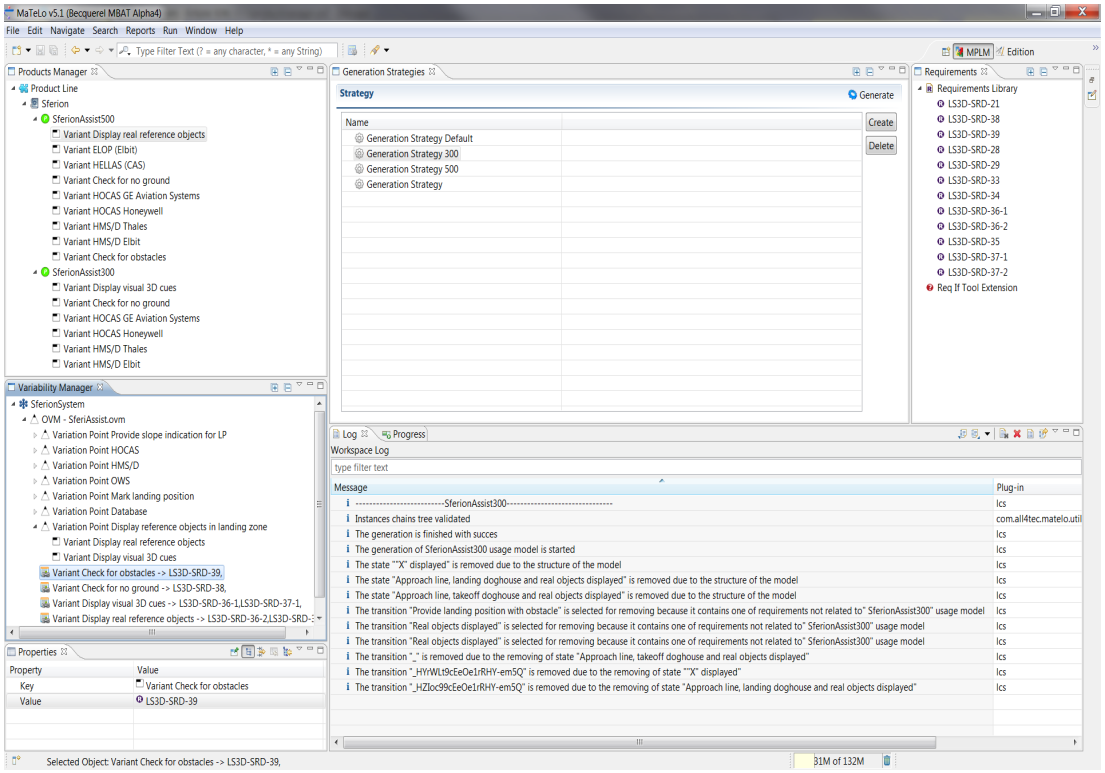


FIGURE *7.1 – L'outil MPLM*

Une étude de cas expérimentale a été réalisée en collaboration avec le partenaire industriel Airbus Defence & Space dans le cadre du projet ARTEMIS Joint Undertaking research project MBAT afin de valider l'approche d'un point de vue industriel (voir le chapitre 8 pour la description du cas d'étude industriel *situational awareness suite Sferion™*). Les sections suivantes récapitulent les étapes effectuées dans l'étude de cas et illustrent l'utilisation de l'outil MPLM et la chaîne d'outils MaTeLo liée.

7.2 Ingénierie des lignes de produits

L'ingénierie des lignes de produits couvre toutes les activités qui permettent la réutilisation systématique. Cela inclut la compréhension du domaine, l'identification

du champ d'application, le développement des artefacts communs et la gestion de la variabilité.

Définition du cadre de la ligne de produits :

Le périmètre de la ligne de produits fournit le point départ de toutes les activités subséquentes. Au cours de l'évaluation, nous identifions les *features* des produits concernés et les configurations de produits potentiels sur la base de renseignements commerciaux et du marché pour définir le champ d'application de la ligne de produits. Cette étape comprend habituellement les activités suivantes :

- Identifier les *features* qui pourraient fournir de la valeur à au moins un client.
- Classifier les éléments selon les catégories obligatoires, optionnelles ou éventail d'alternatives.
- Évaluer la pertinence de chaque *feature* en termes de valeur (capacité de contribuer à la satisfaction des clients), les risques (maturité du développement) et le coût (l'effort requis pour le développement).
- Définir le champ d'application de la ligne de produits en y incluant seulement les *features* de grande pertinence.

Réalisation du modèle de variabilité :

Le résultat de la détermination du champ d'application de la ligne de produits est ensuite formalisé à l'aide de l'approche OVM. L'outil REMiDEMMI [2] est employé pour développer des modèles de variabilité avec le langage OVM. Les *features* variables identifiées lors de la détermination du champ d'application de la ligne de produits sont représentées comme des points de variation dans OVM. Par exemple, le modèle OVM sur la figure 7.2 représente le point de variation "*Mark landing position*" avec des dépendances à la *feature* obligatoire "*Check for no ground*" et la *feature* optionnelle "*Check for obstacles*". Dans le cas où la *feature* optionnelle est sélectionnée, la contrainte de type <*requires*> garantit qu'aussi un "*OWS*" est sélectionnée pendant la configuration de la variante.

2. `/http://remidemmi.cdhq.de/`

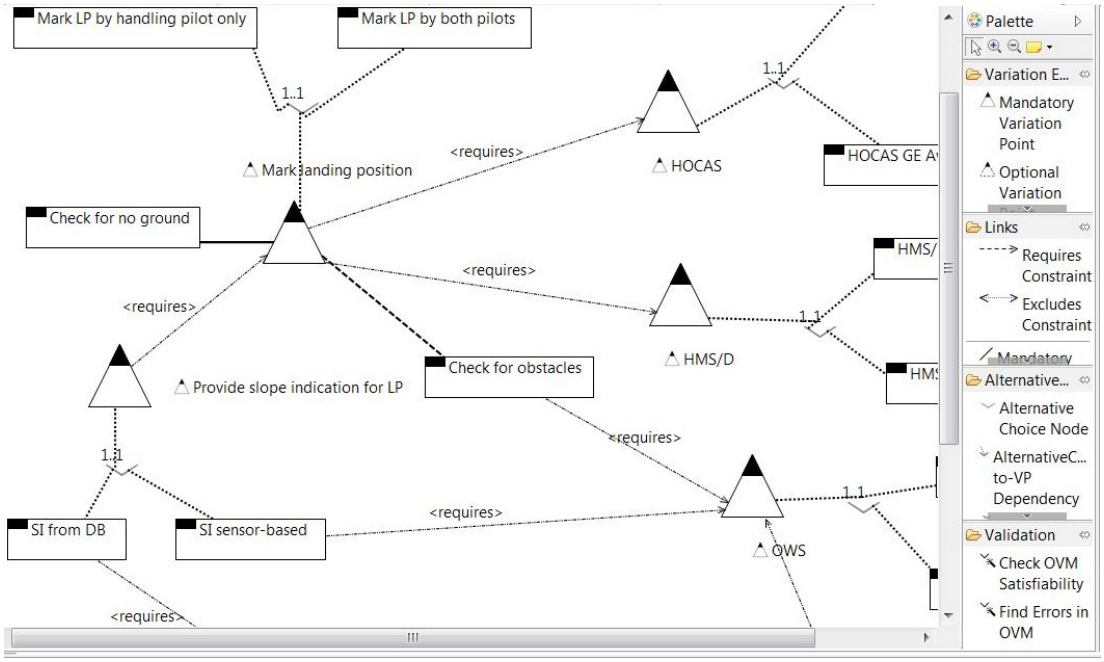


FIGURE *7.2 – Créer OVM dans l'environnement Eclipse*

Élaborer les exigences de la ligne de produits :

À la fois les *features* communes et variables de la fonction "*landing symbology*" sont affinées dans un ensemble d'exigences de système. Les exigences sont créées dans l'outil de gestion des exigences **IBM Rational DOORS**. Dans cette étude de cas que des exigences fonctionnelles sont considérées. Un exemple est donné ci-dessous :

> **LS3S-SRD-38 :** La fonction "*LS3D*" doit visualiser "*NO GROUND*" à la réception de la marque d'atterrissage en position de déclenchement, si le *landing symbology* a été activé et il n'y a aucune intersection entre *LoS* du tracker de la *HMS/D* et la surface du sol à la position marquée d'atterrissage .
>
> **LS3S-SRD-39 :** La fonction *LS3D* doit visualiser "X" à la réception de la marque d'atterrissage en position de déclenchement, chaque fois qu'un *sensor-classified obstacle* est détecté dans le carré du poste de commande centrée à la position marquée d'atterrissage.

Réalisation du modèle d'usage de la ligne de produits :

Dans cette activité, les exigences fonctionnelles du système de la ligne de produits sont importées depuis *DOORS* dans la perspective édition de MaTeLo, ensuite le modèle

d'usage de la ligne de produits est réalisé. La figure 7.6 illustre le modèle d'usage créé couvrant toutes les *features* et les exigences de la ligne de produits à tester. Les exigences du système importées sont affectées aux transitions respectives du modèle d'usage, comme cela est illustré sur la figure 7.3.

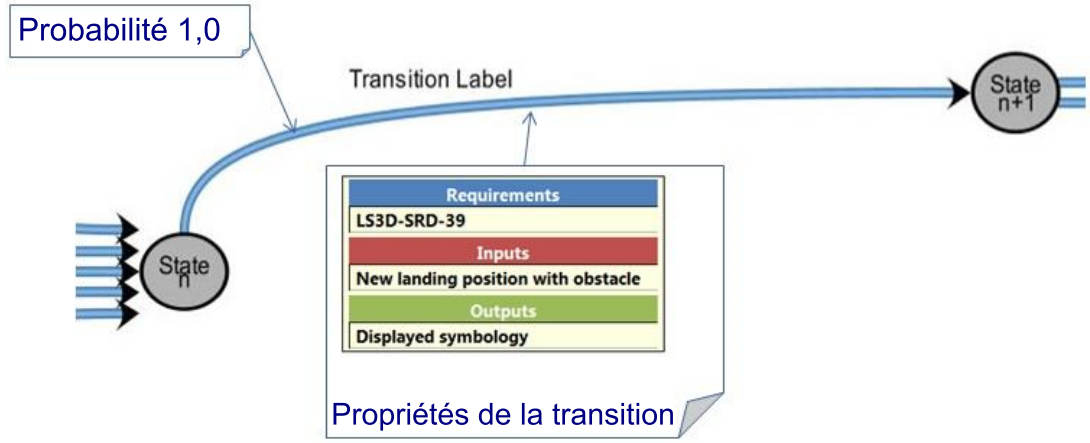


FIGURE *7.3 – Les données associées à la transition*

Association des *features* aux exigences :

L'outil MPLM prend en entrée un modèle d'usage de la ligne de produits et un modèle OVM. Comme illustré à la figure 7.5, MPLM répertorie toutes les exigences couvertes par le modèle d'usage ainsi que les *features* variables fournies par le modèle OVM. Dans cette activité, les associations entre les exigences et les *features* sont définies. Par exemple, la *feature* "*Check for obstacles*" est associée à l'exigence de "**LS3D-SRD-39**".

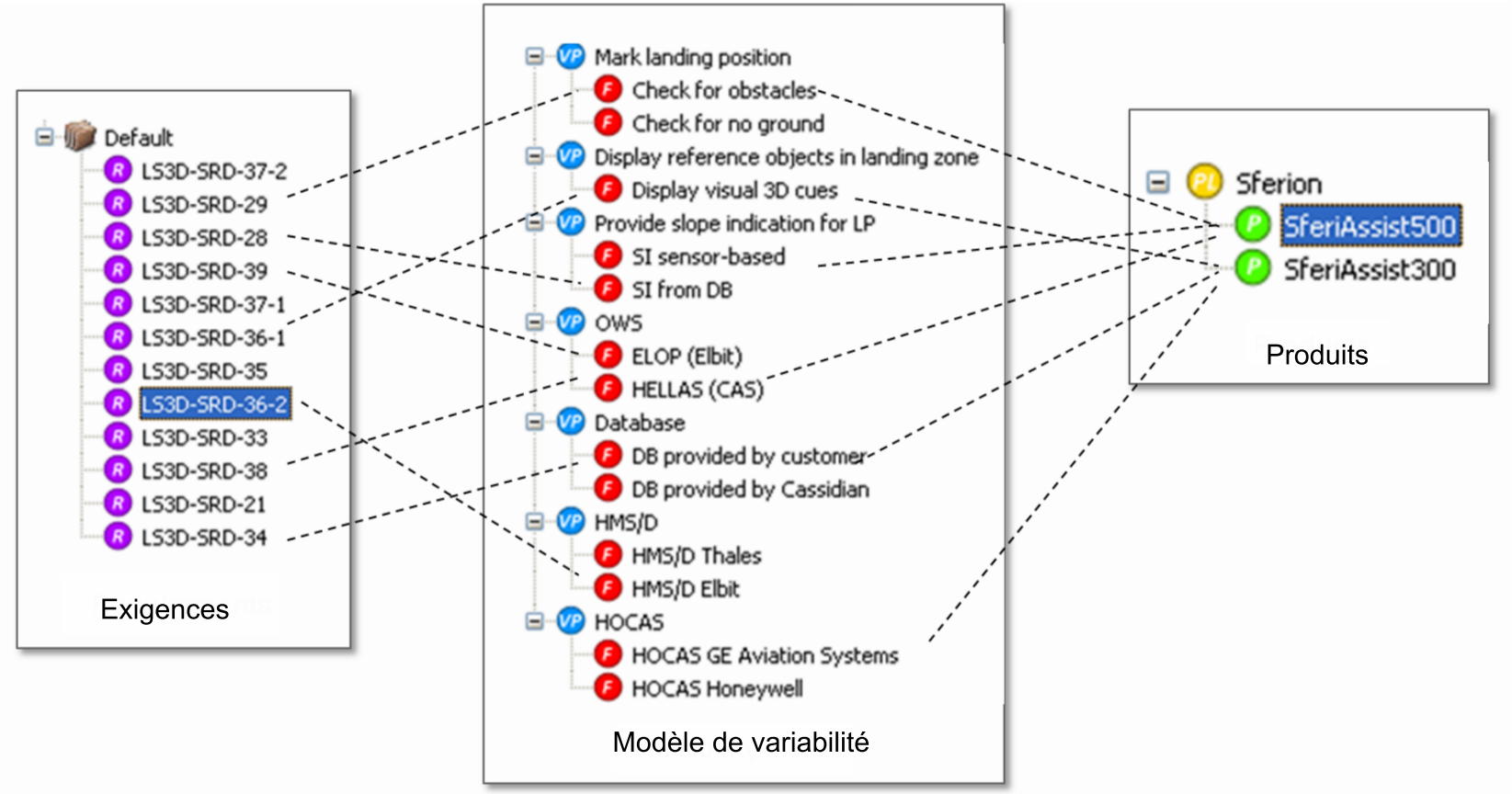


FIGURE *7.4 – Mapping des features avec les exigences de la ligne de produits*

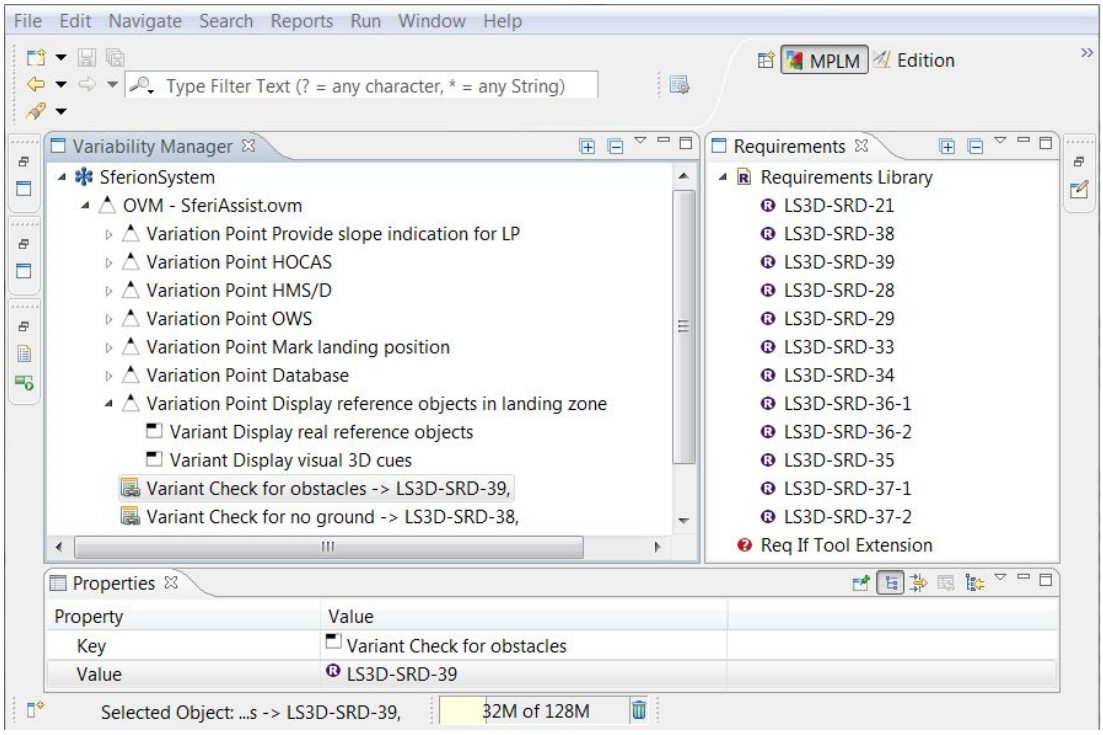


FIGURE *7.5 – Associer les features avec les exigences dans MPLM*

7.3 Ingénierie de produits

La décision de réaliser un produit est prise en se basant sur les exigences des clients, les opportunités commerciales ou des contrats clients. L'objectif principal de l'ingénierie de produits est la réutilisation autant d'atouts que possible pour dériver de nouvelles variantes. Dans cette étude de cas, nous sélectionnons les *features* souhaitées pour une variante et dérivons une variante modèle d'usage qui permet de générer cas de test pour une variante de système.

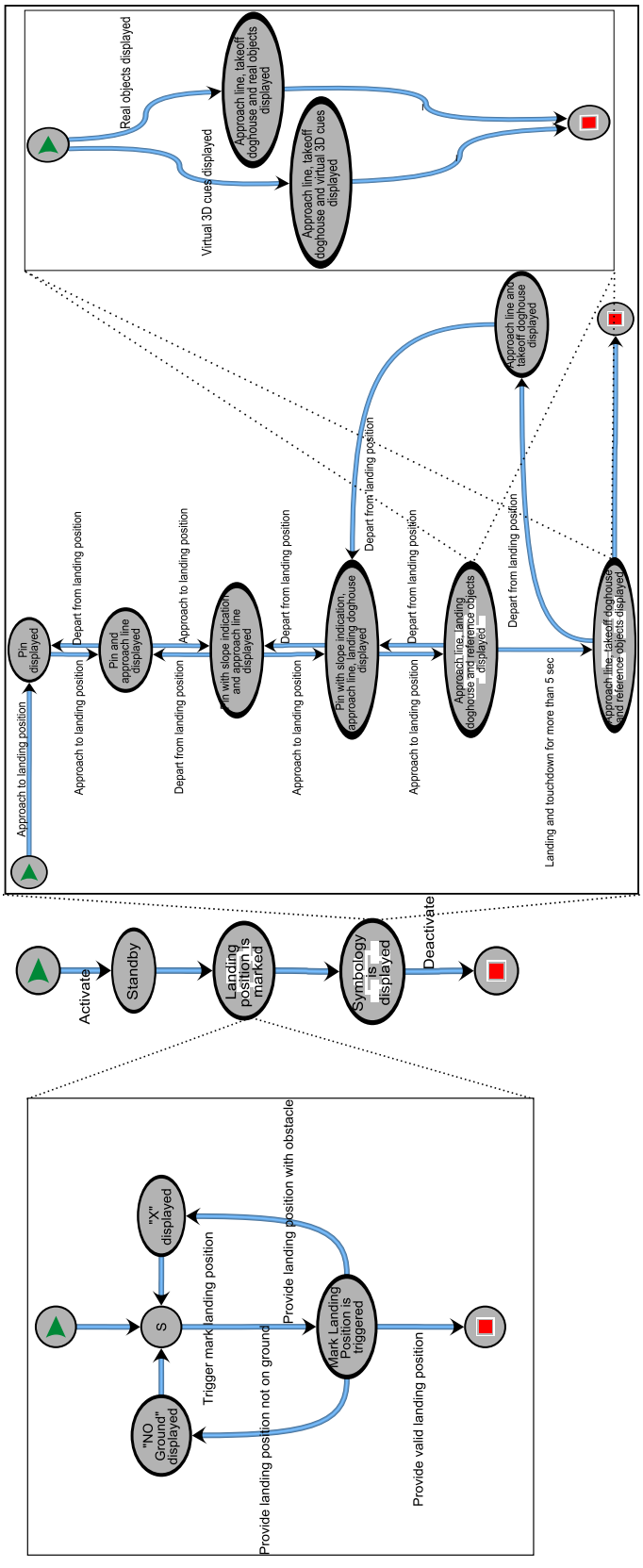


FIGURE 7.6 – *Le modèle d'usage de la ligne de produits Sferion™*

Configuration des variantes :

La configuration des variantes sous MPLM consiste à sélectionner les *features* variables qui devraient être fournis par la variante. Lors de la configuration, les relations entre les points de variation et les *features* (c'est-à-dire obligatoires, optionnelles ou choix alternatifs) ainsi que les contraintes entre les éléments du modèle OVM (par exemple, <*require*> ou <*exclude*>) sont respectées. Selon la figure 7.7, deux variantes ont été définies : *SferiAssist500* (produit haut de gamme) et *SferiAssist300* (produit bas de gamme). La fonction "Check for obstacles" est seulement présente dans le produit haut de gamme.

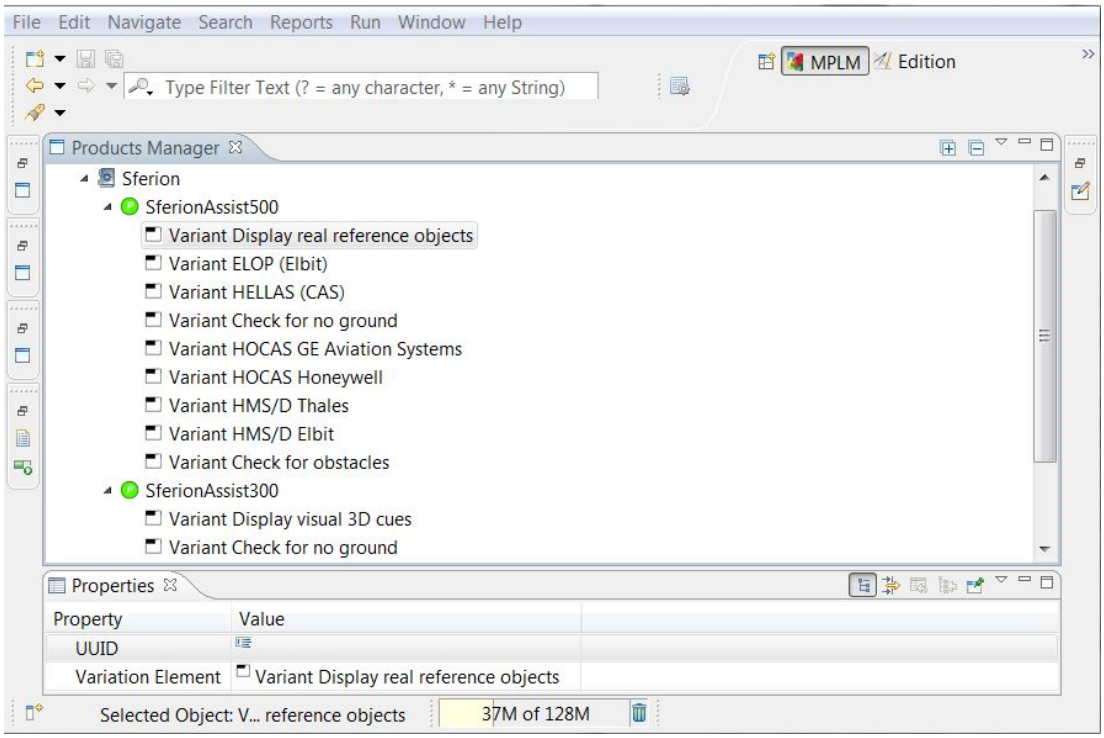


FIGURE *7.7 – Configuration des produits à tester dans MPLM*

Dérivation des variantes du modèles d'usage :

Dans cette activité, une variante du modèle d'usage couvrant toutes les *features* sélectionnées et les exigences associées de la variante sélectionnée, est dérivée à partir du modèle d'usage de la ligne de produits.

La dérivation automatique consiste à projeter un ensemble de *features* composant une variante sur le modèle d'usage de la ligne de produits. Les liaisons entre les *features*, les exigences de la ligne de produits et les transitions du modèle d'usage de la ligne de produits, permettent la réutilisation du modèle d'usage de la ligne de produits pour extraire une variante du modèle d'usage avec seulement les transitions et les exi-

gences de la variante sélectionnée pour le test. Ainsi, la variante du modèle d'usage dérivée représente cas de test nécessaires pour le produit haut de gamme. Le processus de dérivation s'assure que le modèle dérivé est valide, en calculant s'il y a des états inaccessibles dans le modèle, si oui toutes les branches cassées seront supprimés, et les probabilités des transitions retirées seront réparties proportionnellement sur les transitions adjacentes. Ensuite, tous les profils associés seront automatiquement mis à jour. Les détails de génération, tels que les états et les transitions qui ont été supprimés dans le modèle d'usage de la ligne de produits, sont affichés dans un journal d'exécution comme représenté sur la figure 7.8.

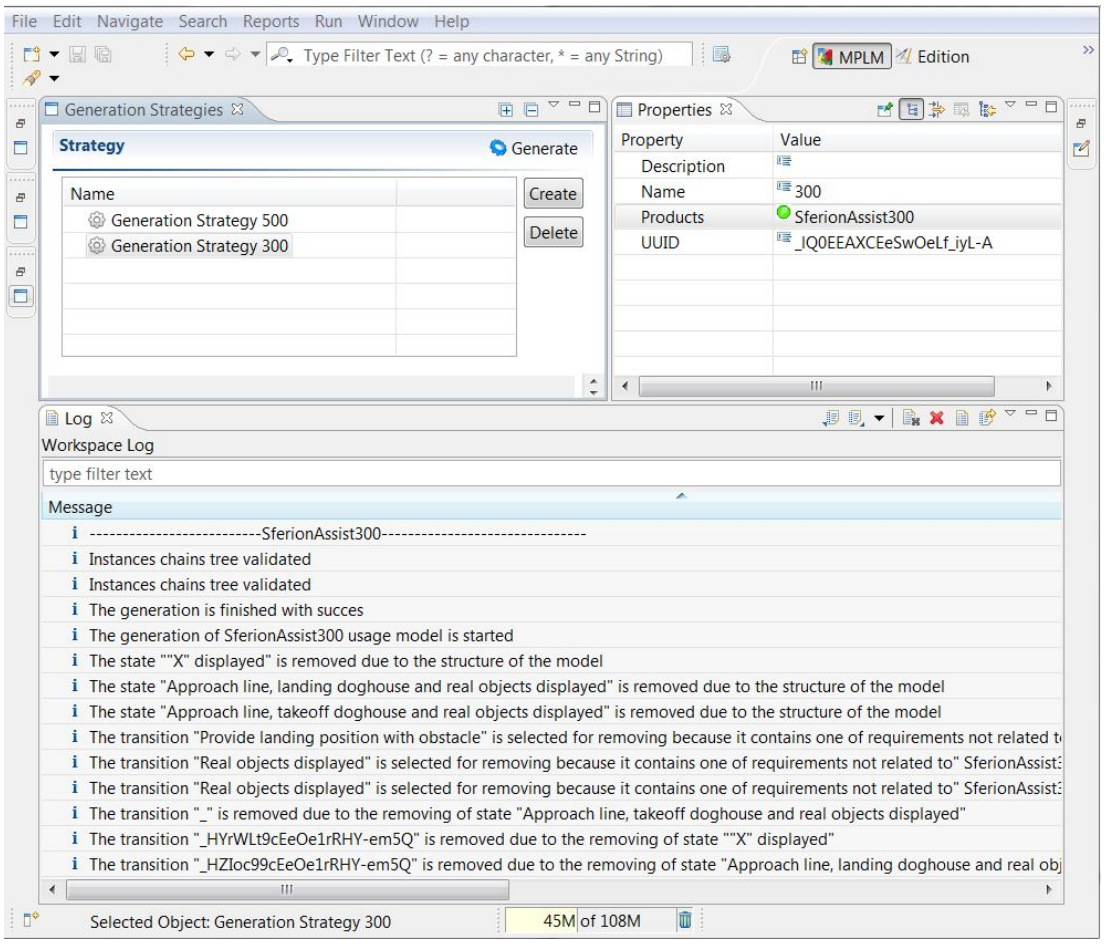


FIGURE *7.8 – Dériver des variantes du modèle d'usage dans MPLM*

Génération des cas de test :

Les cas de test spécifiques à une variante peuvent être automatiquement générés sur la base de la variante de modèle d'usage dérivée en utilisant la perspective de génération de l'outil MaTeLo. La génération des cas du test est basée sur la définition

d'une stratégie de test en choisissant un profil d'usage et l'algorithme de génération. Par la suite, la génération des cas de test consiste à sélectionner des transitions en fonction de leurs probabilités et l'algorithme choisi.

7.4 Résumé

MPLM est une nouvelle approche qui étend le *model-based testing* à l'ingénierie des lignes de produits. Le modèle d'usage de la ligne de produits couvre toutes les *features* de la ligne de produits sous test et par conséquent, il permet de réutiliser efficacement les artefacts de test (par exemple, modèle d'usage) entre les différentes variantes. Dans le passé, des projets distincts ont été installés pour différents clients, et par conséquent l'analyse des exigences et la conception des cas de test a été réalisée indépendamment pour toutes les variantes sans réutilisation systématique. Par conséquent, nous ne bénéficions plus des avantages du *model-based testing*. MPLM est proposé pour remplacer cette pratique.

Nous présentons dans la suite le résultat de l'expérience industrielle avec *"situational awareness suite Sferion™"* réalisée avec MPLM et MaTeLo. Cette expérimentation montre que les praticiens peuvent réduire le coût de développement de cas de test tout en augmentant le niveau d'abstraction. Nous faisons également état des répercussions de l'approche en termes d'adoption et de la méthodologie du test.

Chapitre 8

Évaluation

Nous évaluons dans ce chapitre notre approche implémentée par l'outil MPLM avec une étude de cas industriel, dans le domaine de l'aéronautique de notre partenaire *Airbus Defence & Space*, dans le but de valider les concepts proposés dans un contexte réel. Ainsi donc, vérifier que MPLM arrive bien à dériver des variantes du modèle d'usage utilisables et pouvant servir à générer des cas de test exécutables. L'expérience a été réalisée de bout en bout par le soin de notre partenaire.

8.1 Contexte industriel

L'étude de cas proposée par *Airbus Defence & Space* est un système de guidage qui prend en charge les pilotes d'hélicoptère dans un environnement visuel dégradé.

La motivation d'Airbus Defence & Space pour expérimenter l'approche tient à deux raisons essentielles :

- Introduire la gestion de la ligne de produits pour réduire les coûts de développement, lorsque des objets provenant d'une famille de produits peuvent être stratégiquement réutilisés pour développer des produits différents, ce qui permet l'amélioration de la qualité, puisque les artefacts de la famille de produits sont examinés et utilisés dans plusieurs produits.
- Automatiser les activités de test. Le modèle d'usage permet la génération et l'exécution des cas de test, en outre, la traçabilité entre les exigences et les cas de test peut être établie automatiquement. Ceci est important pour prouver la conformité du produit testé par rapport à ses besoins.

8.1.1 Situational awareness suite Sferion™

Airbus Defence & Space développe des systèmes avioniques qui prennent en charge les pilotes d'hélicoptère dans des environnements visuels dégradés en raison de la pluie, du brouillard, du sable, de poussière et de la neige. De nombreux accidents peuvent être attribués directement à ces environnements visuels dégradés où les pilotes, souvent, perdent l'orientation spatiale et le contact avec leur environnement. Dans cette étude de cas, nous utilisons la fonction symbolique d'atterrissage qui fait partie de la

Situational awareness suite Sferion™. La fonction symbolique d'atterrissage assiste les pilotes d'hélicoptère pendant la phase d'atterrissage. Elle permet au pilote de repérer la position d'atterrissage envisagée sur le sol à l'aide du suivi par tête HMS/D (Helmet Mounted Sight and Display) et HOCAS (Hands on Collective and Stick). Lors de la phase finale d'atterrissage, la fonction symbolique d'atterrissage améliore la perception de l'espace par le personnel navigant en affichant des repères visuels 3D conformes à bord du HMS/D. En outre, les obstacles résidents dans la zone d'atterrissage peuvent être détectés et classés en temps réel à l'aide d'OWS (*Obstacle Warning System*). Le *situational awareness suite Sferion™* constitue une ligne de produits. Les différentes *features* peuvent être sélectionnées pour la fonction symbolique d'atterrissage en fonction du client et de la plateforme de l'hélicoptère dans laquelle nous devons déployer la solution.

FIGURE *8.1* – *Situational awareness suite Sferion™*

8.1.2 La complexité et la configuration du système

Pour la vérification du *Situational awareness suite Sferion™*, Airbus Defence & Space est confronté aux problèmes suivant :

- La complexité des différents capteurs physiques (*OWS*, *FLIR*, *Radar Altimeter*, *Zonal Radar*).
- La complexité des composants du système de contribution relative aux protocoles de messagerie utilisés *(RS422, ARINC429, Ethernet, MIL-Bus, Discretes)*.
- Les variantes du produit *Sferion™*, pour différentes plateformes et dépendantes des exigences des clients (par exemple, *platform-dependent INS ou MFD*)
- Les *features* en options (par exemple, Laser Altimeter supplémentaire pour une plus grande précision).
- La criticité de haute sécurité → Nécessité de démontrer la conformité aux exigences de certification (*DAL C* selon *RTCA/DO-178B*).
- L'utilisation de produits *COTS* pour les différents capteurs physiques.

8.1.3 Méthodes de développement actuel

Ce qui suit est une liste des méthodes généralement adoptées lors de la conception et de développement d'un système avionique :

- Analyse des exigences textuelles.
- Définition des activités et des stratégies de validation & vérifications nécessaires (par exemple, les tests dynamiques, la vérification, l'inspection et l'analyse statique du code).
- Analyse de traçabilité pour vérifier la cohérence, la compatibilité et l'exhaustivité des exigences.
- Développement manuel des cas et des procédures de test.
- Analyse basée sur la couverture des exigences.
- Exécution semi-automatisée des procédures de test et d'analyse des résultats des tests, y compris l'analyse de couverture structurelle.
- Analyse de l'impact des changements (y compris l'analyse des coûts) pour le changement des exigences, le code, les cas de test et les procédures.

8.2 Les phases d'évaluation

L'expérimentation s'est déroulée en quatre phases :

A - La première phase a permis la définition du périmètre des différentes situations possibles de guidage.

B - La deuxième phase a été réalisée à partir de modèle de variabilité OVM, qui contient les *features* de la ligne de produits à tester.

C - La troisième phase a consisté à réaliser le modèle d'usage de ce système avec les différentes situations de guidage et les options qui varient d'un produit à l'autre.

D - Dans la dernière phase, l'initialisation des données est nécessaire pour la génération sous MPLM, afin de générer les modèles de tests spécifiques à chaque produit en entrée de la famille des produits.

Les caractéristiques du modèle d'usage et du projet :

Nombre d'exigences : 12,

Taille du modèle de variabilité : 7 points de variation et 16 *features*,

Taille du modèle de test : 18 états, 32 transitions, 5 chaînes et instances de chaînes,

Nombre de produits possibles : 6,

Nombre de produits à tester : 2,

Niveau d'expertise : expert,

Temps de mise en œuvre : 1 jour/ homme.

Le temps de la mise en œuvre couvre l'activité de la réalisation du modèle OVM, le modèle d'usage de la ligne de produits, le *mapping* des *features* avec les exigences et la configuration des variantes à tester.

Deux modèles valides (modèle correspondant au produit à tester) ont été générés avec MPLM, avec un temps moyen de génération de moins d'une seconde.

L'expérimentation s'est déroulée avec un PC de développement standard, un processeur 2.8 GHZ et 8 Go de RAM. Notre partenaire rapporte que les gains apportés avec l'approche présentée sont : l'automatisation du processus, ainsi que la capacité de générer avec plus d'efficacité et de pertinence des modèles de test qui couvrent les exigences des produits en entrée à partir du modèle de test de la ligne de produits. Pour une première phase, Airbus a choisi d'appliquer notre approche sur une petite partie du SUT avant d'évaluer l'approche avec un projet concret.

Problématiques industrielles :

Les problématiques industrielles que MPLM a pour objectif de résoudre sont les suivantes :

- Réduire les coûts de test,
- Améliorer la qualité des produits,
- Réduire les coûts de déploiement.

Les questions suivantes sont tirées des problématiques ci-dessus et constituent la base de l'évaluation que nous avons réalisée :
1) Peut-on réduire les coûts du développement des cas de test ? 2) Peut-on réutiliser les artefacts de test dans les variantes du produit ? 3) Peut-on réduire les coûts de

test résultant des changements dans les exigences ? 4) Peut-on réduire le nombre de défauts présents dans les variantes de produit ? 5) Quel est l'effort supplémentaire nécessaire pour configurer l'environnement de l'outil et pour offrir la formation nécessaire à l'équipe du test ?

8.3 Évaluation

Les questions découlant des problématiques de l'industrie ont été évaluées sur la base de l'expérience acquise lors de la réalisation de l'étude de cas par notre partenaire Airbus Defence & Space.

8.3.1 Réduction des coûts pour le développement de cas de test

Au cours de l'étude de cas, nous avons mesuré les efforts nécessaires pour réaliser un modèle d'usage, afin de comparer ces efforts avec un projet de référence du domaine avionique qui ne s'appliquait pas au *model-based testing*. Nous avons réalisé des entretiens avec les responsables de test afin d'obtenir des informations sur les efforts consacrés au cours du développement des cas de test. Le nombre des exigences couvertes par les cas de test a été compté et normalisé, en utilisant des facteurs de pondération qui permettent d'avoir des exigences de différente complexité. Au cours de l'étude de cas, nous avons mesuré les efforts pour mettre en place le modèle d'usage et l'automatisation de test. Le nombre normalisé des exigences a été déterminé. L'efficacité du développement des cas de test, en d'autres termes l'effort moyen nécessaire pour couvrir une exigence normalisée par le test, a été calculée à la fois pour le projet de référence et pour l'étude de cas. La comparaison de l'efficacité a abouti à une réduction des coûts de 18 % de l'effort de test.

8.3.2 La réutilisation des artefacts de test

Dans le passé, des projets distincts ont été réalisés pour différents clients et par conséquent la conception des cas de test a été effectuée de façon indépendante pour toutes les variantes de produits sans réutilisation systématique. Maintenant, le modèle d'usage est l'artefact principal qui est réutilisé au niveau de la ligne de produits. En outre, nous gérons la réutilisation au niveau du modèle d'usage, plutôt qu'au niveau des cas de test. L'avantage est que beaucoup moins d'artefacts doivent être gérés pour la ligne de produits. Toutefois, afin d'éviter les itérations inutiles, il est obligatoire d'avoir une bonne compréhension de la variabilité de la ligne de produits a priori. C'est utile pour structurer les exigences et le modèle d'usage en fonction de la variabilité et par conséquent pour faciliter la transformation du modèle d'usage de la ligne de produits en variantes du modèle d'usage. Par exemple, les exigences doivent être rédigées d'une manière qui permet de les associer clairement avec des *features*. Les modèles d'usage devraient être développés de telles sorte que les variantes puissent être facilement obtenues (par exemple en utilisant des chemins distincts représentant le comportement variable, des sous-chaînes pour chaque fonctionnalité).

8.3.3 Coût de déploiement

L'approche est basée principalement sur les artefacts déjà établis comme les exigences et les modèles d'usage. Seule l'approche OVM est ajoutée. La complexité du lien entre les modèles de variabilité et les modèles d'usage est réduite au minimum puisque les associations exigences - *feature* sont utilisées. Toutes les extensions d'outils sont fournies par une seule extension, pour l'environnement de l'outil MaTeLo. Ainsi, l'effort supplémentaire pour le déploiement de l'outil est négligeable. En ce qui concerne la formation, il serait recommandé de fournir une formation de base sur l'ingénierie des lignes de produits pour mettre l'équipe au courant des nouveaux sujets tels que le cadre de la ligne de produits et la modélisation de la variabilité.

8.3.4 Pistes d'amélioration

Nous avons appris que l'approche ouvre des pistes pour d'éventuelles extensions et plus de recherches.

Réduction du nombre de défauts

Notre approche consiste à effectuer des tests pour tous les produits. Différentes stratégies de génération des cas de test peuvent être utilisées dans les variantes du produit donnant des cas de test différents. Par conséquent, la couverture des critères de test est augmentée et par conséquent la possibilité de détecter des défauts. L'approche pourrait être étendue pour soutenir l'analyse des résultats des tests. Par exemple, si un test échoue dans un seul produit, un indice peut être accordé à la ligne de produits et aux autres produits dérivés, afin d'éviter des tests inutiles tant que le défaut n'est pas supprimé.

Prise en charge des changements

Dans un environnement réaliste, les exigences sont susceptibles de changer. Basé sur la traçabilité entre les exigences et les éléments du modèle d'usage, l'impact des changements d'exigences peut être analysé. Actuellement, il n'y a pas de séparation dans la variante du modèle d'usage entre la partie qui est dérivée à partir du modèle d'usage de la ligne de produits et la partie spécifique au produit. Cela implique que l'impact du changement doit être analysé dans chaque variante du modèle d'usage ; des stratégies de tests supplémentaires peuvent donc être considérées (voir par exemple, [LSKL12]).

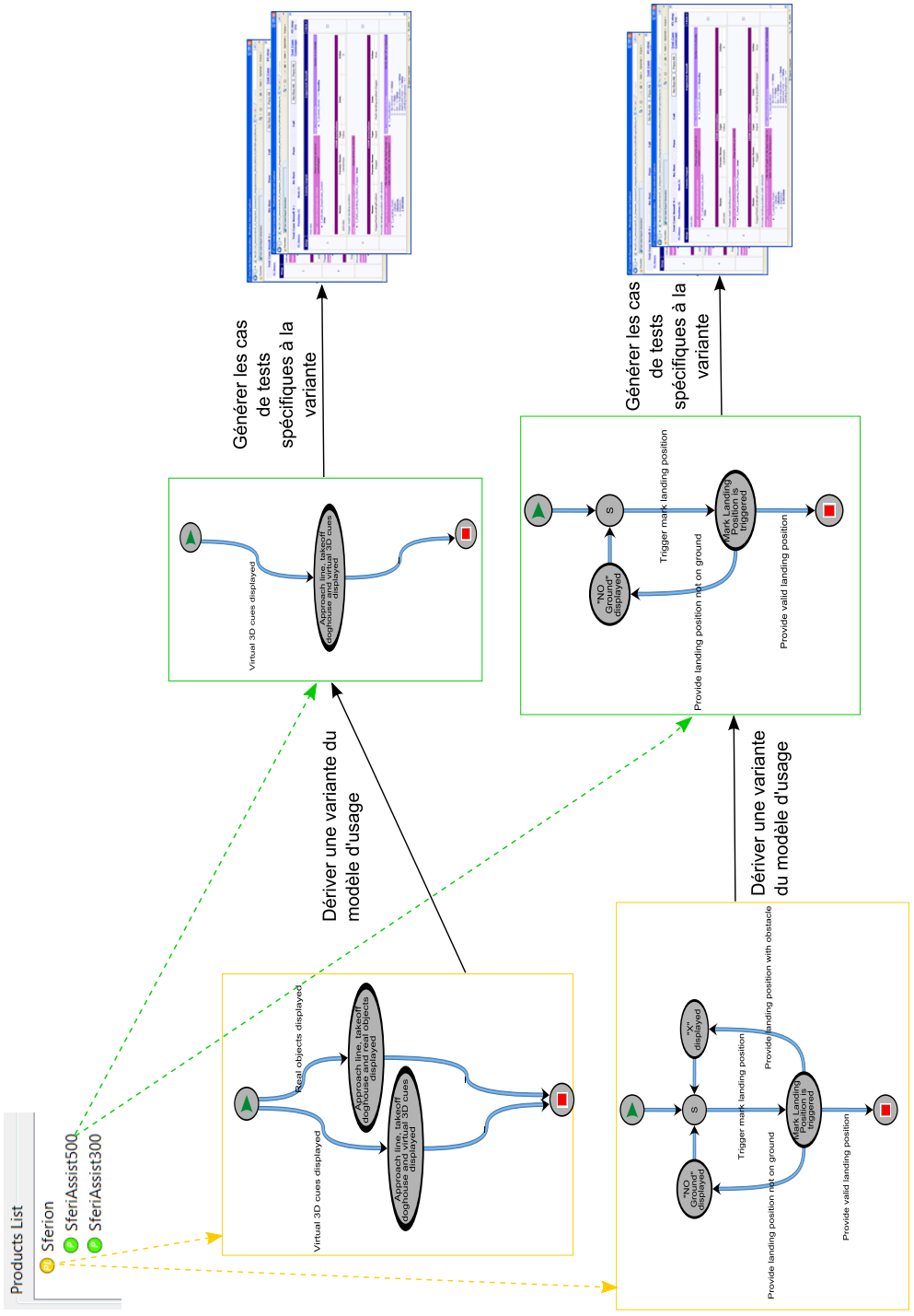


FIGURE *8.2 – Dérivation des variantes du modèle d'usage*

8.4 Évaluation de l'algorithme 3

Nous évaluons l'algorithme 3 sur trois cas d'étude issus de l'industrie. Le premier cas d'étude est la ligne de produits **Situational awareness suite Sferion™**, le deuxième cas d'étude est l'exemple du **Tableau de bord** que nous avons évalué dans son état d'origine. Le dernier est un cas d'étude qui concerne la validation du contrôle moteur d'automobile.

Contrôle-moteur d'autombile

Le constructeur développe plusieurs variantes du moteur d'automobile pour une variante de catégorie de véhicules. Un modèle de voiture peut être compatible avec plusieurs variantes du moteur. Cependant, plusieurs variantes de pédales sont possibles pour un type de moteur et différents types de bus de communication sont disponibles selon la configuration du moteur. Ce sont aussi des types d'injecteur différents pour le même modèle de moteur. En plus, des options peuvent être intégrées comme l'ESP et ABS.

Le Tableau 8.1 présente les caractéristiques du modèle d'usage et du modèle OVM des trois cas d'utilisations. Nous avons sélectionné trois tailles différentes du modèle d'usage, moins de 100 transitions et de 5 chaînes, moins de 1000 transitions et de 50 chaînes et plus de 1000 transitions et de 50 chaînes. La dernière tranche correspond plus à la taille des modèles réalisés en pratique, par contre, le modèle du tableau de bord est grand par rapport aux modèles habituels des clients d'ALL4TEC. Pour rappel, une chaîne est une partie de modèle d'usage référencée par un état d'une autre chaîne au niveau supérieur. Une instance chaîne est une chaîne référencée plusieurs fois dans le modèle d'usage par d'autres états de d'autres chaînes au niveau supérieur.

Caractéristiques	Sferion	Moteur des véhicules	Tableau de bord
Nombre d'exigences	12	74	261
Nombre de *features*	16	19	18
Nombre de Points de variation	7	7	6
Nombre de produits à tester	2	3	12
Nombre d'états	18	148	7580
Nombre de transitions	32	245	14853
Nombre de chaines	5	15	63
Nombre d'instances de chaines	5	15	6392
Niveau d'expertise	expert	expert	expert

TABLE *8.1 – Caractéristiques des lignes de produits (Modèle d'usage et modèle de variabilité)*

Pour les trois cas d'étude nous avons exécuté l'algorithme 3 et nous avons obtenu les résultats représentés dans le tableau 8.2.

Caractéristiques et détails de génération	Sferion	Moteur des véhicules	Tableau de bord
Nombre de transitions supprimées	3	76	166
Nombre de branches cassées	3	102	232
Nombre de chaines supprimées	0	5	13
Nombre d'instances de chaines supprimées	5	6	6378
Nombre de produits testés	2	3	15
Nombre de modèles générés	2	3	15
Nombre d'états du modèle généré	15	42	26
Nombre de transitions du modèle généré	26	64	45
Nombre de chaines du modèle généré	5	10	50
Nombre d'instances de chaines du modèle généré	5	9	14
Temps de dérivation de chaque modèle	~1 second	~1 second	~3 seconds

TABLE *8.2 – Résultat d'algorithme 3*

8.5 Discussion des résultats

Le tableau 8.2 présente les résultats obtenus de la dérivation automatique des variantes du modèle d'usage des cas d'étude présentés. Les colonnes du tableau représentent les caractéristiques des cas d'étude choisis. Les lignes du tableau 8.2 représentent les caractéristiques du modèle d'usage en termes de nombre d'états, de transitions et de chaînes ainsi que de nombre d'éléments supprimés et impactés.

Nous nous basons essentiellement sur les caractéristiques des modèles d'usage générés, pour les comparer avec les caractéristiques des modèles d'usage d'origine présentés dans le tableau 8.1. Pour rappel, un modèle d'usage est constitué de branches composées de transitions. L'algorithme 3 identifie les éléments du modèle d'usage correspondant à la variante sélectionnée, puis dérive le modèle équivalent. Lors de la dérivation, l'algorithme détecte un ensemble de branches cassées. Ces branches cassées ne constituent plus un chemin valide allant de l'état de début à l'état final. Ces cas incohérents rendent le modèle inutilisable, il est donc nécessaire de les réparer afin d'obtenir un modèle d'usage valide et utilisable. Nous réparons un certain nombre de branches cassées dans la variante du modèle d'usage dérivée. Ce nombre dépend du nombre de cas incohérent détectés, c'est-à-dire, si une transition à conserver composant la branche cassée est supprimée, l'algorithme répare la branche en annulant la suppression, puis signal l'in-

cohérence dans le journal d'exécution. Cependant, les temps de génération restent très faibles. Les autres branches cassées ne sont pas réparées, car elles ne contiennent pas de transitions à conserver, ainsi donc leur suppression est nécessaire pour avoir un modèle cohérent avec une structure valide. Nous observons que tant que le nombre de transitions supprimées est important, le nombre de branches impactées est aussi important. Nous remarquons aussi que le nombre de chaînes supprimées est plus important quand le nombre de branches cassées dépasse les 100 unités. Dans ce cas-là, le nombre final de chaînes est réduit ainsi que le nombre d'états et de transitions restantes. Il s'agit ici, de supprimer toutes les incohérences et de réparer le modèle. Par la suite, l'algorithme corrige les probabilités des profils d'usage associés au modèle, en appliquant la règle suivante, la somme des probabilités des transitions sortantes d'un état doivent être égal à 1.

La génération des cas de test dépend des probabilités de chaque profil associé au modèle d'usage, donc il est important de corriger les probabilités du modèle réduit. Ce processus de mise à jour est automatisé dans notre approche, pour enlever le risque d'échec de la génération des cas de test à cause d'une simple erreur de mise à jour manuelle des probabilités dans le modèle réduit. De plus, mettre à jour manuellement les probabilités est très difficile dans le cas d'un modèle de taille importante, comme dans le cas du modèle des tableaux de bord. Ces nouvelles probabilités associées au modèle d'usage généré représentent l'usage du système dans une situation équiprobable (une expérience où tous les événements ont la même probabilité d'être réalisés).

En outre, l'algorithme génère un modèle réduit avec une structure valide et des probabilités correctes pour les trois tailles des modèles. Ce tableau croisé semble nous indiquer que le nombre d'états, transitions et chaînes du modèle dérivé est influencé par le nombre de branches cassées et non directement par le nombre de transitions supprimées. Les résultats de l'algorithme 3 obtenus avec le cas d'étude tableaux de bord, permettent de confirmer la capacité de MPLM à passer à l'échelle, puisque le modèle des tableaux de bord est suffisamment gros pour en discuter. Pour chaque variante du modèle d'usage générée, nous avons vérifié à la main, que tous les comportements du produit à tester sont bien dans le modèle généré et que ce modèle ne comporte pas plus de comportements que ceux à tester. Le constat est que l'algorithme réussi à générer le bon modèle à chaque génération, à condition que le modèle d'usage de la ligne de produits ne contienne pas d'incohérences. Dans le cas contraire, les cas incohérents sont détectés et après une mise à jour du modèle d'usage, l'algorithme 3 peut générer le bon modèle pour le produit à tester.

Pour chaque génération, le taux moyen de réduction du nombre d'états est de ~63 % et de ~65 % pour les transitions en moins dans le modèle réduit. Ces résultats obtenus montrent que l'effort pour dériver une variante du modèle d'usage est important, en particulier pour un grand modèle comme le cas d'étude tableau de bord, où il faut traiter 14 853 transitions. Par conséquent, un processus automatisé de la dérivation, peut être la solution pour dériver un modèle réduit valide et en moins d'effort. L'algorithme de dérivation proposé, réduit d'autant l'effort de test pour un produit d'une famille de produits. En outre, plus besoin de trier à la main ou à travers des profils, les cas de test qui ne correspondent pas au produit à tester.

8.6 Résumé

Notre approche permet d'exercer le *model-based testing* dans le cadre de l'ingénierie des lignes de produits. Les cas de test peuvent être générés automatiquement non seulement pour un produit, mais pour de nombreuses variantes à partir d'un seul modèle d'usage d'une ligne de produits.

La solution est implémentée dans l'outil MPLM qui étend MaTeLo un outil du *model-based testing*. Nous avons évalué MPLM dans le cadre du projet européen MBAT avec l'étude de cas industriel *Situational awareness suite Sferion™* du secteur de l'aéronautique. L'évaluation a été réalisée avec notre partenaire industriel Airbus Défense & Space. Elle nous a permis d'avoir un retour industriel sur l'utilisabilité et l'efficacité de l'approche. Notre partenaire rapporte que l'approche a permis de réduire considérablement l'effort nécessaire pour isoler les cas de test pour un produit particulier de la ligne de produits à tester. Il rapporte également que le choix de maintenir le modèle OVM distinct des modèles d'usage déjà établis, permet de faciliter le déploiement de la solution et de réduire la complexité de la mise en place de la solution.

Quatrième partie

Conclusion et Perspectives

Chapitre 9

Conclusion et Perspectives

9.1 Conclusion

Le *model-based testing* est une technique qui vise à réduire les efforts de test en générant automatiquement des cas de test pour un système complexe. Par contre, la diversité des exigences suscitées par différents clients conduit à un développement de systèmes de plus en plus complexes et en de nombreuses variantes. Ces variantes du système forment une ligne de produits. L'ingénierie des lignes de produits est utilisée dans l'industrie pour parvenir à un développement logiciel plus efficace. Cependant, le processus du *model-based testing* n'est pas adapté au test des lignes de produits. En conséquence, la vérification dans l'ingénierie des lignes de produits consiste aujourd'hui à vérifier chaque produit indépendamment les uns des autres, réduisant considérablement les bénéfices attendus de la réutilisation accrue proposée par cette approche et de la puissance du MBT.

Dans cette thèse, nous introduisons une approche qui permet d'utiliser le *model-based testing* dans un contexte de ligne de produits, afin de générer automatiquement les cas de test pour les variantes des produits de la ligne de produits. L'objectif de notre méthodologie est de réduire l'effort de test des lignes de produits, la figure 9.1 présente une synthèse de notre approche.

Dans ce travail, nous utilisons le modèle d'usage de MaTeLo comme modèle de test. Le modèle d'usage est un modèle d'états - transitions probabiliste basé sur les chaînes de Markov, permettant de représenter le comportement attendu des *features* d'un système sous test. Nous proposons de réaliser un seul et unique modèle d'usage pour une ligne de produits. Néanmoins, pour concevoir un modèle de test de la ligne de produits, le formalisme de modèle d'usage doit être capable de représenter les points communs et les points de variation de la ligne de produits. Pour répondre à notre besoin, nous introduisons une adaptation dans le chapitre 5.2. Cette introduction de la variabilité dans le modèle d'usage va permettre la dérivation automatique des variantes de modèle d'usage pour un ensemble de produits similaires à partir du modèle d'usage de la ligne de produits. Au cours de la réalisation du modèle d'usage, les exigences fonctionnelles sont associées aux transitions.

Nous utilisons dans notre approche OVM (*Orthogonal variability Model*), un formalisme fournissant une structure à plat des exigences de la ligne de produits, utilisé pour représenter graphiquement la variabilité de la ligne de produits. OVM permet de se concentrer uniquement sur la variabilité, afin de la gérer indépendamment des autres artefacts de test. La sélection des *features* selon les dépendances et les contraintes donne lieu à des variantes de configurations. En outre, nous proposons une amélioration de la formalisation d'OVM proposée par Metzger et al. [MPH+07] pour exprimer les liens entre les *features* et les exigences fonctionnelles.

Pour le test de la ligne de produits, nous configurons un ensemble de produits selon le modèle OVM. Pour pouvoir générer les cas de test à partir du modèle d'usage de la ligne de produits pour chaque variante, nous proposons de dériver des variantes du modèle d'usage puis réutiliser le processus traditionnel du *model-based testing*. Pour cela nous avons présenté trois contributions :

La *première contribution* consiste à associer les exigences fonctionnelles aux *features* du modèle OVM. À l'aide de cette association, les correspondances entre les *features* et les transitions du modèle d'usage sont déduites automatiquement.

Nous décrivons dans la *deuxième contribution* un algorithme de dérivation automatique des variantes du modèle d'usage. L'algorithme utilise les liens établis entre OVM et le modèle d'usage pour identifier les parties correspondantes à chaque *feature* composant une configuration à tester. Le processus de dérivation d'une variante du modèle d'usage est basé sur un algorithme qui consiste à sélectionner les exigences relatives à chaque *feature* composant une configuration, ensuite, il sélectionne toutes les transitions associées à l'ensemble d'exigences identifié. Enfin, l'algorithme élague le modèle d'usage de la ligne de produits en fonction des exigences et des transitions identifiées, en dérivant ainsi, un modèle d'usage valide équivalent à la spécification de la configuration sélectionnée. Le modèle d'usage dérivé représente les scénarios de test d'une variante d'un produit de la ligne de produits. Nous réutilisons par la suite le processus traditionnel du *model-based testing* en se basant sur l'outil MaTeLo pour dériver les cas de test.

La *troisième contribution* est l'implémentation de notre approche par l'outil MaTeLo Product Line Manager (MPLM) que nous avons développé dans le cadre de la thèse pour évaluer notre approche avec plusieurs projets industriels.

Nous avons évalué l'approche dans le contexte du projet Européen MBAT avec notre partenaire Airbus Defence & Space. L'évaluation de l'approche a montré que le coût du test de la ligne de produits est considérablement réduit grâce à la réutilisation du modèle d'usage de la ligne de produits pour générer les cas de test des variants d'une famille de produits, puisque les variantes du modèle de test dérivées sont examinées et utilisées par de nombreux produits dans différents contextes et divers scénarios, ce qui conduit à une meilleure qualité. Les modèles de test sont réutilisés pour chaque nouveau produit, plutôt que de partir de zéro. Le déploiement est facilité grâce au choix de maintenir le modèle OVM séparé des modèles d'usage déjà établis.

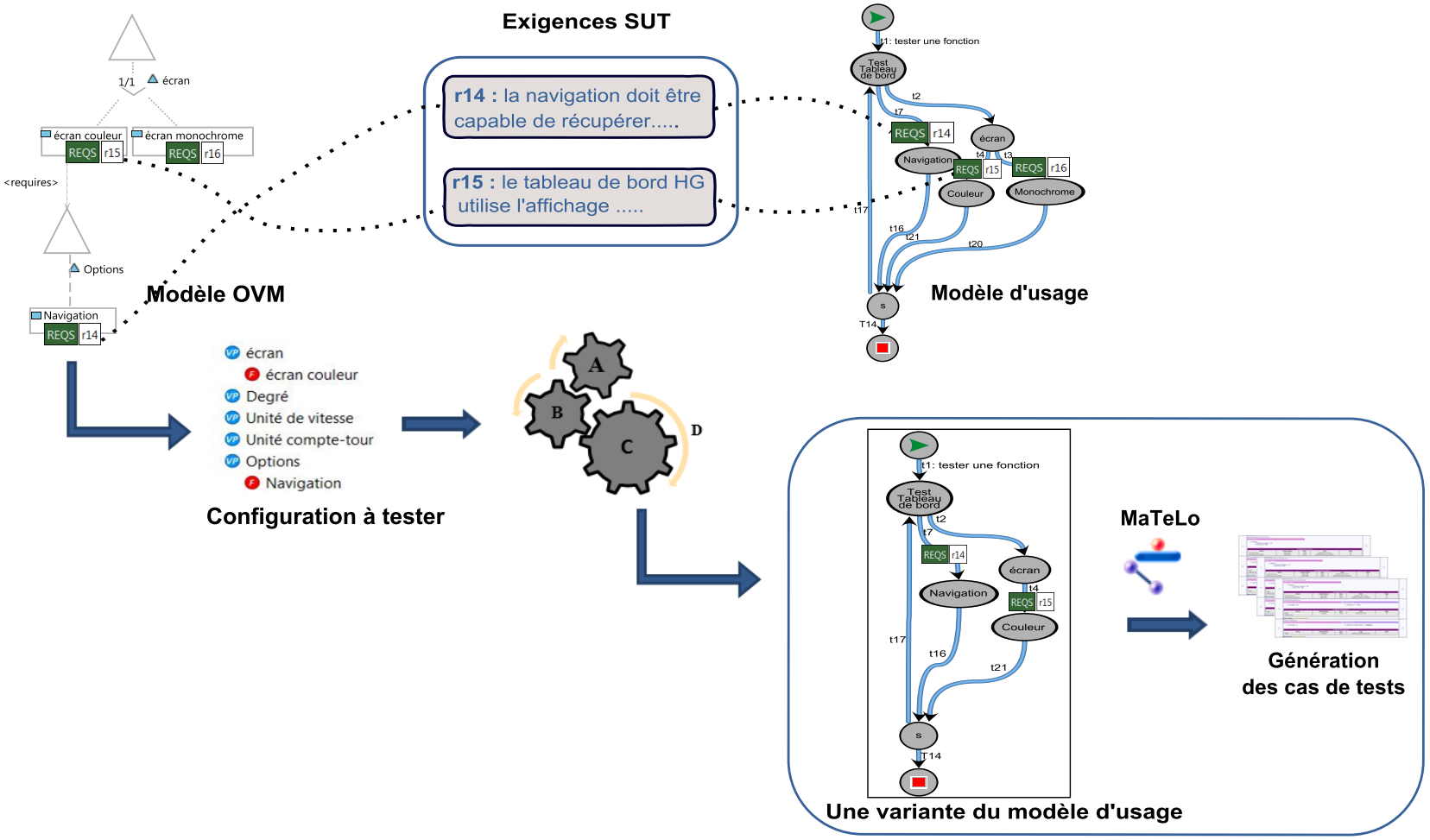


FIGURE 9.1 – *Approche globale de la dérivation des variantes du modèle d'usage*

9.2 Perspective

Plusieurs travaux restent à réaliser dans le cadre de notre approche de test de ligne de produits basé sur les modèles et également pour l'extension de notre méthodologie dans le but d'optimiser le processus. Dans cette section, nous présentons les perspectives de recherche identifiées.

9.2.1 Évaluation industrielle

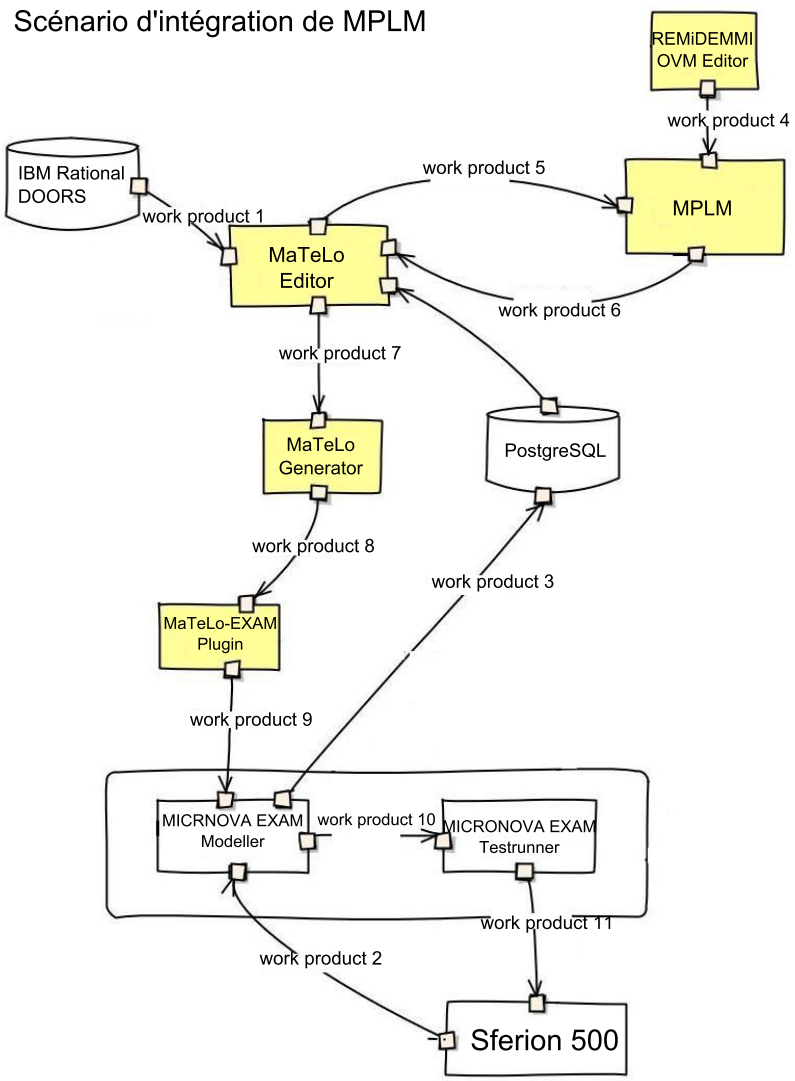


FIGURE *9.2 – Intégration de MPLM dans le processus d'industrialisation*

Nous avons amélioré la version actuelle de l'outil MPLM en l'intégrant dans MaTeLo, afin de lancer une étude empirique à plus grande échelle pour évaluer l'approche avec nos partenaires du projet européen MBAT et nos clients.

Notre objectif est de pousser l'adoption industrielle de l'approche, en évaluant les données empiriques recueillies, la complexité de la mise en place et la complexité de la mise en œuvre de l'approche pour différents cas d'étude industriels. En outre, nous voulons récolter plus de retour d'expériences des utilisateurs et confirmer les concepts théoriques, pour vue d'améliorer les points problématiques identifiés dans notre approche.

L'objectif est de générer avec notre approche des cas de test et les exécuter sur une variante du système à tester pour laquelle nous avons dérivé le modèle d'usage correspondant.

La figure 9.2 illustre le scénario d'intégration de MPLM dans la chaîne d'outillage réalisée par Airbus Defence & Space pour le cas d'étude *Situational awareness suite Sferion™* présenté dans la section 8.1.1. Nous travaillons sur cette intégration, pour générer les cas de test des variantes *Sferion™* et les exécuter à l'aide de l'outil d'exécution automatique des tests *EXAM*[1]. Le but est d'évaluer notre méthodologie sur le produit final et d'obtenir les résultats d'exécution des cas de test, puis d'en tirer les conclusions.

9.2.2 Intégration de FaMa-OVM

Dans notre cas, Eclipse RCP apporte à MPLM la capacité d'être interconnecté avec des outils d'analyse formelle utilisés dans le cadre de l'ingénierie des lignes de produits pour faire face à l'explosion combinatoire. En outre, ces paramètres peuvent aider les testeurs à adopter la réutilisation dans le processus de test pour les grands projets.

La modélisation de la variabilité de la ligne de produits dans notre approche est basée sur OVM. Notre méthodologie n'intègre pas de méthode automatique dédiée à l'analyse de la variabilité d'un modèle OVM. La configuration des variantes de produits d'une ligne de produits à tester est réalisée manuellement via MPLM. FaMa-OVM [RFBRC+12] est un outil d'analyser du modèle OVM avec ou sans attributs de qualité associés aux *features*. FaMa est capable de vérifier une expression de *features* et de dériver un ensemble de configuration valides selon les conditions et critères en entrée. Nous travaillerons sur l'intégration de FaMa dans notre outil, afin de générer automatiquement un ensemble de configurations à tester selon des contraintes spécifiques, en plus de l'utiliser pour vérifier les configurations créées.

9.2.3 Extension de l'approche

À l'aide du modèle d'usage dérivé, nous pouvons générer de nombreux cas de test selon les critères de couvertures appliqués à la génération. Cependant, ils ne sont pas tous aussi importants pour trouver des bugs. Nous collaborons avec l'équipe projet

1. Un système d'automatisation d'exécution des cas de test développé par Volkswagen AG

PReCISE de l'université Namur en Belgique, pour prioriser les cas de test générés et prioriser les produits d'une famille de produits pour le test à partir de ces cas de test.

Nous travaillons sur la combinaison de la méthode formelle FTS (*Featured Transition System*) avec notre approche. Le FTS est un modèle de transition étiqueté avec des *features* représentant l'ensemble des traces d'exécution possibles d'une ligne de produits. Le FTS est conçu manuellement.

Cette extension consiste à exploiter les liens établis entre les *features* et les transitions du modèle d'usage à l'aide des exigences pour synthétiser automatiquement le FTS équivalent au modèle d'usage de la ligne de produits. L'objectif est de générer un ensemble de cas de test à partir du modèle d'usage de la ligne de produits et filtrer les cas de test pour en sélectionner que ceux qui sont valides, c'est-à-dire, un cas de test qui correspond à une expression de *features* valide. Le principe est de déduire l'expression de *features* qui correspond au cas de test en entrée et de vérifier avec FaMa-OVM si cette expression et valide par rapport au modèle OVM. Si oui, le cas de test est conservé. Ensuite, nous allons à l'aide du tuple (cas de test(CT), produits de la ligne de produits, probabilités des CT) prioriser les produits de la ligne de produits pour le test et appliquer l'approche de Devroey et al. [DPC+14].

Glossaire

Acronymes

IPL : Ingénierie des lignes de produits

PL : Ligne de produits

MBT : Model-based Testing

SUT : System Under Test

V&V : Vérification et Validation

MaTeLo : Markov Test Logic

MPLM : MaTeLo Product Line Manager

MBAT : Combined Model-based Analysis & Testing

OVM : Orthogonal Variability Model

CT : Cas de test

FTS : Featured Transition System

OCL : Object Constraint Language

FODA : Feature-Oriented Domain Analysis

CVL : Common Variability Language

MOF : Meta-Object Facility

DSL : Domain-Specific Languages

FAMILIAR : for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning

TVL : Text-based Variability Language

Sémantiques

Model-based Testing : Test basé sur les modèles.

Feature : Caractéristique qui distingue un produit d'un autre.

Framework : Plan

Mapping : Mise en correspondance

Template : maquette
Optionnal : optionnel
Variation point : point de variation
Variant : Variante
And : Et (conjonction)
Or : Ou (disjonction)
Alternative choice : Choix alternatif
Mandatory : Obligatoire
Requires : Exige
Excludes : Exclut
Slicing : Découpage
Checking : Vérification

Bibliographie

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Managing feature models with familiar : a demonstration of the language and its tool support. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 91–96. ACM, 2011.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Familiar : A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6) :657–681, 2013.

[BBN04] Mark Blackburn, Robert Busser, and Aaron Nauman. Why model-based test automation is different and what you should know to get started. In *International conference on practical software quality and testing*, pages 212–232, 2004.

[BCFH10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems, Linz, Austria, January*, pages 159–162, 2010.

[BG03] Antonia Bertolino and Stefania Gnesi. Use case-based testing of product lines. *ACM SIGSOFT Software Engineering Notes*, 28(5) :355–358, 2003.

[BG04] Antonia Bertolino and Stefania Gnesi. Pluto : A test methodology for product families. In *Software Product-Family Engineering*, pages 181–197. Springer, 2004.

[BGGB02] Martin Becker, Lars Geyer, Andreas Gilbert, and Karsten Becker. Comprehensive variability modelling to facilitate efficient variability treatment. In *Software Product-Family Engineering*, pages 294–303. Springer, 2002.

[BLLP04] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications : Gsm 11-11 standard case study. *Software : Practice and Experience*, 34(10) :915–948, 2004.

[BM11] Rex Black and Jamie L Mitchell. *Advanced Software Testing-Vol. 3 : Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. Rocky Nook, 2011.

[Bos00] Jan Bosch. *Design and use of software architectures : adopting and evolving a product-line approach*. Pearson Education, 2000.

[BR97] Grady Booch and James Rumbaugh. *Unified Method for Object-Oriented Development Version 1.0*. Rational Software Corporation, 1997.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of tvl. *Science of Computer Programming*, 76(12) :1130–1143, 2011.

[CCR10] Isis Cabral, Myra B Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *Software Product Lines : Going Beyond*, pages 241–255. Springer, 2010.

[CCS+13] Andreas Classen, Maxime Cordy, P-Y Schobbens, Patrick Heymans, Axel Legay, and J-F Raskin. Featured transition systems : Foundations for verifying variability-intensive systems and their application to ltl model checking. *Software Engineering, IEEE Transactions on*, 39(8) :1069–1089, 2013.

[CDS06] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63. ACM, 2006.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature : A requirements engineering perspective. In *Fundamental Approaches to Software Engineering*, pages 16–30. Springer, 2008.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[Cin75] E. Cinlar. *Introduction to Stochastic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints : A progress report. In *International Workshop on Software Factories*, pages 16–20, 2005.

[CN02] Paul Clements and Linda Northrop. *Software product lines : practices and patterns*, volume 59. Addison-Wesley Reading, 2002.

[Con04] Chris Condron. A domain approach to test automation of product lines. In *International Workshop on Software Product Line Testing*, volume 2004, page 27. Citeseer, 2004.

[CZZM05] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 31–40. IEEE, 2005.

[Dar91] Ian F Darwin. *Checking C Programs with lint*. " O'Reilly Media, Inc.", 1991.

[DBI12] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. Jsxm : A tool for automated test generation. In *Software Engineering and Formal Methods*, pages 352–366. Springer, 2012.

[DFJ54] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4) :393–410, 1954.

[DJK+99] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294. ACM, 1999.

[dMSNRdCM+11] Paulo Anselmo da Mota Silveira Neto, Per Runeson, Ivan do Carmo Machado, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Emelie Engström. Testing software product lines. *IEEE Software*, 28(5) :16–20, 2011.

[DPC+14] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards statistical prioritization for software product lines testing. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, page 10. ACM, 2014.

[ER11] Emelie Engström and Per Runeson. Software product line testing–a systematic mapping study. *Information and Software Technology*, 53(1) :2–13, 2011.

[FHP02] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. Using a model-based test generator to test for standard conformance. *IBM systems journal*, 41(1) :89–110, 2002.

[GAA+14] José A. Galindo, Mauricio Alférez, Mathieu Acher, Benoit Baudry, and David Benavides. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 293–303, 2014.

[GKK+07] Dharmalingam Ganesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. Comparing costs and benefits of different test strategies for a software product line : A study from testo ag. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 74–83. IEEE, 2007.

[Gom05] Hassan Gomaa. Designing software product lines with uml. In *2012 35th Annual IEEE Software Engineering Workshop*, pages 160–216. IEEE Computer Society, 2005.

[GWT+14] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems – a systematic literature review. *Software Engineering, IEEE Transactions on*, 40(3) :282–306, March 2014.

[HBG11] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen : Automatic generation of pairwise test configurations from feature models. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 120–129. IEEE, 2011.

[HBG12] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Managing execution environment variability during software testing : An industrial experience. In *ICTSS*, pages 24–38, 2012.

[HMPO+08] Oystein Haugen, Birger Moller-Pedersen, Jon Oldevik, Gøran K Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 139–148. IEEE, 2008.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM.

[HS12] Benoit Baudry Hamza Samih, Hélène Le Guen. Extension du model-based testing pour la prise en compte de la variabilité dans les systèmes complexes. In *Poster presented at Journées Nationales du GDR Génie de la Programmation et du Logiciel*, 2012.

[HS13] Ralf Bogusch Hamza Samih. An approach of combining model-based testing with product family management. In *User Conference on Advanced Automated Testing*, 2013.

[Hui07] Antti Huima. Implementing conformiq qtronic. In *Testing of Software and Communicating Systems*, pages 1–12. Springer, 2007.

[HVR04] Jean Hartmann, Marlon Vieira, and Axel Ruder. A uml-based approach for validating product lines. In *Intl. Workshop on Software Product Line Testing (SPLiT), Avaya Labs Technical Report*, pages 58–64. Citeseer, 2004.

[JWEG07] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *Model Driven Engineering Languages and Systems*, pages 151–165. Springer, 2007.

[KBBK10] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In *Runtime Verification*, pages 285–299. Springer, 2010.

[KBK11] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 57–68. ACM, 2011.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form : A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5 :143–168, January 1998.

[KL13] Kyo Chul Kang and Hyesun Lee. Variability modeling. In *Systems and Software Variability Management*, pages 25–42. 2013.

[KM] R. Kolb and D. Muthig. Techniques and strategies for testing component-based software and product lines, development of component-based information systems. In *Advances in Management Information Systems*, volume v2.

[KS] Peter Knauber and Johannes Schneider. Tracing variability from implementation to test using aspect-oriented programming.

[KS60] John G Kemeny and James Laurie Snell. *Finite markov chains*, volume 356. van Nostrand Princeton, NJ, 1960.

[Lew04] William E Lewis. *Software testing and continuous quality improvement*. CRC press, 2004.

[LG05] Hélène Le Guen. *Validation d'un logiciel par le test statistique d'usage : de la modélisation à la décision de livraison*. PhD thesis, Rennes 1, 2005.

[LGT03] Helene Le Guen and Thomas Thelin. Practical experiences with statistical usage testing. In *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*, pages 87–93. IEEE, 2003.

[LKL02] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Software Reuse : Methods, Techniques, and Tools*, pages 62–77. Springer, 2002.

[LKL12] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 31–40. ACM, 2012.

[LM02] Hélène Le Guen and R. Marie. Visiting probabilities in non irreducible Markov chains with strongly connected components. In *Proceedings of the 9th International Conference on Analytical and Stochastic Modelling Techniques (ASMT'02)*, pages 548–552, 2002.

[LMP10] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality assurance in the presence of variability. In *Intentional Perspectives on Information Systems Engineering*, pages 319–333. Springer, 2010.

[LMT04] Hélène Le Guen, Raymond Marie, and Thomas Thelin. Reliability estimation for statistical usage testing using markov chains. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 54–65. IEEE, 2004.

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs*, pages 67–82. Springer, 2012.

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action : The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[McG01] John D. McGregor. Testing a software product line, 2001.

[McG08] John D. McGregor. Toward a fault model for software product lines. In *SPLC (2)*, pages 157–162, 2008.

[Met07] Andreas Metzger. Quality issues in software product lines : Feature interactions and beyond. In *ICFI*, pages 1–12, 2007.

[Mis06] Satish Mishra. Specification based software product line testing : a case study. *Concurrency, Specification and Programming*, 2006.

[MP07] Andreas Metzger and Klaus Pohl. Variability management in software product line engineering. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 186–187. IEEE Computer Society, 2007.

[MP14] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management : achievements and challenges. In *Proceedings of the on Future of Software Engineering*, pages 70–84. ACM, 2014.

[MPH+07] Andreas Metzger, Klaus Pohl, Patrick Heymans, P Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines : A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 243–253. IEEE, 2007.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[MSM04] John D McGregor, Prakash Sodhani, and Sai Madhavapeddi. Testing variability in a software product line. In *Proceedings of the International Workshop on Software Product Line Testing, Avaya Labs, ALR-2004-031*, pages 45–50. Citeseer, 2004.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[NFLTJ04] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. A requirement-based approach to test product families. In *Software Product-Family Engineering*, pages 198–210. Springer, 2004.

[Nie10] Nielson Hanne Riis Hankin Chris. Nielson, Flemming. *Principles of program analysis*. Springer, Berlin ; New York, 2010.

[NL11] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2) :11, 2011.

[OG09] Erika Mir Olimpiew and Hassan Gomaa. Reusable model-based testing. In *Formal Foundations of Reuse and Domain Engineering*, pages 76–85. Springer, 2009.

[Oli08] Erika Mir Olimpiew. Model-based testing for software product lines. 2008.

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Lines : Going Beyond*, pages 196–210. Springer, 2010.

[OZML11] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite : tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 79–82. ACM, 2011.

[PBVDL05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. Software product line engineering : Foundations, principles and techniques. *Springer*, 10 :3–540, 2005.

[POS+12] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise testing for software product lines : comparison of two approaches. *Software Quality Journal*, 20(3-4) :605–643, 2012.

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 459–468. IEEE, 2010.

[RB11] Youssef Ridene and Franck Barbier. A model-driven approach for automating mobile applications testing. In *Proceedings of the 5th European Conference on Software Architecture : Companion Volume*, page 9. ACM, 2011.

[RFBRC09] Fabricia Roos-Frantz, David Benavides, and Antonio Ruiz-Cortés. Feature model to orthogonal variability model transformation towards interoperability between tools. *KISS@. ASE, New Zealand*, 2009.

[RFBRC+12] Fabricia Roos-Frantz, David Benavides, Antonio Ruiz-Cortés, André Heuer, and Kim Lauenroth. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal*, 20(3-4) :519–565, 2012.

[RKPR05] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. In *Advanced Information Systems Engineering*, pages 519–534. Springer, 2005.

[RMP06] Sacha Reis, Andreas Metzger, and Klaus Pohl. A reuse technique for performance testing of software product lines. In *Proceedings of the International. Workshop on Software Product Line Testing, Mannheim University of Applied Sciences, Report*, number 003.06, pages 5–10, 2006.

[SAB+14] Hamza Samih, Mathieu Acher, Ralf Bogusch, Hélène Le Guen, and Benoit Baudry. Deriving Usage Model Variants for Model-based Testing : An Industrial Case Study. In IEEE, editor, *2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS 2014)*, Tianjin, Chine, August 2014.

[Sam12] H. Samih. Relating variability modeling and model-based testing for software product lines testing. In *ICTSS'12*, Aalborg, Danemark, November 2012.

[Sam14] Hamza Samih. Étendre le test basé sur des modèles d'usage pour prendre en compte la variabilité. *Technique et Science Informatiques*, 33(3) :203–229, 2014.

[SB14] Hamza Samih and Ralf Bogusch. Mplm-matelo product line manager :[relating variability modelling and model-based testing]. In

Proceedings of the 18th International Software Product Line Conference : Companion Volume for Workshops, Demonstrations and Tools-Volume 2, pages 138–142. ACM, 2014.

[Sep90] Approved September. Ieee standard glossary of software engineering terminology. *Office*, 121990(1) :1, 1990.

[SLB+14] Hamza Samih, Hélène Le Guen, Ralf Bogusch, Mathieu Acher, and Benoit Baudry. An approach to derive usage models variants for model-based testing. In *Testing Software and Systems - 26th IFIP WG 6.1 International Conference, ICTSS 2014, Madrid, Spain, September 23-25, 2014. Proceedings*, pages 80–96, 2014.

[SLS11] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations : a study guide for the certified tester exam*. Rocky Nook, 2011.

[SMJU10] Monalisa Sarma, PVR Murthy, Sylvia Jell, and Andreas Ulrich. Model-based testing in industry : a case study with two mbt tools. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 87–90. ACM, 2010.

[SOS11] G. Engels S. Oster, A. Wöbbeke and A. Schürr. *Model-Based Testing for Embedded Systems*, chapter Model-based software product lines testing survey, pages 339–382. CRC Press, 2011.

[SVGB05] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software : Practice and Experience*, 35(8) :705–754, 2005.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1) :6, 2014.

[TTK04] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing : a survey. *ACM SIGSOFT Software Engineering Notes*, 29(2) :12–12, 2004.

[UL07] Mark Utting and Bruno Legeard. *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2007.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5) :297–312, 2012.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal methods and testing*, pages 39–76. Springer, 2008.

[VdL02] Frank Van der Linden. Software product families in europe : the esaps & cafe projects. *IEEE software*, 19(4) :41–49, 2002.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Whi92] J. A. Whittaker. *Software Testing and Reliability Analysis.* PhD thesis, The university of Tennessee, Knoxville, May 1992.

[WM10] John Watkins and Simon Mills. *Testing IT : An Off-the-Shelf Software Testing Process.* Cambridge University Press, New York, NY, USA, 2nd edition, 2010.

[ZHJ04] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In *Software Product-Family Engineering*, pages 129–139. Springer, 2004.

[ZZM13] Jian Zhang, Zhiqiang Zhang, and Feifei Ma. Introduction to combinatorial testing. In *Automatic Generation of Combinatorial Test Data*, pages 1–16. Springer, 1st edition, 2013.

Table des figures

Résumé

L'ingénierie des lignes de produits est une approche utilisée pour développer une famille de produits. Ces produits partagent un ensemble de points communs et un ensemble de points de variation. Aujourd'hui, la validation est une activité disjointe du processus de développement des lignes de produits. L'effort et les moyens fournis dans les campagnes de tests de chaque produit peuvent être optimisés dans un contexte plus global au niveau de la ligne de produits. Le model-based testing est une technique de génération automatique des cas de test à partir d'un modèle d'états et de transitions construit à partir des exigences fonctionnelles. Dans cette thèse, nous présentons une approche pour tester une ligne de produits logiciels avec le model-based testing. La première contribution consiste à établir un lien entre le modèle de variabilité et le modèle de test, à l'aide des exigences fonctionnelles. La deuxième contribution est un algorithme qui extrait automatiquement un modèle de test spécifique à un produit membre de la famille de produits sous test. L'approche est illustrée par une famille de produits de tableaux de bord d'automobiles et expérimentée par un industriel du domaine aéronautique dans le cadre du projet Européen MBAT.

Abstract

Software product line engineering is an approach that supports developing products in family. These products are described by common and variable features. Currently, the validation activity is disjointed from the product lines development process. The effort and resources provided in the test campaigns for each product can be optimized in the context of product lines. Model-based testing is a technique for automatically generating a suite of test cases from requirements. In this thesis report, we present an approach to test a software product line with model-based testing. This technique is based on an algorithm that establishes the relationship between the variability model released with OVM and the test model, using traceability of functional requirements present in both formalisms. Our contribution is an algorithm that automatically extracts a product test model. It is illustrated with a real industrial case of automotive dashboards and experimented by an industrial of aeronautic domain in the MBAT European project context.