UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

ÉCOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS ED 211

THÈSE

pour l'obtention du

Doctorat de l'Université de Pau et des Pays de l'Adour

(spécialité informatique)

présentée par

Youssef RIDENE

Ingénierie dirigée par les modèles pour la gestion de la variabilité dans le test d'applications mobiles

soutenue publiquement le 23 septembre 2011

Composition du jury

Président :	Xavier BLANC	Professeur à l'université de Bordeaux 1
Rapporteurs :	Benoit BAUDRY	Chargé de recherche (HDR) à l'INRIA de Rennes
	Fabrice BOUQUET	Professeur à l'université de Besançon
Examinateurs :	Franck BARBIER	Professeur à l'université de Pau
	Nadine COUTURE	Enseignant-chercheur (HDR) à l'ESTIA
Invité :	Jean-Gilles HOURSIANGOU	Directeur de la société Neomades

Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour — EA 3000

LIUPPA

Mis en page avec la classe thloria.

Remerciements

Ce mémoire de thèse représente l'achèvement de trois années d'un long travail qui n'aurait pu voir le jour sans la participation, l'aide, les conseils, ou encore la présence de nombreuses personnes.

Mes remerciements vont tout d'abord à M. Benoit Baudry et M. Fabrice Bouquet pour avoir accepté de relire mon manuscrit et d'en être les rapporteurs. Je tiens aussi à remercier également M. Xavier Blanc, de m'avoir honoré par sa présence et d'avoir accepté de présider le jury de ma soutenance de thèse.

Je tiens aussi à remercier Mme Nadine Couture et M. Nicolas Belloir pour leur écoute tout au long de mes travaux et pour les échanges, qui ont su à chaque fois m'éclairer dans mon travail et pas que. Mes prochains remerciements s'adressent à M. Franck Barbier qui a dirigé mes travaux et a su donner les directions de recherche nécessaires. Un immense merci pour m'avoir encadré avec autant de sérieux, de rigueur, mais aussi de gentillesse et d'humilité. Merci de m'avoir toujours fait confiance, sans jamais m'avoir sous-estimé et de m'avoir toujours encouragé et même soutenu. Merci pour la relecture minutieuse du mémoire. Je n'oublierai pas non plus les longues discussions, parfois même personnelles pendant les repas de midi et les pauses café.

Je tiens à témoigner ma sincère et profonde gratitude à M. Jean-Gilles Hoursiangou pour m'avoir permis d'effectuer cette thèse au sein de la société Neomades et pour la confiance qu'il m'accordé depuis quelques années. Je remercie également tous les membres de l'équipe Neomades (Aurélien, Damien, Julien, Marie-Dominique, Maxime, Nicolas Rafal, Roch et Sébastien) pour la bonne ambiance qu'ils instaurent et pour les différents échanges intéressants. Un merci tout particulier à Julien sur qui on peut compter tout le temps et à Marie-Dominique pour ses relectures. Je tiens également à remercier Mickaël pour son sérieux, son efficiacité mais aussi pour son aide.

Je souhaite adresser un remerciement à tous les membres du laboratoire et du département informatique de l'université de Pau et des Pays de l'Adour pour leur accueil, leur gentillesse, disponibilité et pour toutes les discussions qu'on a eu. Un remerciement particulier à Annig, Cong Duc, Eric, Laurent, Régine et Sophie... sans oublier les anciens, actuels et nouveaux doctorants avec qui j'ai partagé les galères et les jois des thésards et qui en général finissent par devenir des amis. Allez, je me lance : Cyril, Damien, Ehsan, Eric, Gaetan et Nour.

Une section à part pour adresser un immense merci à deux personnes que je considère de ma famille, je les remercie pour leur ouverture d'esprit, pour leur gentillesse, pour leur sens de l'aide et du partage, pour leur générosité et pour leur disponibilité. Merci à Henriette et Claude pour tout ce qu'ils ont fait pour moi.

C'est un énorme remerciement que j'adresse maintenant à ma famille. Vous avez su à votre manière, par vos paroles et vos gestes, m'encourager et m'accompagner non seulement durant cette thèse mais depuis mon premier jour à l'école. Un gros merci à mes parents Rafik et Hassiba qui ont toujours cru en moi et qui n'ont pas hésité une seule seconde à foncer et à me soutenir sur tous les plans dans mes projets universitaires. Merci à eux pour tout ce qu'ils ont fait pour moi, pour leurs conseils, pour leur soutien et pour tous les sacrifices qu'ils ont fait pour que je ne manque de rien. Je suis fier d'être votre fils. Un immense merci à mon frère Mohamed Salah qui m'a toujours encouragé et cru en moi. Je te souhaite plein de belles choses avec Malou ! J'adresse un gros merci à ma sœur Hajer pour tous les moments inoubliables passés ensemble. Je ne pourrais ne pas remercier mon tonton Khaled pour son soutien et pour avoir cru en moi dés le départ de cette aventure. Je tiens aussi à remercier ma seconde famille qui m'a adaopté. Un immense particulier à Chhouba pour ses conseils avérés.

Je remercie en dernier ma chère et tendre épouse Yosra. Un énorme merci pour ton amour et ton soutien sans faille grâce auxquels j'ai pu aboutir ce projet. Merci pour ta patience, merci pour les soirées et les weekend que j'ai passé derrière mon ordianteur, merci pour tout ce que tu fais pour moi. Enfin je ne peux ne pas ne pas te remercier pour l'adorable petit ange, notre chère Yakine. Je te dois beaucoup.

Cet exercice de style s'avère plus ardu que je ne l'aurais pensé... Je souhaiterais n'omettre personne mais cela me semble difficile. Aussi, je remercie par anticipation tous ceux qui liront ces lignes en ayant le sentiment d'avoir été oublié dans la liste précédente.

Je dédie cette thèse à ma famille.

Table des matières

Table des figures 1

Liste des tableaux 3

Partie I Introduction générale 5

1 Contexte du travail . . . 7
2 Problématique . . . 8
3 Objectifs de la thèse . . . 11
4 Organisation du mémoire . . . 12

Partie II Contexte technologique et scientifique 15

Chapitre 1
Le développement d'applications mobiles

1.1 La fragmentation . . . 18
1.1.1 La fragmentation Java ME . . . 18
1.1.2 La fragmentation des plateformes mobiles . . . 21
1.2 Le test des applications mobiles . . . 23
1.2.1 Le test de logiciel . . . 23
1.2.2 Simulateurs ou téléphones réels . . . 25
1.2.3 Procédure de test . . . 26
1.2.4 Le test manuel . . . 28
1.2.5 Le *cloud testing* . . . 29
1.3 Synthèse et objectifs . . . 31

Chapitre 2
L'ingénierie dirigée par les modèles - IDM

2.1 Introduction . . . 34
2.1.1 L'exemple *MDA* . . . 34
2.1.2 Définitions . . . 35
2.2 Langages de modélisation . . . 35
2.2.1 Définition d'un profil UML . . . 36
2.2.2 Définition d'un DSML . . . 39
2.2.3 Bilan . . . 43
2.3 Transformation de modèles . . . 43
2.4 Le test et les langages de modélisation . . . 44
2.4.1 Langage de modélisation vs. langage spécifique de test . . . 45
2.4.2 *Model-Based Testing* vs. *Model-Driven Testing* . . . 47
2.5 Synthèse . . . 49

Chapitre 3
Les lignes de produits logiciels

3.1 Introduction . . . 52
3.2 Définitions . . . 52
3.3 L'ingénierie de domaine et d'application . . . 53
3.3.1 Ingénierie de domaine . . . 53
3.3.2 Ingénierie d'application . . . 54
3.4 Les contraintes de dépendance . . . 54
3.5 Expression de la variabilité . . . 55
3.5.1 Au niveau du code . . . 55
3.5.2 Au niveau des modèles . . . 57
3.6 Mise en œuvre dans l'industrie . . . 58
3.6.1 L'exemple Renault . . . 58
3.6.2 L'exemple Nokia . . . 59
3.7 Le test dans une ligne de produits logiciels . . . 59
3.8 Synthèse . . . 61

Partie III MATeL : un DSML pour la définition de scénarios de test d'applications mobiles **63**

Chapitre 4
MATeL

4.1 Introduction . . . 66
4.1.1 MATeL : un langage spécifique . . . 66
4.1.2 Objectifs de MATeL . . . 67
4.2 Le métamodèle MATeL . . . 67
4.2.1 Noyau du métamodèle . . . 68
4.2.2 Gestion des interruptions . . . 76
4.2.3 Gestion des résultats . . . 80
4.2.4 Gestion de la variabilité . . . 81
4.3 Conception de l'éditeur MATeL . . . 87
4.4 Génération automatique de scénarios MATeL . . . 88
4.5 Synthèse . . . 93

Chapitre 5
Architecture logicielle et matérielle de MATeL

5.1 Introduction . . . 96
5.2 Architecture du banc de test . . . 96
5.3 Editeur MATeL . . . 100
5.3.1 Interface avec le banc de test . . . 100
5.3.2 Métatypes et icônes . . . 103
5.4 Synthèse . . . 106

Chapitre 6
Evaluation

6.1 Introduction . . . 108
6.2 Etude de cas . . . 108
6.2.1 Description de l'application . . . 108
6.2.2 Description des mobiles ciblés . . . 109
6.2.3 Description du scénario MATeL . . . 110

6.2.4 Gestion des résultats de test 114
6.3 Apports et limitations de MATeL 116
6.3.1 Apports 116
6.3.2 Limitations 117
6.4 Synthèse 118

Partie IV Conclusion et perspectives **121**

Chapitre 7
Conclusions et perspectives

7.1 Bilan 124
7.2 Perspectives 125
7.2.1 Benchmark 125
7.2.2 IDM et Eclipse 125
7.2.3 Métamodèle MATeL 125

Partie V Annexes **127**

Chapitre 8
Annexes

8.1 Plan de test 130
8.2 Metamodèle complet 134

Bibliographie **137**

Table des figures

1 Aperçu des différents « stores » et des dates de leur lancement 10
2 Aperçu du nombre d'applications téléchargées (en millions) 11

1.1 Différentes API du MSA 19
1.2 Architecture globale d'une plateforme pour le test à distance 29
1.3 Exemple de trois claviers différents 30

2.1 Principes de l'approche MDA 35
2.2 L'architecture de métamodélisation en couches [UML09] 36
2.3 Exemple d'un profil UML pour les EJB [UML09] 37
2.4 Concepts de base pour la métamodélisation (EMF/Ecore) 40
2.5 Approches de mise en œuvre d'un DSML 41
2.6 Exemple d'un DSML pour la modélisation de sites Web 42
2.7 Processus de transformation de modèles [CH06] 44
2.8 Les différentes étapes dans un processus Model-Based Testing 48

3.1 L'ingénierie de lignes de produits logiciels 54
3.2 Exemple d'un *Feature Model* pour la fabrication d'un téléphone mobile [LH10] 55
3.3 Exemple de mise en œuvre de l'héritage 56
3.4 Diagramme de classe d'une LdP basée sur l'utilisation d'un profil UML [TZJ03] 57
3.5 Relation entre produits et tests [McG07] 60

4.1 Principaux concepts du métamodèle MATeL 70
4.2 Extrait du métamodèle pour la gestion des entrées clavier 73
4.3 Extrait du métamodèle pour la gestion des entrées écran 74
4.4 Extrait du métamodèle pour la configuration du mobile 75
4.5 Extrait du métamodèle pour le nettoyage du téléphone 77
4.6 Métatypes pour la gestion des interruptions 78
4.7 Extrait du métamodèle pour la gestion des interruptions utilisateur 79
4.8 Métatypes pour la gestion des résultats 80
4.9 Dimensions de la variabilité 83
4.10 Métatypes pour la gestion de la variabilité 84

4.11 Exemple de modélisation de variabilité 85
4.12 Processus de génération d'un modeleur avec GMF 88
4.13 Vue globale d'une application mobile 89
4.14 Métamodèle MobiAM 91

5.1 Architecture globale d'une plateforme pour le test à distance 96
5.2 Câblage de quelques téléphones 97
5.3 Banc de test développé par Neomades 98
5.4 Architecture globale du banc de test 99
5.5 Editeur MATeL 101
5.6 Du modèle au test effectif 102
5.7 Paramètres du métatypes *WriteText* 103

6.1 Ecrans de l'application à tester 109
6.2 Exemple de scénario de test 111
6.3 Vue Eclipse pour la gestion des résultats 115

8.1 Partie 1 du métamodèle MATeL 134
8.2 Partie 2 du métamodèle MATeL 135

Liste des tableaux

1.1 Aperçu des différentes plateformes (systèmes d'exploitation) mobiles . . . 22

5.1 Détails de la commande Modbus 98
5.2 Différents métatypes utilisables dans l'éditeur MATeL 104

6.1 comparaison du Samsung D800 et du Sony Ericsson P990i 110

Première partie

Introduction générale

Ce travail de thèse a été effectué dans le cadre d'une convention CIFRE entre le Laboratoire d'informatique de l'université de Pau et des Pays de l'Adour (LIUPPA), l'Ecole supérieure des technologies industrielles avancées (ESTIA) et la société Neomades. Cette thèse a été dirigée par M. Franck Barbier en collaboration avec Mme Nadine Couture et M. Nicolas Belloir.

Cette thèse traite de la problématique relative aux tests d'applications embarquées sur téléphones mobiles. Cette première partie introduit le contexte scientifique et technologique, les enjeux et les objectifs de cette thèse.

1 Contexte du travail

L'utilisation des systèmes informatiques prend de plus en plus de place dans notre vie quotidienne. Aujourd'hui, nos voitures, nos téléviseurs, nos micro-ondes, nos cartes bancaires, nos tablettes Internet... fonctionnent avec des programmes informatiques communicants plus ou moins complexes. Dans ce contexte d'informatique ubiquitaire [Wei93], ces programmes sont amenés à interagir avec l'utilisateur et avec le monde extérieur. Ces interactions peuvent parfois être critiques. Par exemple, un conducteur doit à tout moment être capable de compter sur le système de freinage d'urgence (ABS [1]) de sa voiture. Cela implique que le logiciel embarqué dans ce système doit fonctionner correctement à tout moment. Plusieurs accidents ont eu lieu à cause de dysfonctionnements logiciels et ont causé des pertes humaines et matérielles considérables. On peut citer à titre d'exemple l'explosion en vol de la fusée Ariane 5 dont les analyses ont démontré que la cause de l'accident était bien liée à un bug logiciel [2] [JM97]. L'omniprésence de l'informatique dans notre vie oblige les informaticiens à s'assurer de la fiabilité de tout programme gérant un dispositif quelconque. Cette fiabilité est assurée d'une part par un processus de spécification et de développement très rigoureux, d'autre part par une phase de test logiciel dont le but est de détecter les erreurs de conception et de réalisation des programmes pour tendre vers des systèmes avec zéro défaut. Toutefois, la majorité des dysfonctionnements logiciels est dénuée de risque pour l'utilisateur. Avec la banalisation de l'informatique, l'utilisateur, *i.e.* le client final, est devenu de plus en plus exigeant quant à la qualité des logiciels qu'il utilise.

Le domaine de la téléphonie mobile est le domaine d'intérêt de cette thèse. Les avancées technologiques dans le domaine de l'électronique et notamment dans la miniaturisation des composants (mémoire, processeur, capteurs...) ont permis de concevoir des appareils de plus en plus petits, dotés de plus en plus de fonctionnalités. Durant ces dernières années, nous avons assisté à une évolution fulgurante des mobiles : ils sont devenus des mini-ordinateurs aux spécificités matérielles et logicielles évoluées (écran multi-touches sans stylet, détecteur de position, GPS, reconnaissance vocale, clavier virtuel, grande capacité de stockage et de calcul, systèmes d'exploitation multitâches, interfaces homme-machine (IHM) plus évoluées et plus ergonomiques...). L'évolution des

1. Anti-lock Braking System
2. http ://www.irisa.fr/pampa/EPEE/Ariane5.html

réseaux de télécommunication mobiles - qui en sont aujourd'hui à la troisième génération (3G) et très bientôt à la quatrième génération (4G) - a aussi permis de transformer les téléphones mobiles en systèmes connectés en permanence.

Le rôle de cette nouvelle génération de téléphones intelligents (*smartphones*) ne se limite plus à téléphoner ou envoyer des messages courts (SMS [3]) mais s'étend aussi à prendre des photos et les partager en temps réel grâce aux réseaux sociaux, à regarder la télévision, à lire et rédiger des mails, à naviguer sur Internet, à écouter la radio, à jouer... Ces *smartphones* offrent aussi la possibilité de télécharger et installer des applications souvent produites par des développeurs tiers. Le succès ou l'échec de ces applications est lié à l'originalité et à l'utilité de celles-ci mais aussi à leur qualité. Du point de vue du client, cette qualité correspond principalement au fait que l'application s'installe « proprement » sur le téléphone, démarre sans générer d'erreurs, ne fait que ce pour quoi elle a été prévue, ne se bloque pas, ne s'arrête pas brutalement... Le développeur doit aussi garantir à l'utilisateur la robustesse de l'application en termes de sécurité et de respect de la confidentialité des données personnnelles, notamment du fait de la prolifération de nouvelles applications avancées liées au commerce mobile (m-commerce) ou encore au paiement mobile (m-payment).

Le nombre important d'applications mobiles disponibles, qui ne cesse de croître, a rendu les utilisateurs de plus en plus exigeants quant à la qualité des applications. Seule une procédure de test efficace permet de répondre à ces exigences. Cette phase incontournable doit être effectuée avant de mettre l'application à la disposition du client final. Dans le contexte d'applications embarquées sur téléphones mobiles, le test est un vrai défi, principalement à cause du nombre important de terminaux mobiles (plateformes cibles). Une application qui fonctionne dans un contexte précis et défini (en l'occurence un téléphone de marque X, modèle Y connecté à l'opérateur de télécommunications Z dans le pays T), fonctionne-t-elle dans un autre contexte ?

2 Problématique

La simplicité et le confort d'utilisation des terminaux, conjugués à la généralisation des réseaux 3G et à des formules tarifaires illimitées pour l'accès à l'Internet dopent la consommation de contenus mobiles tels qu'applications et jeux embarqués. La société Apple [4] a bousculé les habitudes des usagers avec l'iPhone [Bou09] lancé en association avec une vraie révolution, « l'App Store » [5]. Comme son nom l'indique, il s'agit d'une boutique d'applications accessible directement à partir du *smartphone*. Ce concept a rencontré un succès sans précédent avec des millions d'applications téléchargées. Bien avant Apple, quelques entreprises avaient eu l'idée de lancer des « stores » : des sites comme

3. Short Messaging Service : par abus de langage, le nom du service est aussi utilisé pour désigner le message envoyé.
4. http ://www.apple.com
5. http ://www.apple.com/iphone/apps-for-iphone

Handango [6] ou Getjar [7] proposent aux développeurs de mettre en ligne leurs applications mobiles. Pour un utilisateur lambda, la démarche pour télécharger et installer un logiciel sur son mobile n'est pas toujours évidente. Elle peut rapidement devenir un vrai casse-tête alors qu'avec l'iPhone, tout se fait directement avec le téléphone en lançant une application. Suite à ce fulgurant succès, tous les géants du marché se sont empressés de créer leurs propres « stores » [tS10]. On peut citer Windows MarketPlace [8], Nokia Ovi Store [9], Google Android Market [10], Palm App Catalog [11], Sony Ericsson PlayNow [12], Orange Application Shop [13], SFR Appli Store [14] (cf. figure 1).

Grâce à ces magasins d'applications, le marché des applications mobiles connaît un engouement très important de la part du grand public (cf. figure 2). Cet engouement a encouragé les développeurs à produire de plus en plus d'applications pour toutes les plateformes mobiles : ceci s'avère une tâche difficile et coûteuse compte tenu du nombre important de modèles de téléphones différents. Leurs différences se vérifient à tous les niveaux : systèmes d'exploitation, formats audio/vidéo supportés, types de clavier (Azerty, Qwerty, numérique, virtuel...), protocoles de communication supportés (HTTP, HTTPS, RTSP...) ou encore tailles d'écran. Pour atteindre un maximum d'utilisateurs et garantir le succès d'une application mobile, le développeur est obligé de produire une application qui fonctionne sur plusieurs mobiles, donc qui prend en compte les spécificités de chaque plateforme. Cette problématique est connue dans le monde industriel sous le nom de *fragmentation mobile* [GE11].

Ces différences entre modèles de téléphones mobiles offrent aux utilisateurs un large éventail de choix mais posent un problème majeur aux développeurs. En effet, quasiment tous les constructeurs de mobile offrent aux développeurs la possibilité de produire leurs propres applications et ceci en utilisant la ou les technologie(s) supportée(s) par leurs appareils. Pour l'iPhone, le développeur doit développer l'application en Objective-C sous l'environnement XCode [15], pour Android en Java en utilisant un SDK spécifique [16], pour d'autres téléphones tels que Sony Ericsson, Nokia... en Java ME [17] et ses multiples versions. Le passage d'une plateforme à une autre n'est pas sans risque et peut altérer le bon fonctionnement de l'application. En effet, le développeur doit d'une part adapter son code source, voire tout réécrire, et d'autre part, fournir les ressources adéquates en fonction du téléphone (images selon la taille de l'écran, formats des fichiers audio et vidéo selon les capacités du mobile...). Ajoutons à cela qu'il faut parfois contourner

6. http ://www.handango.com
7. http ://www.getjar.com
8. http ://www.windowsmarketplace.com
9. http ://store.ovi.com
10. https ://market.android.com
11. http ://www.palm.com/fr/fr/products/software/mobile-applications.html
12. http ://www.playnow-arena.com
13. http ://www.orangepartner.com
14. http ://atelier.sfr.fr
15. http ://developer.apple.com/tools/xcode/
16. http ://developer.android.com/sdk
17. http ://www.oracle.com/technetwork/java/javame

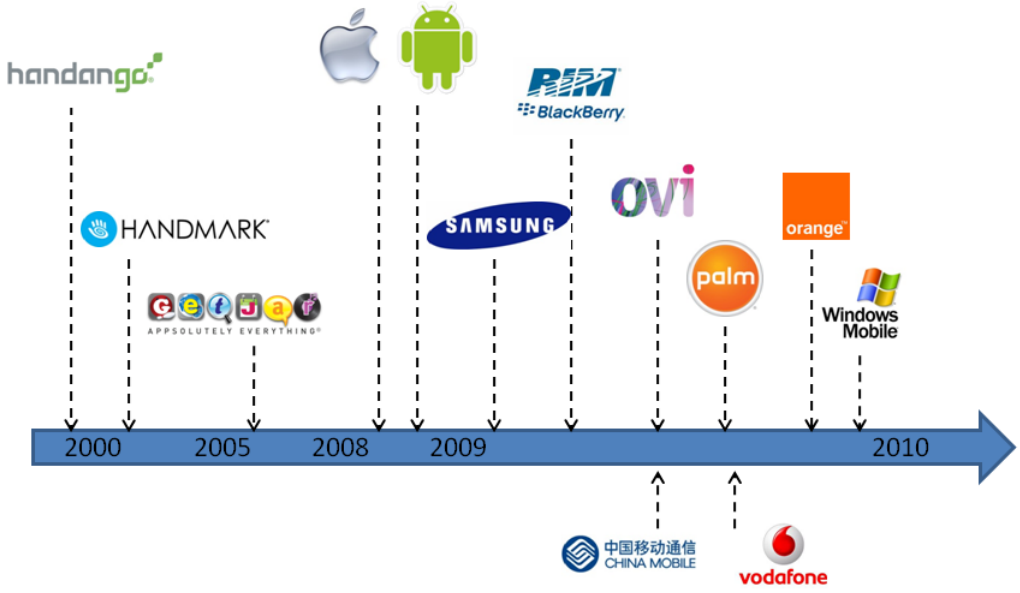


FIGURE 1 – Aperçu des différents « stores » et des dates de leur lancement

des bugs d'implémentation des *firmwares* ou des systèmes d'exploitation des téléphones eux-mêmes.

Durant le développement d'une application mobile, des tests doivent fréquemment être réalisés pour valider les différentes composantes de l'application (fonctionnelle, affichage, interaction avec l'utilisateur...). La solution la plus facile serait d'utiliser des simulateurs sur le poste de développement tel que WTK [18]. Cette approche est intéressante pour valider certains points mais les simulateurs génèrent un grand nombre de problèmes. Parmi les plus importants, nous soulignons que :

- ces simulateurs ne couvrent pas tous les téléphones ;
- le nombre de simulateurs à installer est considérable (générique, Nokia, Sony Ericsson, Android, BlackBerry, iPhone...) ;
- très peu de simulateurs fonctionnent sous un système d'exploitation autre que Windows ;

18. Wireless ToolKit : http ://www.oracle.com/technetwork/java/index-jsp-137162.html

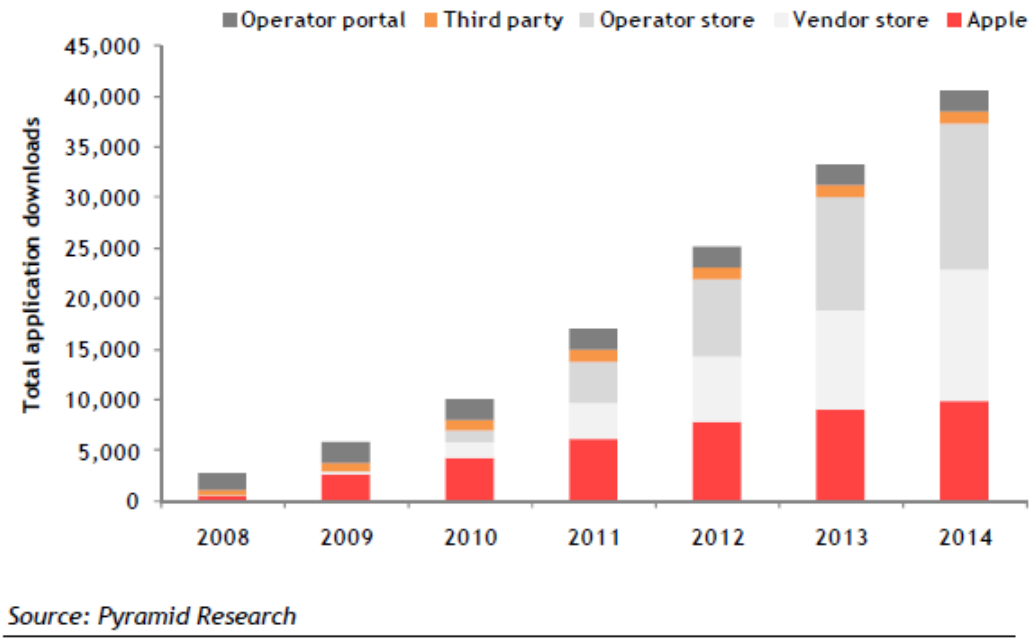


FIGURE 2 – Aperçu du nombre d'applications téléchargées (en millions)

- les simulateurs ont les capacités mémoires et CPU du PC sur lequel ils tournent et non pas celles du vrai téléphone ;
- même si l'interface du simulateur peut parfois être changée selon le mobile, la simulation, quant à elle, reste générique et ne prend pas en compte les caractéristiques techniques et le comportement réels du téléphone.

Face à la problématique de simulation des applications mobiles et afin de s'assurer du bon fonctionnement de son application sur l'ensemble des téléphones ciblés, le développeur doit la tester sur chaque mobile. Ce test devient rapidement un parcours du combattant, vu le nombre important de téléphones à acquérir, la difficulté de mise en place et de gestion d'une équipe de testeurs dédiée (organisation, choix et configuration d'un outil de suivi de bugs, mise en place d'une procédure, etc). Ajoutons à cela le fait que les tests sont répétitifs et sans valeur ajoutée ni pour les testeurs ni pour l'entreprise notamment à cause de la non capitalisation des tests et l'absence de réutilisation.

3 Objectifs de la thèse

Afin de répondre à la problématique de la fragmentation mobile, plusieurs solutions ont vu le jour et permettent de garantir à un développeur d'applications mobiles l'indépendance et la portabilité de son application sur la majorité des téléphones du marché,

tout en ne développant qu'un seul code source par projet mobile. Parmi ces produits, on peut citer NeoMAD [19], solution fournie par l'entreprise Neomades. Ce type d'outil facilite la phase de développement mais ne dispense en aucun cas le développeur d'effectuer le test final sur les téléphones cibles.

Le travail de cette thèse a pour objectif d'apporter une solution à la problématique du test : répétitivité, absence de valeur ajoutée, absence de partage de savoir-faire... Nous souhaitons proposer une méthode outillée permettant la description de scénarios de test en prenant en compte les variabilités entre les différents téléphones et en permettant leur exécution automatique sur différents mobiles. Notre travail est basé sur une approche dirigée par les modèles. Ce choix est justifié par les apports de l'Ingénirie Dirigée par les Modèles - IDM (cf. chapitre 2). Nous proposons la définition d'un langage de modélisation spécifique au domaine du test d'applications mobiles ou *Domain-Specific Modeling Language* (DSML) en anglais. Ce langage dédié permet au concepteur du test de spécifier, dans son scénario, les points communs et les points de variabilité (dans l'esprit de l'ingénierie des lignes de produits logiciels ou *Software Product-Lines* (SPL) en anglais) qui feront que la procédure de test sera différente d'un téléphone à un autre. Ce langage doit à la fois offrir à l'utilisateur une expressivité riche pour modéliser tout type de test et être facile à utiliser pour des utilisateurs qui ne sont pas forcément des informaticiens. Les trois champs scientifiques dans lesquels cette thèse s'inscrit sont ainsi :

- le test « dirigé par les modèles » pour formaliser les tests et les exécuter ;
- l'ingénierie dirigée par les modèles pour définir des DSMLs ;
- l'ingénierie des lignes de produit pour traiter la problématique de variabilité.

4 Organisation du mémoire

Ce document est organisé en six chapitres. Dans les trois premiers chapitres, nous présentons le contexte scientifique et technologique dans lesquel nous inscrivons nos travaux. Dans le premier chapitre, nous présentons le monde du développement des applications mobiles. Nous y exposons sa spécificité et nous présentons plus en détails la problématique de la fragmentation mobile. Ensuite, nous donnons une vue générale sur le test logiciel afin de mettre en évidence le type de test que l'on souhaite cibler. Le deuxième chapitre concerne l'ingénierie dirigée par les modèles et plus particulièrement la métamodélisation et les langages spécifiques de domaines (DSML). Dans le troisième chapitre, nous présentons le domaine des lignes de produits logiciels ou *Software Product-Lines* (SPL) en anglais et notamment la gestion de la variabilité logicielle. Ces deux approches, *i.e.* DSML et SPL, constituent le cœur de la méthodologie dans notre démarche visant à répondre à la problématique des tests. Nous présentons par la suite, dans les deux chapitres suivants, notre contribution à la conception de scénarios de test, avec la possibilité d'exprimer des points de variabilité dépendant des caractéristiques des mobiles cibles.

19. http ://www.neomades.com

Dans le quatrième chapitre, nous détaillons le langage développé et ses différentes composantes. Ensuite, dans le cinquième chapitre, nous présentons la plateforme d'exécution que nous avons construite et nous décrivons la procédure de conception et d'exécution d'un scénario d'une application mobile. Quant au sixième et dernier chapitre, nous l'utilisons pour évaluer notre contribution en exposant les apports et les limites de celle-ci.

Au terme de ce document, nous tirons des conclusions d'ordre général et proposons un ensemble de perspectives à ces travaux.

Deuxième partie

Contexte technologique et scientifique

Chapitre 1

Le développement d'applications mobiles

Sommaire

1.1 La fragmentation 18
1.1.1 La fragmentation Java ME 18
1.1.2 La fragmentation des plateformes mobiles 21
1.2 Le test des applications mobiles 23
1.2.1 Le test de logiciel 23
1.2.2 Simulateurs ou téléphones réels 25
1.2.3 Procédure de test 26
1.2.4 Le test manuel 28
1.2.5 Le *cloud testing* 29
1.3 Synthèse et objectifs 31

Fragmentation is the inability to « write once and run anywhere ». Fragmentation is the evil twin of differentiation, a term marketing managers use to explain the need to create so many different handsets[20].

1.1 La fragmentation

Quelques années auparavant, le monde du mobile profitait principalement aux fabricants de téléphones portables et aux opérateurs de télécommunication grâce à la vente de leurs appareils et à la facturation des communications. Aujourd'hui, l'écosystème a évolué pour intégrer un nombre plus important d'acteurs, notamment les développeurs d'applications [HO09]. En effet, le marché des applications mobiles connaît un succès sans précédent et ceci grâce à la prolifération de nouvelles plateformes mobiles telles que l'iPhone et Android.

Nous étudions dans ce chapitre les spécificités du développement d'un projet mobile, notamment le syndrome de la fragmentation mobile et l'impact que cela a sur la phase de test. Nous proposons pour cela de dinstinguer deux types de fragmentation : la première liée à la technologie Java ME [Whi01] et la seconde aux plateformes mobiles [GE11].

1.1.1 La fragmentation Java ME

A la fin des années 90, à l'initiative de la société Sun et de quelques acteurs majeurs du domaine de la téléphonie mobile (fabricants de terminaux, opérateurs, éditeurs de logiciels, etc.), la technologie Java ME a été lancée. Il s'agit d'une spécification de machine virtuelle Java adaptée aux terminaux mobiles. Cette JVM [21] regroupe un ensemble d'APIs qui offre aux développeurs la possibilité d'implémenter des applications embarquées sur téléphones mobiles plus ou moins complexes en utilisant par exemple les interfaces graphiques, l'audio, la vidéo, le Bluetooth ou encore la 3D [Law02]. Une architecture appelée MSA [22] a été définie pour spécifier les différentes APIs de la technologie (cf. figure 1.1). Notons que certains fabricants ont aussi ajouté leurs propres APIs optionnelles.

Avant l'arrivée des *smartphones* de nouvelle génération, la quasi-totalité des téléphones portables embarquait des JVM plus ou moins conformes aux spécifications de la technologie Java ME. En effet, les spécifications Java ME n'imposent pas aux fabricants d'implémenter toutes ces APIs surtout celles qui nécessitent des dispositifs matériels supplémentaires comme une caméra, un capteur Bluetooth ou encore une carte graphique 3D. Ainsi, les fabricants ont une certaine liberté quant à l'implémentation de la JVM qu'ils embarquent sachant que CLDC [23] et MIDP [24] (deux premières couches de MSA,

20. http ://www.comp.nus.edu.sg/ damithch/df/device-fragmentation.htm
21. Java Virtual Machine
22. Mobile Service Architecture
23. **Connected Limited Device Configuration :** ensemble des classes minimales du langage Java nécessaires pour le développeur (java.lang.*, java.util.*), les E/S, la sécurité...
24. **Mobile Information Device Profil :** il s'agit de l'ensemble des APIs qui définissent l'interaction

MSA:	
JSR 238 (Internationalization)	
JSR 234 (Multimedia Supplements)	
JSR 229 (Payment)	
JSR 211 (Content Handler)	
JSR 180 (SIP)	
JSR 179 (Location)	
JSR 177 (Security & Trust)	
JSR 172 (Web Serviecs)	**MSA Subset:**
JSR 226 (Vector Graphics)	JSR 226 (Vector Graphics)
JSR 205 (Messaging)	JSR 205 (Messaging)
JSR 184 (3D Graphics)	JSR 184 (3D Graphics)
JSR 135 (Mobile Media)	JSR 135 (Mobile Media)
JSR 82 (Bluetooth)	JSR 82 (Bluetooth)
JSR 75 (File & PIM)	JSR 75 (File & PIM)
JSR 118 (MIDP)	JSR 118 (MIDP)
JSR 139 (CLDC)	JSR 139 (CLDC)

FIGURE 1.1 – Différentes API du MSA

cf. figure 1.1) sont obligatoires, vu qu'ils représentent le cœur de la technologie Java ME. La diversité des associations JVM/fabricants implique des implémentations différentes de ces JVM d'un modèle à un autre même entre les téléphones du même fabricant. Nous nous retrouvons ainsi avec des modèles de téléphones qui se différencient à deux niveaux :

- **matériel** - avec des écrans de résolutions différentes, des claviers différents, des capteurs différents ou encore des capacités mémoires et CPU différentes ;
- **logiciel** - avec des machines virtuelles Java différentes en fonction de l'implémentation choisie par le constructeur mais aussi en fonction des interprétations que chaque constructeur aurait fait des spécifications ; sans oublier les bugs introduits dans ces implémentations [KMH07].

Ces différents modèles de mobiles offrent un choix intéressant à l'utilisateur mais posent un problème majeur aux développeurs d'applications mobiles, connu sous le nom de « fragmentation mobile » [BBG+08]. Plusieurs travaux académiques et industriels ont été menés afin de proposer des solutions à cette problématique [ACV+05] et [Raj08] ; ces solutions sont souvent appelées « outils de portage ». Ces travaux plus ou moins aboutis suivent différentes approches. Certaines comme [Whi05], [Dis08] ou encore [Vir05] se

de l'utilisateur avec le téléphone lui-même en passant par la JVM (écran, son, vibreur, mémoire...).

basent sur l'utilisation des caractéristiques des mobiles (taille d'écran, mémoire, bugs connus, formats audio et vidéo supportés, protocoles de communication supportés, etc.) pour conditionner le code source de l'application selon les caractéristiques des mobiles ciblés (cf. listing 1.1) . Ces informations sont souvent stockées dans des bases de données telles que celle de J2MEPolish [25].

Listing 1.1 – Exemple de code conditionné selon la taille de l'écran du mobile

```
1   Image background_img = null;
2 //#if screen_height == 320
3   background_img = Image.createImage(bg_320.png);
4 //#else
5   background_img = Image.createImage(bg.png);
6 //#endif
```

D'autres solutions comme [GCH+05], [twu10] ou encore [Sar09] proposent des APIs qui permettent de faire abstraction de certaines caractéristiques du mobile, principalement la taille de l'écran et le type du clavier, et ceci en adaptant l'application dynamiquement au moment de son exécution. Ces APIs sont souvent une implémentation personnalisée de tous les composants graphiques de Java ME (bouton, zone de texte, liste...). Le principal inconvénient de ces APIs réside dans le fait qu'elles n'apportent qu'une solution partielle à la fragmentation : elles ne concernent que les problèmes de fragmentaiton liés aux interfaces graphiques (couche MIDP de MSA, cf. figure 1.1) alors que les autres APIs telles que « Mobile Media » ou encore « Location » posent autant de problèmes que l'IHM voir plus.

Le développeur se trouve ainsi obligé d'adapter son application afin de pouvoir supporter tous les téléphones sur lesquels son application devra fonctionner. Ces adaptations sont principalement positionnées sur trois niveaux :

- **code source** - le développeur est souvent contraint de prévoir des traitements différents en fonction des spécificités de chaque téléphone. Ceci est utile par exemple pour le contournement de bugs d'implémentation connus de la JVM ;
- **fonctionnalités** - il arrive parfois qu'un ou plusieurs des téléphones ciblés ne supportent pas une fonctionnalité spécifique. Dans ce cas, le développeur n'a pas d'autre choix que de la supprimer ou la remplacer pour ces téléphones là ;
- **ressources** - afin de pouvoir supporter toutes les résolutions d'écran, il est nécessaire d'adapter les graphismes en fonction de ces résolutions. De la même manière, les fichiers audio et vidéo doivent avoir un format supporté par le mobile ciblé.

Souvent, dans les projets Java ME, le terme « portage » est utilisé pour désigner cette phase d'adaptation. Cette phase peut être réalisée au même temps que le développement dans le cas d'utilisation d'un outil de portage ou après le développement si le portage est réalisé manuellement.

25. http ://devices.j2mepolish.org

1.1.2 La fragmentation des plateformes mobiles

Le développement d'une application mobile est un projet qui peut être géré comme tout projet informatique classique à une exception près. Depuis la prolifération des plateformes mobiles, le développeur doit choisir la ou les plateformes cibles sur lesquelles l'application devra fonctionner correctement et ceci avant le démarrage du projet. Nous pouvons faire l'analogie avec le développement d'un site Internet ou d'un service Web qui doit fonctionner correctement et offrir un service de qualité [KSB+10] sous Internet Explorer 7, 8 ou 9, Opera 10 ou Firefox 3 ou 4, avec Windows XP, Windows Seven ou sous Mac OS et Linux.

Dans le cas d'un développement Internet, le développeur utilisera les mêmes technologies, *i.e.* les mêmes langages de programmation, le même éditeur et pourra s'assurer du bon fonctionnement de son site Internet directement sur son poste de travail. Quant au développement d'une application mobile, le choix des plateformes a des conséquences sur le projet, notamment sur la charge de travail, et donc sur le coût. En effet, les différences entre les nouvelles plateformes mobiles sont multiples et à tous les niveaux : OS et versions de ces OS, langages de programmation, environnements de développement, fabricants, etc. (cf. tableau 1.1). Dans [GE11], les auteurs ont présenté une étude comparative d'une application simple développée pour 4 plateformes mobiles différentes, ils concluent cette expérience en disant : *Devices vary along so many axes that it's almost impossible to write a single version of a mobile application.*

Dans l'optique d'apporter une solution à ce deuxième type de fragmentation, certains travaux tels que PhoneGap [AGL+10], XMLVM [PY10] ou encore NeoMAD [26], ont vu le jour. Il s'agit principalement d'outils qui permettent, à partir d'un langage d'entrée unique, de générer des binaires compatibles pour certaines plateformes. D'autres travaux tels que [MPS03] et [EVP00] proposent des solutions basées sur l'ingénierie dirigée par les modèles. Le principe consiste à utiliser des transformations de modèles abstraits d'applications mobiles afin de générer le code source et donc les exécutables adéquats pour chaque plateforme mobile cible.

Pour résumer cette section, nous pouvons dire que le développeur d'applications mobiles est confronté d'une part à la fragmentation Java ME, s'il souhaite cibler tous les téléphones standard appelés aussi « *feature phones* » et les *smartphones* milieu de gamme, d'autre part à la fragmentation des plateformes s'il souhaite cibler les *smartphones* de nouvelle génération. Ajoutons à cela la croissance du marché des tablettes Internet qu'il faudra aussi inclure dans les plateformes cibles. En fonction des spécificités du projet, c'est au développeur de décider s'il faut utiliser un outil de portage ou développer le même projet nativement pour chaque plateforme mobile ; le coût et la charge ne seront évidemment pas les mêmes. Dans un cas comme dans l'autre, une fois le développement terminé, le développeur devra s'assurer du bon fonctionnement de son application sur l'ensemble des téléphones qu'il souhaite cibler, pour cela il doit rigoureusement tester son application. Nous constatons donc, et mettons en lumière dans la section suivante, les

26. www.neomades.com

OS	**Langages**	**Versions**	**Environnement de développement**
Android	Java, C	1.5, 1.6, 2.0, 2.1, 2.2, 2.2.3, 3.0...	Eclipse (Windows, Linux, Mac OS)
Bada	C, C++	1.0, 1.1, 1.2, 2.0	RCP Eclipse (Windows)
BlackBerry	Java RIM, J2ME	4.5, 4.6, 4.6.1, 5.0, 6.0...	Eclipse, JDE, Netbeans (Windows, Linux)
MeeGo	Qt, Web Apps, C++	1.0, 1.1	MeeGo SDK (Ubuntu Lucid et Fedora 13 (32bit))
iPhone	Objective-C	3.1.3, 4.2.1, 4.2.6, 4.3	XCode (Mac OS)
Symbian OS	C, C++, Qt, J2ME	8.0, 8.1, 9.0...9.5	Eclipse, Qt Creator... (Windows, Linux, Mac OS)
WebOS	HTML, CSS, Javascript, C	1.0.1, 1.0.2..., 2.2, 3.0	Eclipse, Ares (Windows)
Windows Mobile	C# .NET CF, J2ME	6.0, 6.5	Microsoft Visual Studio (Windows)
Windows Phone	C#, Silverlight, XNA	-	Microsoft Visual Studio (Windows)
BREW	C, C++	2.x, 3.1	Brew SDK, Eclipse, Visual Studio (Windows)

TABLE 1.1 – Aperçu des différentes plateformes (systèmes d'exploitation) mobiles

spécificités du test résultant de cette fragmentation difficile à maîtriser. Evidemment, les enjeux de test s'avèrent grandissants, compte tenu de la multiplicité des codes spécifiques à gérer.

1.2 Le test des applications mobiles

Nous commençons cette section par une brève introduction au test de logiciel puis nous montrons les spécificités du test des applications mobiles, les outils existants et leurs limitations.

1.2.1 Le test de logiciel

Testing is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. [IEE90]

Dans tout projet informatique, le test est une activité incontournable. Le but du test est de valider le bon fonctionnement des programmes. Pour cela, le test a pour rôle de détecter les erreurs de conception et les erreurs de réalisation. De manière générale, le test consiste à exécuter un programme avec un ensemble de données en entrée et à comparer les résultats obtenus avec les résultats attendus. Si les résultats diffèrent, alors une erreur a été détectée et donc il faut la localiser et la corriger. Quand l'erreur est corrigée, il convient de re-tester entièrement ou partiellement le programme pour s'assurer de la correction et de la non régression du programme [LT04]. Il existe plusieurs types de tests avec des objectifs différents [LBN09]. Etant donné que nous nous intéressons au monde de la téléphonie mobile, nous présentons ici tous les tests subis par le logiciel du téléphone lui-même (*firmware* ou système d'exploitation). La majorité de ces tests est aussi applicable à tout système informatique.

Tests unitaires : il s'agit des premiers tests effectués par les développeurs afin de s'assurer du bon fonctionnement d'une partie déterminée d'un logiciel. Ce type de test peut se faire dès le démarrage du projet et à chaque fois que le code est modifié. Certaines techniques de développement logiciel, telles que Test Driven Development (TDD) [Bec02], sont guidées par ces tests ;

Tests de fonctionnalité : ils sont effectués par l'équipe de développement et/ou l'équipe d'intégration. L'objectif de ces tests est de répondre aux exigences des spécifications d'une fonctionnalité donnée ;

Tests d'intégration : ils servent à vérifier que les fonctionnalités s'interconnectent correctement entre elles (contrats d'interface). Ces tests peuvent porter sur des intégrations partielles (seuls quelques modules/fonctionnalités) avant de porter sur une intégration complète et donc sur le logiciel entier ;

Tests de conformité : ils ont pour objectif de s'assurer que certaines implémentations sont conformes à des standards. Ils sont souvent passés par ou avec l'aide d'organismes spécialisés pour la certification (Java, 3G/4G ou encore Bluetooth) ;

Tests système : ils servent à vérifier qu'il n'y a pas d'effet de bord non désiré entre les différentes fonctionnalités. La sous-catégorie dite « *smoke tests* » permet, par un jeu de tests restreint, de s'assurer qu'il n'y a pas un problème majeur de stabilité ;

Tests de stress/robustesse : ces tests ont pour objectif de s'assurer qu'en cas d'erreurs, aucun dommage irréparable (perte de données, corruption de fichiers ou encore accès non prévu à des informations) n'est subi par le logiciel. Ils servent aussi à vérifier les temps de réponse et les limites du logiciel ;

Field tests : ce type de test est très spécifique au monde de la téléphonie mobile. Il est souvent effectué par une équipe spécialisée pour vérifier la compatibilité des piles 2G/3G/4G avec les infrastructures réseaux prévues chez les opérateurs. Cette équipe passe souvent deux semaines à silloner les routes de plusieurs pays en prenant des traces réseau ;

Tests de performance : ces tests sont souvent réalisés au niveau système et avec une forte implication des équipes d'intégration et d'optimisation ;

Friendly User Test (FUT) : ils sont comparables à des *beta-tests*, mais en interne. Ils permettent une meilleure vision de la perception client du produit avant commercialisation.

Le but de tous ces tests est de détecter le plus d'anomalies possibles et le plus tôt possible dans le projet. En effet, la correction d'un bug détecté au niveau des tests unitaires est moins coûteuse que la correction d'un bug en test d'intégration.

Nous voyons dans les sections suivantes que le test d'applications mobiles est un mélange de la majorité des types de tests énumérés ci-dessus, *i.e.* tests de fonctionnalité, de stress/robustesse, de performance, d'expérience utilisateurs, etc. Les techniques de tests utilisées diffèrent en fonction de l'objectif et des outils utilisés mais l'activité de test comporte généralement toujours les mêmes phases [Pac05] :

1. **génération des jeux de test** à partir des spécifications, du code ou encore du manuel d'utilisation ;
2. **exécution des jeux** de test produits. Cette exécution est une interaction entre un testeur et le logiciel à tester souvent dans son environnement de déploiement final. Dans notre cas (voir section 1.2.3), cette phase consiste à **exécuter des scénarios de tests sur tous les téléphones ciblés par le développeur** ;

3. **interprétation des résultats** afin de détecter les éventuelles erreurs du logiciel testé.

Les phases 1 et 2, *i.e.* la génération et l'exécution des tests, dépendent du niveau d'accessibilité et d'observabilité des éléments donnés (spécifications, code source, système à tester, etc.). Cette observabilité peut être classifiée selon deux types :

- **boîte blanche** - les tests peuvent s'effectuer sur les données internes et/ou externes du programme. En d'autres termes, nous disposons du code source du programme ;
- **boîte noire** - le test est réduit à l'interface du système à tester. Dans ce cas, le code et l'architecture interne de l'implantation à tester ne sont pas connus. Seul le comportement du programme lié aux interactions avec son environnement est perçu.

Le test auquel nous nous intéressons dans cette thèse rentre dans le deuxième type, *i.e.* « boîte noire » (*Black Box* en anglais). En effet, nous nous intéressons au comportement de l'application au cours de son exécution sur le téléphone et de son interaction avec l'utilisateur et avec le réseau mobile.

1.2.2 Simulateurs ou téléphones réels

L'une des particularités du développement sur systèmes embarqués réside dans le fait que la plateforme de développement soit différente de la plateforme de déploiement (d'exécution finale). Dans le cas du développement d'une application mobile, le développeur réalise son programme sur PC puis le transfère sur téléphone. Avant de le transférer sur la ou les plateformes mobiles cibles et de le mettre à la disposition des utilisateurs finaux, le développeur doit vérifier le bon fonctionnement de son programme. Dans ce contexte, il existe des simulateurs tels que le WTK [27] de Sun ou encore le SDK Android [28] pour permettre la simulation d'une application mobile sur ordinateur. Comme nous l'avons précisé dans l'introduction de ce manuscrit, ces simulateurs ont souvent des comportements et des caractéristiques très différents des vrais téléphones et ne permettent pas de tester l'interaction de l'application avec le réseau mobile (communication Internet, appels entrants, SMS...). Compte tenu de la spécificité de chaque réseau, ce dernier point est très important. En effet, une application qui fonctionne avec un réseau donné pourrait ne pas fonctionner correctement avec un autre, notamment à cause de restrictions techniques (ports bloqués, bande passante insuffisante...). Ainsi, le test de validation d'une application mobile ne peut se limiter à l'utilisation des simulateurs, même si ces derniers sont indispensables pour la mise au point de l'application pendant son développement. Ce test sert à garantir à l'utilisateur final que le programme fonctionne correctement (démarre, s'exécute et se termine sans erreurs, ne fasse que ce pour quoi il a été prévu, n'altère pas les données personnelles de l'utilisateur...) et sert aussi

27. Wireless ToolKit : http ://www.oracle.com/technetwork/Java/index-jsp-137162.html
28. SDK Android : http ://developer.android.com/sdk/index.html

à s'assurer que l'application puisse réussir les tests de conformité imposés par la majorité des distributeurs d'applications. Suite à une initiative de Sun et de plusieurs acteurs du marché mobile, une liste exhaustive des tests à effectuer sur toute application mobile a été établie [29] .

1.2.3 Procédure de test

Le processus de test d'une application mobile pourrait varier d'une entreprise à une autre. Mais les étapes de réalisation d'un test sont quasiment les mêmes partout. En effet, les étapes ci-dessous sont incontournables :

1. mettre la carte SIM dans le téléphone et le démarrer ;
2. règler certains paramètres - accès Internet, date et heure ou encore connexion Bluetooth ;
3. mettre à disposition les binaires à tester (souvent sur un serveur accessible via Internet) ;
4. télécharger et installer l'application ;
5. naviguer jusqu'à l'application et la démarrer ;
6. réaliser le test effectif ;
7. reporter les bugs en utilisant un tableau électronique ou un outil de gestion de bugs ;
8. désinstaller l'application et effacer ses données.

La sixième phase est la plus importante dans ce processus de test, elle est aussi la plus coûteuse en temps. Le test d'applications mobiles est assez similaire à celui des interfaces graphiques standard (applications PC ou Internet) dans le sens où il s'agit d'effectuer une suite d'actions comme appuyer sur des touches ou cliquer sur une zone de l'écran puis interpréter par la suite les résultats, *i.e.* « le menu est correctement affiché », « le bouton a la bonne dimension », etc. Le testeur doit répéter ces actions sur l'ensemble des téléphones sachant que si l'application n'a pas été validée, il faudra refaire totalement ou partiellement toutes les étapes jusqu'à ce qu'elle le soit. Souvent il faut mettre en place une procédure itérative entre le développeur qui corrige les bugs et le testeur qui valide ces corrections. Cette phase de test a pour but de s'assurer des points suivants :

- **caractéristiques de l'application** - le testeur doit vérifier toutes les informations de type nom de l'application, nom du développeur, icône ou encore numéro de version ;
- **démarrage de l'application** - le testeur doit s'assurer que l'application démarre « proprement » sans générer d'exception ;

29. Unified Testing Criteria : http ://javaverified.com/graphics/PDF/UTC_3_0_FINAL.pdf

- **stabilité** - l'application doit s'installer et s'exécuter « proprement » sans se bloquer ou s'arrêter inopinément. Elle ne doit pas être gourmande en consommation batterie et mémoire, elle doit gérer « proprement » les interruptions utilisateurs (fermeture du clapet, branchement du chargeur...) et les interruptions systèmes (appel entrant, SMS...). En général, l'application doit se mettre en pause et afficher un écran pour avertir l'utililsateur. Si l'application propose de jouer un son, celui-ci doit s'arrêter dans le cas d'un appel entrant. L'application doit aussi être capable de gérer les actions imprévues des utilisateurs comme par exemple l'appui sur des touches de façon aléatoire ;
- **interface utilisateur** - cette partie du test est assez importante. Le testeur doit s'assurer que tous les composants graphiques sont « proprement » affichés (couleurs, images, animations, menus, boutons...). Il doit aussi vérifier que le temps de réponse à une action donnée est raisonnable ;
- **internationalisation** - pour les applications proposant plusieurs langues, le testeur doit s'assurer que tous les textes sont « proprement » affichés, que le choix de langue de l'utilisateur est sauvegardé et qu'au prochain démarrage, la langue par défaut sera celle définie par l'utilisateur ;
- **fonctionnalités** - ce point sert à vérifier que seules les fonctionnalités décrites dans l'aide et la description de l'application sont fournies. L'application ne doit en aucun cas cacher d'autres fonctionnalités qui pourraient éventuellement accéder à des informations personnelles et/ou les altérer ;
- **connectivité** - dans le cas où l'application utilise des connexions Internet, le testeur doit vérifier qu'elle gère « proprement » les problèmes de connexions (délais trop longs, perte de connexion, connexion impossible...). Ceci est aussi valable pour les autres types de communication tels que SMS/MMS ou Bluetooth ;
- **gestion des informations personnelles** - l'application ne doit pas altérer les informations personnelles de l'utilisateur (contacts, notes et agenda).

Souvent, pour que le testeur puisse effectuer les tests efficacement et afin de ne pas reporter de faux bugs, le développeur écrit un plan de test dans lequel il détaille le cheminement à suivre. Un exemple de plan de test est fourni dans l'annexe A. En plus du plan de test, on fournit aussi au testeur un schéma global (appelé aussi *storyboard*) (cf. figure 6.1) de l'application ainsi que des captures d'écran des différents menus et résultats attendus.

Il existe aujourd'hui deux approches pour tester un logiciel en général : la première consiste à le faire manuellement par un être humain, la deuxième consiste à l'automatiser et le faire faire par un robot. Ces deux approches sont complémentaires. Il est, en effet, impossible d'automatiser tous les types de test. Nous verrons dans les sections suivantes ces deux approches appliquées au test d'applications mobiles et leurs limitations. Nous verrons aussi l'absence d'une vraie solution d'automatisation globale à cause des différences entre mobiles.

1.2.4 Le test manuel

Comme mentionné dans la section précédente, afin de s'assurer de la qualité d'une application mobile, le développeur n'a d'autre choix que de la tester sur tous les téléphones ciblés ou, du moins, sur les téléphones de référence [30]. Le test manuel est la solution classique la plus utilisée par les développeurs. Elle consiste à faire tous les tests, un par un, à la main. Cette approche est très efficace en termes de résultat. En effet, les testeurs peuvent détecter des bugs qui n'ont pas été initialement prévus dans le plan de test décrit par le développeur de l'application. Ceci rend le test plus exhaustif et plus proche de ce que l'utilisateur final attend. Le testeur saisit par la suite les résultats dans un tableau électronique ou dans un outil de suivi de bugs, le développeur corrige le bug, génère une nouvelle version de l'application et le testeur reprend la procédure de test. L'investissement dans l'achat de terminaux, qui sortent à un rythme soutenu, la formation des testeurs, la mise en place d'un processus de test et des outils nécessaires... font du test manuel d'applications mobiles l'une des phases les plus coûteuses d'un projet. En plus, il s'agit d'une tâche peu reproductible, longue et laborieuse où le testeur refait exactement la même chose pour chaque téléphone. Aujourd'hui, nous distinguons principalement deux catégories de testeurs : les équipes dédiées et les communautés.

1.2.4.1 Equipe de test dédiée

Face au nombre important de tests à effectuer pour la validation d'une application mobile, certaines sociétés de développement mobile optent pour la mise en place d'une équipe dédiée au test. Afin de minimiser les coûts, plusieurs entreprises ont recours à des équipes de testeurs offshore. L'équipe de testeurs offshore offre un avantage certain au niveau coût mais exige une importante formation au démarrage, un suivi rigoureux, des outils performants pour remonter les bugs avec un processus clair. Il est en revanche très difficile d'utiliser une organisation offshore pour les tests d'intégration et de performances. En effet, il est souhaitable de travailler main dans la main avec les équipes de développement pour la montée en compétences sur les nouvelles fonctionnalités qui évoluent encore, sous peine de voir remonter de nombreux faux problèmes. Indépendamment du domaine du test d'applications mobiles, l'offshore présente souvent des risques et cache souvent des dépenses imprévues [Ove02].

1.2.4.2 Communauté de testeurs

Certaines entreprises optent pour des tests beta menés auprès d'utilisateurs qui ne sont pas des testeurs professionnels. Il existe aujourd'hui plusieurs plateformes Internet qui proposent de mettre en contact des développeurs et des testeurs. [Vuk09] propose une taxonomie de toutes ces platformes. Parmi ces dernières, nous pouvons citer uTest [31] ou encore Mob4hire [32]. Le développeur a la possibilité de choisir les membres de la

30. Des téléphones représentatifs de l'ensemble des téléphones à cibler.
31. http ://www.utest.com
32. http ://www.mob4hire.com

communauté qu'il souhaite faire participer aux tests. Ce choix se fait principalement en fonction de leurs mobiles, du réseau et de leur âge afin de mieux cibler sa clientèle finale. Ces plateformes offrent aussi l'environnement technique pour la mise en ligne des applications, des outils de suivi de bugs et des procédures de rémunération des testeurs. Ce type de solution connu sous le nom de *Crowdsourcing* [SHS09] peut effectivement être efficace pour une première phase de *beta-test* dont le but est de valider certains points clés d'une application (la facilité d'utilisation, fluidité ou encore *game play* s'il s'agit d'un jeu). Toutefois, elle ne peut servir de test final rigoureux et complet. En effet, les membres de la communauté n'étant pas des testeurs professionnels, le développeur ne pourra pas compter sur leur contribution pour avoir des tests exhaustifs.

1.2.5 Le *cloud testing*

Afin de résoudre la problématique de procuration des téléphones mobiles et de suivre l'évolution effrénée du marché, certaines entreprises ont lancé des plateformes de test à distance (banc de test en ligne). L'idée globale est de fournir un service Internet qui permet d'accéder et de manipuler de vrais téléphones mobiles via le Web (cf. figure 1.2). Cette approche est connue sous le nom de *cloud testing*.

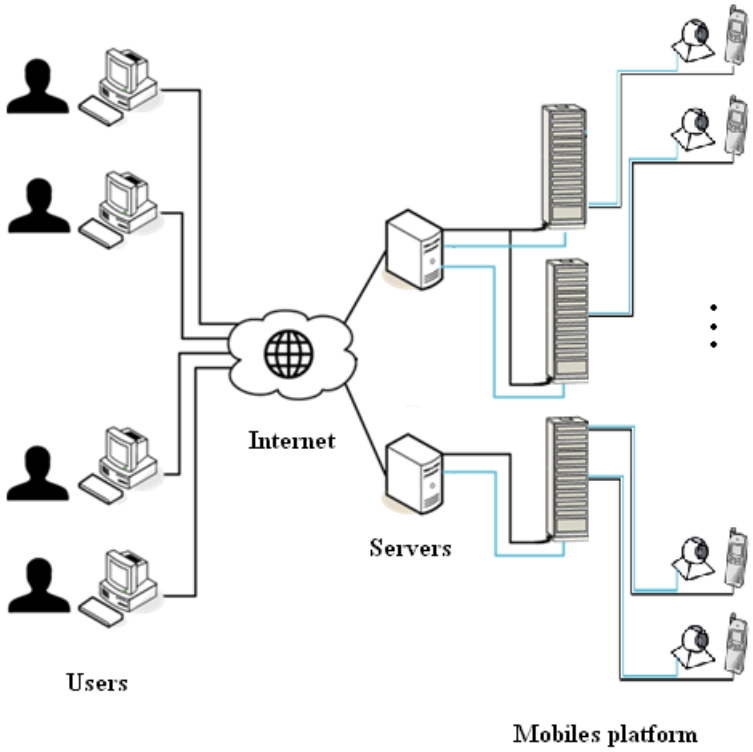


FIGURE 1.2 – Architecture globale d'une plateforme pour le test à distance

Parmi ces plateformes, nous pouvons citer Device Anywhere [33], Keynote [34] ou encore Perfecto Mobile [35]. Pour nos travaux de recherche, nous avons fabriqué un prototype de banc de test que nous présenterons dans le chapitre 5. Les architectures de ces plateformes sont légèrement différentes mais offrent quasiment toutes le même service, c'est-à-dire un accès à des téléphones mobiles connectés aux différents réseaux de télécommunication du monde entier (ceci est possible en déployant physiquement les bancs de test dans des pays différents).

L'utilisateur se connecte au banc de test via une application Internet et peut ainsi manipuler n'importe quel téléphone disponible. Une fonctionnalité de *capture-and-replay* [BM07] est disponible. Ainsi, l'utilisateur peut jouer un scénario sur un téléphone donné, un script est alors généré. Ce dernier est une succession d'actions (cf. listing 1.2) effectuées sur le téléphone (appui sur une touche, appui sur l'écran...). Ce type de test est connu sous le nom de *Event-Driven Testing* et utilisé pour les interfaces graphiques, les applications Web, les drivers et aussi les systèmes embarqués [Mem07].

Listing 1.2 – Exemple de scénario de test

```
1 pressKey (UP)
2 pressKey (FIRE)
3 wait (2000)
4 pressKey (DOWN)
5 writeText ("demo")
6 pressKey (FIRE)
7 wait (3000)
```

Ces séquences peuvent être rejouées sur le même téléphone ou sur des téléphones similaires qui ont le même type de clavier, la même résolution d'écran ou encore le même temps de réponse. En effet, ces trois points sont importants. Prenons par exemple la saisie d'un texte (ligne 5 du listing 1.2) afin d'écrire le mot « demo » sur des téléphones dont les claviers sont différents (cf. figure 1.3) : l'utilisateur devra effectuer des combinaisons de touches différentes.

FIGURE 1.3 – Exemple de trois claviers différents

L'utilisation de la méthode *capture-and-replay* pour le test d'interface graphique n'est

33. http ://www.deviceanywhere.com
34. http ://www.keynote.com
35. http ://www.perfectomobile.com

pas propre à ces plateformes de *cloud testing*. En effet, plusieurs outils tels que Selenium [36] pour le test de sites Internet ou encore TestPlant [37] et Frologic [38] permettent de tester certaines plateformes mobiles. L'utilisateur doit alors connecter le téléphone à tester à son PC via un câble USB ou une connexion basée sur le système VNC [39] puis tester son application, directement sur mobile, via ces outils.

D'après [Kra00], ce type d'outil ne sert pas à automatiser les tests, principalement parce qu'une fois le code de l'application modifié, il faudra modifier le script ou refaire le scénario pour regénérer un nouveau scénario. De plus, un script par mobile ou par famille de mobile est nécessaire pour tester tous les téléphones ciblés. Cela rend la maintenance difficile et limite la réutilisabilité.

Ce type d'outil peut être utile pendant la phase de développement et de mise au point de l'application mais ne peut pas être utilisé pour automatiser les tests, à grande échelle, sur plusieurs téléphones différents. Il est plus adapté aux sites Internet et aux applications standard.

1.3 Synthèse et objectifs

La fragmentation n'est pas un phénomène spécifique uniquement aux applications mobiles embarquées. Elle touche aussi, d'une manière plus ou moins importante, d'autres domaines tels que le développement de sites Internet mobiles [FLFL09] ou encore le développement de jeux vidéo, *i.e.* un développeur de jeux doit choisir la ou les plateformes sur lesquelles il souhaiterait rendre disponible son application. Nous avons vu dans ce chapitre la fragmentation mobile et l'impact qu'elle a sur le test des applications, notamment en termes de coût et de répétitivité. Nous avons aussi vu la spécificité de ces tests et les méthodes utilisées, notamment le *cloud testing* sur lequel notre approche est basée. En effet, nous proposons d'offrir aux utilisateurs de ces plateformes la possibilité de décrire des scénarios de test exécutables sur n'importe quel téléphone.

Le chapitre suivant présente l'ingénierie dirigée par les modèles. Nous nous intéressons particulièrement à la conception de langage spécifique et l'application de celui-ci au test logiciel.

36. http ://seleniumhq.org/
37. http ://www.testplant.com/
38. http ://www.froglogic.com/
39. Virtual Network Computing

Chapitre 2

L'ingénierie dirigée par les modèles - IDM

Sommaire

2.1 Introduction 34
2.1.1 L'exemple *MDA* 34
2.1.2 Définitions 35
2.2 Langages de modélisation 35
2.2.1 Définition d'un profil UML 36
2.2.2 Définition d'un DSML 39
2.2.3 Bilan 43
2.3 Transformation de modèles 43
2.4 Le test et les langages de modélisation 44
2.4.1 Langage de modélisation vs. langage spécifique de test . . 45
2.4.2 *Model-Based Testing* vs. *Model-Driven Testing* 47
2.5 Synthèse 49

2.1 Introduction

L'ingénierie dirigée par les modèles, ou *Model Driven Engineering* (MDE) en anglais, est de plus en plus utilisée dans des projets informatiques complexes de milieu aussi variés que l'automobile ou les télécommunications. L'adoption de l'IDM à la fois par les industriels et les universitaires a fortement contribué à faire émerger de nouveaux concepts et outils facilitant la mise en œuvre de cette approche dans la production de logiciels. L'intérêt de cette approche, qui met la notion de modèle au cœur du dispositif, réside dans le fait qu'elle offre un moyen d'abstraire un système par le biais de ces modèles, en simplifiant sa manipulation et son appréhension par les concepteurs. L'IDM apporte aussi une grande part d'automatisation des processus de développement, essentiellement grâce aux transformations de modèles.

L'ingénierie dirigée par les modèles ne se limite pas à l'utilisation de modèles UML (*Unified Modeling Language*) [OMG10] dans les phases préliminaires du développement d'un logiciel. Cette utilisation limite souvent l'apport des modèles à un rôle de spécification initiale indépendante du code final de l'application. La modélisation est pour beaucoup de développeurs une étape obscure voire inutile : cette réputation est principalement due au fait d'isoler la phase de modélisation du reste de la réalisation d'un projet, notamment l'implémentation et le test. L'un des principaux but de l'IDM est de capitaliser le savoir-faire au niveau des modèles et non plus au niveau du code source, tout en ayant comme objectif final de gérer une application sous la forme d'un code afin de pouvoir l'exécuter sur une ou plusieurs plateformes cibles.

Nous présentons dans ce chapitre les principes clés de l'IDM et les différents aspects qui nous intéressent dans le cadre de la définition d'un langage spécifique de domaine ou *Domain-Specific Modeling Language* (DSML) en anglais et son utilisation pour le test.

2.1.1 L'exemple *MDA*

L'un des exemples les plus connus de mise en œuvre de l'IDM est l'approche *Model Driven Architecture* (MDA) [Sol00], lancée en 2000 à l'initiative de l'OMG [40]. L'objectif de cette initiative est de promulguer de bonnes pratiques de modélisation et d'exploiter pleinement les avantages des modèles. Le principe clé de cette méthode de conception est basé sur l'exploitation de modèles UML abstraits d'un système appelés *Platform Independent Model* (PIM), pour obtenir des modèles, appelés *Platform Specific Model* (PSM), spécifiques à la plateforme d'accueil du système (.NET, Java EE, etc.). Le passage de PIM à PSM est le résultat de transformations automatiques. Le modèle PSM est par la suite utilisé pour générer le code de l'application (cf. figure 2.1 [41]).

40. Object Management Group : http ://www.omg.org
41. Figure extraite de : http ://web.univ-pau.fr/~ecariou/cours/idm/cours-intro.pdf

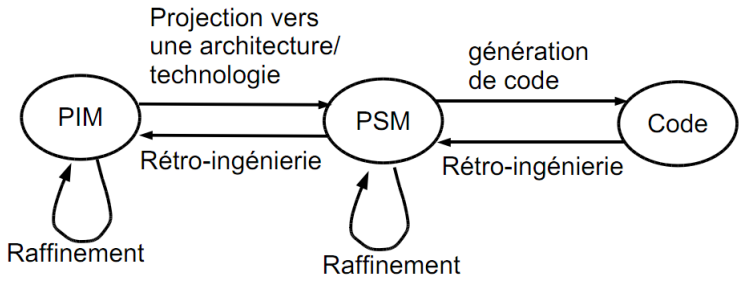


FIGURE 2.1 – Principes de l'approche MDA

2.1.2 Définitions

Dans [MFBC10] les auteurs font une classification des différentes définitions données aux modèles. Nous adoptons celle-ci : Un modèle est une abstraction d'un système. Ce modèle est conforme à un métamodèle qui, lui, est spécifique à un domaine donné et composé d'un ensemble de concepts et de leurs relations. Il est possible d'avoir plusieurs métamodèles pour un même domaine mais un modèle est conforme à un seul méta-modèle. Il est possible de déroger à ce principe [KÖ6] mais cela suppose de considérer d'autres types de métamodélisation comme les ontologies par exemple. Un métamodèle est lui-même conforme à un méta-métamodèle dont le rôle est de fournir un langage pour la définition de métamodèles.

Une architecture de métamodélisation basée sur quatre couches a été proposée par l'OMG dans [UML09]. La figure 2.2 présente ces différentes couches.

- **M0** : niveau le plus bas de l'architecture où se trouvent les instances des modèles « utilisateur » (objets du monde réel) ;
- **M1** : niveau qui contient les modèles ;
- **M2** : niveau du métamodèle, c'est-à-dire des langages de modélisation utilisés pour définir des modèles (par exemple UML) ;
- **M3** : correspond au méta-métamodèle.

Différents travaux ont investigué la couche M3 et ont proposé plusieurs alternatives (cf. paragraphe suivant). Ces dernières ont facilité la définition de nouveaux métamodèles et de nouveaux langages.

2.2 Langages de modélisation

Nous distinguons deux grandes catégories de langages de modélisation, les spécifiques ou *Domain-Specific Modeling Languages*, répondant aux exigences d'un domaine bien précis, et les génériques ou *General Purpose Modeling Languages* en anglais, comme UML, qui peuvent s'adapter à tous les domaines et problèmes de conception. En utilisant

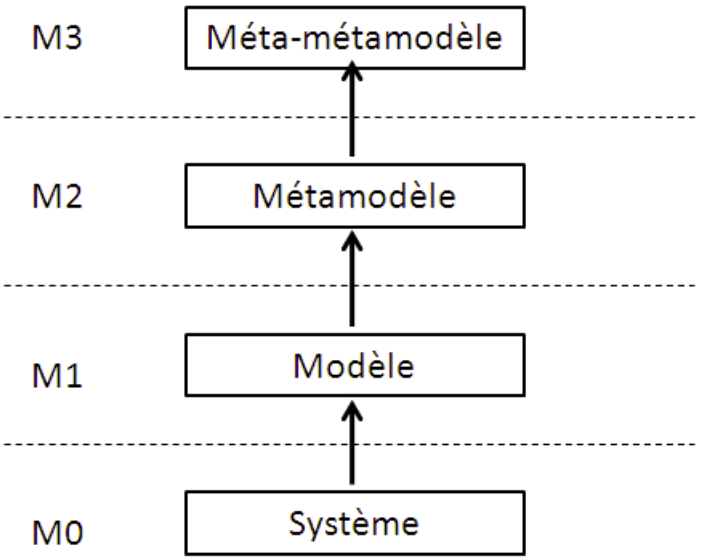


FIGURE 2.2 – L'architecture de métamodélisation en couches [UML09]

UML, le processus de développement est plus long par rapport à l'utilisation d'un DSML parce qu'on part de modèles plus abstraits et moins ciblés sur les notions du domaine. Cependant, en utilisant UML, les modèles pourront être plus facilement accessibles alors qu'avec un langage de modélisation spécifique, le processus est uniquement maitrisé par les experts du domaine.
UML est un langage de modélisation générique destiné à la modélisation de systèmes logiciels, il permet de décrire différentes vues systèmes et il est inspiré des concepts des langages orientés objets. La nature générale de ces concepts a pour conséquence qu'UML peut être étendu pour décrire n'importe quel domaine spécifique. Il est toutefois possible d'adapter la sémantique de celui-ci à un domaine particulier. Ceci est possible grâce aux mécanismes d'extensions que le métamodèle UML fournit. Un ensemble cohérent d'extensions est appelé un profil UML [OMG10]. En effet, la création d'un profil vise à définir une terminologie, une notation spécifique à un domaine, ainsi qu'une syntaxe d'éléments spécifiques.

2.2.1 Définition d'un profil UML

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The profiles mechanism is consistent with the OMG Meta Object Facility (MOF). [UML09]

Le métamodèle standard UML fournit des mécanismes d'extension appelés valeurs marquées, stéréotypes et contraintes. Ces éléments permettent d'adapter la sémantique sans changer le métamodèle d'UML. C'est pour cela qu'ils sont souvent référencés comme mécanismes d'extension légers (Lightweight extension). Nous détaillons ce mécanisme plus bas. A l'opposé, le standard MOF [OMG00] fournit des moyens pour étendre les métamodèles, comme la définition de nouvelles méta-classes, qui sont référencés comme des mécanismes d'extension lourds (Heavyweight extension).

Un profil permet de définir la sémantique des règles et des contraintes d'usage du métamodèle UML. D'un point de vue technique, un profil est un paquetage stéréotypé qui peut importer des sources externes. Un profil est un ensemble de stéréotypes spécifiques à un domaine. Un stéréotype est défini comme extension d'une méta-classe UML, il peut avoir des propriétés et/ou des opérations particulières. Ainsi, un profil est une spécification qui spécialise un ou plusieurs métamodèles standards appelés les métamodèles de référence. Le profil est alors dédié à un domaine spécifique de ces métamodèles de référence. La figure 2.3 montre un exemple de profil pour la modélisation de composants EJB [42] ; le profil est défini ici sous la forme d'un diagramme de classe décrivant les stéréotypes et les définitions de valeurs marquées du profil EJB ainsi que les éléments du métamodèle d'UML auxquels ils se rattachent.

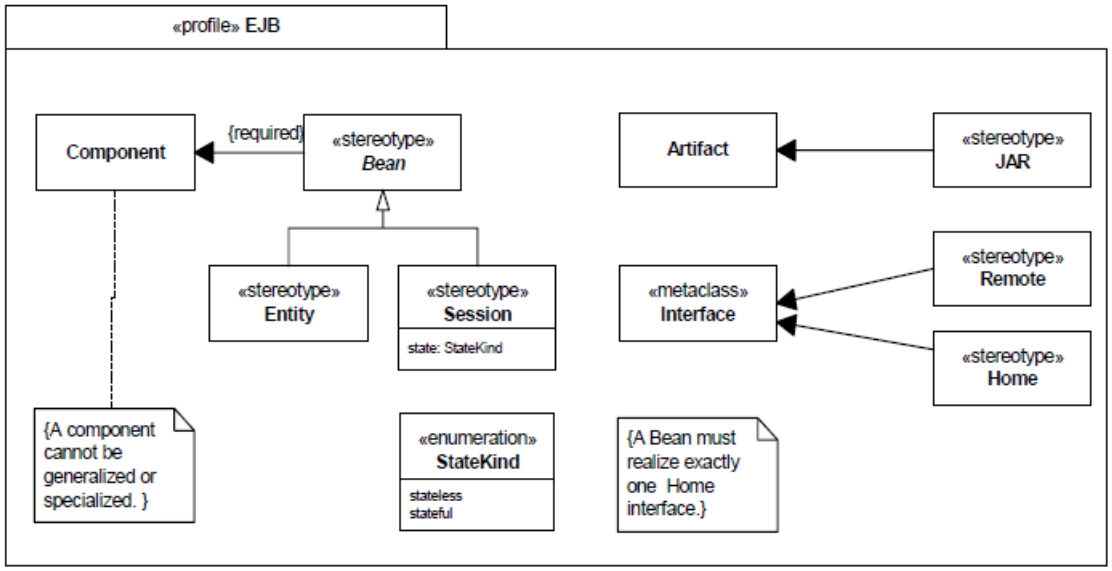


FIGURE 2.3 – Exemple d'un profil UML pour les EJB [UML09]

Dans [Laf04], l'auteur présente une synthèse des différents éléments structurant un

42. Entreprise JavaBeans : http ://www.oracle.com/technetwork/java/javaee/ejb/index.html

profil UML à savoir :

- **les éléments sélectionnés du métamodèle de référence** - un profil fournit la sélection du métamodèle de référence qui constitue la focalisation particulière choisie. Cette sélection n'exclut pas les autres éléments du métamodèle de référence, mais simplement spécifie ceux qui sont spécialisés ;

- **les stéréotypes** - un stéréotype est défini pour une méta-classe spécifique du métamodèle de référence. Dans un profil UML, le stéréotype crée une méta-classe UML virtuelle basée sur la méta-classe UML existante. Il fournit ainsi un moyen de classer les instances de cette méta-classe de base, et peut également aussi spécifier des contraintes additionnelles ou des valeurs marquées requises ;

- **les définitions de valeurs marquées** - la sémantique d'une valeur marquée est définie pour une méta-classe spécifique du métamodèle de référence. La définition d'une valeur marquée contient le nom des valeurs marquées correspondantes, le type des valeurs qu'elles peuvent prendre, la description de la sémantique et des contraintes s'appliquant à chaque valeur marquée correspondante. Comme mécanisme d'extension d'UML, la valeur marquée agit comme un attribut d'une méta-classe UML, permettant ainsi l'attachement arbitraire d'informations à une instance. Un ensemble de valeurs marquées peut être associé à un stéréotype afin d'être appliqué aux éléments de modélisation portant ce stéréotype ;

- **les contraintes** - elles peuvent être définies au niveau d'une méta-classe particulière comme au niveau d'un stéréotype particulier. Elles permettent de spécialiser davantage la sémantique des éléments du métamodèle de référence utilisés dans le profil. Cette spécification est écrite sous la forme d'une expression dans un langage de contrainte particulier. Le langage de contrainte formel utilisé par le métamodèle UML est le *Object Constraint Language* (OCL). Mais les contraintes peuvent être spécifiées de manière informelle en langage naturel. Une ou plusieurs contraintes peuvent être appliquées à tout élément de modélisation pour spécifier l'utilisation de ses instances. De plus, les contraintes peuvent être associées à un stéréotype pour ainsi ne s'appliquer qu'à des éléments de modélisation classés par ce stéréotype ;

- **les descriptions** - il est possible de préciser la sémantique d'un profil (comme des éléments qu'il contient) par des descriptions en langage naturel. Par exemple, les objectifs d'un profil ou sa compatibilité avec d'autres profils peuvent ainsi être décrits en détail ;

- **la notation** - la notation d'UML peut être personnalisée par le mécanisme des profils : définition d'icônes associés aux stéréotypes, disposition pour les diagrammes, etc. ;

- **les règles** - les profils doivent être capables de définir des règles dédiées à leur domaine spécifique. Elles peuvent être de différents types.

  - **Règles de transformation** pour exprimer comment un modèle peut être transformé pour être modélisé ou implémenté vers un but spécifique. Par exemple, le profil CORBA exprime comment un modèle UML peut être transformé en une implémentation CORBA. Certaines règles de transformation permettent la définition de produits de développement ou bien assistent ou automatisent le développement de certains types d'activités ;

  - **Règles de validation** pour vérifier que le modèle possède les bonnes propriétés du domaine du profil. Ces règles vérifient les critères de cohérence sur le modèle ;

  - **Règles de présentation** pour définir quels types d'éléments de modélisation doivent apparaître dans tel type de diagramme et indiquer aussi quelles informations doivent être cachées.

Nous venons de voir qu'un profil est une extension d'un métamodèle de référence ou un autre profil. Nous verrons dans la section suivante qu'un DSML est un métamodèle construit à partir de zéro.

2.2.2 Définition d'un DSML

L'utilisation d'un DSML aide à se concentrer sur un sujet déterminé en fixant un cadre précis. D'un côté, un champ d'application soigneusement défini pour faire en sorte que les connaissances et l'expertise soient capturées d'une manière structurée et détaillée par les experts du domaine. D'un autre côté, l'expertise et les compétences sont capitalisées et réutilisées par les utilisateurs du domaine. Le même niveau d'expression et la compréhension d'un domaine sont possibles en utilisant un langage générique tel que UML, mais le niveau attendu de connaissances concernant le domaine est considérablement plus élevé par rapport à une approche basée sur un DSML. De plus, celle-ci exige également moins de temps pour apprendre à utiliser les outils associés.

La définition d'un DSML est une tâche dont la complexité est proportionnelle à celle du domaine cible. Certaines règles sont à respecter sinon on risque de tomber dans les travers de cette approche. Dans [Kov07], l'auteur liste quelques mauvaises pratiques. Parmi ces dernières, nous pouvons citer :

- le mélange des niveaux d'abstraction (par exemple : le mélange des cas d'utilisation et de détails de l'interface utilisateur) ;
- la définition d'un nombre important de contraintes en raison de la structure rigide du domaine à cibler ;
- l'utilisation d'un nombre important d'éléments ;
- la conception d'un langage fermé très peu évolutif ;

– la définition d'un langage trop compliqué : difficile à concevoir et à outiller qui nécessite un temps de formation long et coûteux.

La fabrication d'un DSML est principalement axée sur deux phases : l'élaboration de la syntaxe abstraite et la définition de la syntaxe concrète.

2.2.2.1 Syntaxe abstraite

Dans le contexte de l'IDM, la syntaxe abstraite est la base de tout langage de modélisation : il s'agit de l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation (méta-métamodèle), tel que le standard de l'OMG MOF [OMG00] utilisé pour UML [UML09], offrent les concepts et relations élémentaires avec lesquels il est possible de décrire un métamodèle représentant cette syntaxe abstraite. Nous disposons, à ce jour, pour décrire cette syntaxe de nombreux environnements et langages de métamodélisation : Eclipse-EMF/Ecore [SBPM09], Kermeta [43] [DFV+09], Xtext [44] [SE06] ou encore MetaEdit+ [Tol04]. La majorité de ces langages reposent sur les mêmes constructions élémentaires (cf. figure 2.4).

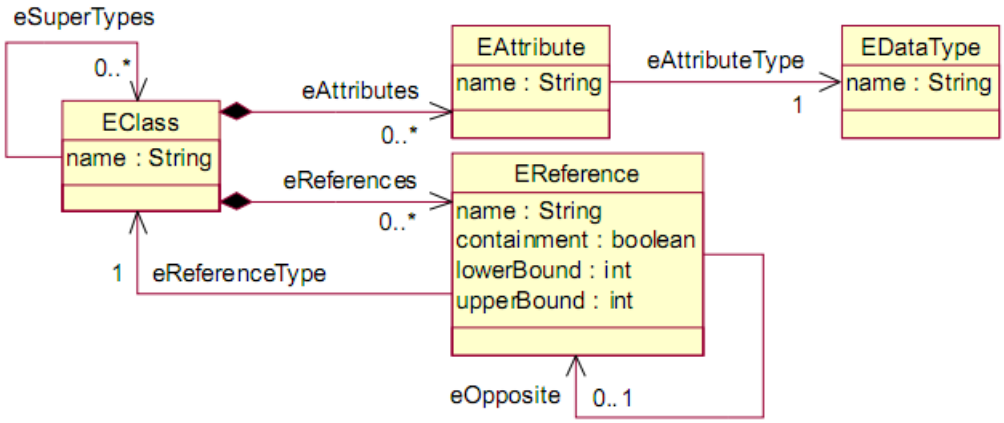


FIGURE 2.4 – Concepts de base pour la métamodélisation (EMF/Ecore)

Les langages de métamodélisation tel que Ecore ne permettent pas au concepteur d'un langage d'exprimer toutes les règles qu'il souhaite appliquer à son métamodèle. Ces règles devront être respectées par les modèles conformes à ce métamodèle. Pour exprimer ces règles, l'OMG définit le langage OCL (*Object Constraint Language*) [OMG06]. Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés qui n'ont pas pu être spécifiées par les concepts fournis par le méta-métamodèle. Il s'agit donc d'un moyen

43. http ://www.kermeta.org
44. http ://www.eclipse.org/Xtext/

de préciser la sémantique du métamodèle en limitant les modèles conformes.

La syntaxe abstraite fixe les concepts du langage mais ne leur donne aucune représentation afin de pouvoir les utiliser. Cet apsect visuel d'un DSML est sa syntaxe concrète.

2.2.2.2 Syntaxe concrète

Chaque élément d'un modèle (instance d'un métamodèle et donc du DSML) a une représentation/forme graphique ou textuelle particulière. La définition de la syntaxe concrète est donc la définition de ces décorations textuelles ou graphiques. Il est envisageable de définir plusieurs syntaxes concrètes pour une même syntaxe abstraite et donc d'avoir plusieurs représentations d'un même modèle. Le langage peut alors être manipulé avec différents formalismes mais avec les mêmes constructions et la même représentation abstraite.

La définition d'une syntaxe concrète est une tâche relativement coûteuse, même si plusieurs outils existent tel que GMF [Gro09]. En effet, le projet « *Eclipse Modeling* » [45] propose le projet *Graphical Modeling Framework* (GMF) pour définir des représentations graphiques. GMF permet de décrire la représentation graphique de chaque concept et construit un *Domain-Specific Modeler* (DSM). Les modeleurs ainsi produits possèdent une bonne ergonomie et une standardisation du format de stockage des informations graphiques. GMF demeure complexe à appréhender et requiert un niveau d'expertise élevé pour construire des modeleurs de qualité industrielle. Il existe aussi un nombre de projets qui permettent de définir des représentations textuelles. Nous citons par exemple xText [EB10] et TCS [46] [JBK06].

La figure 2.5 illustre le processus de mise en œuvre d'un DSML.

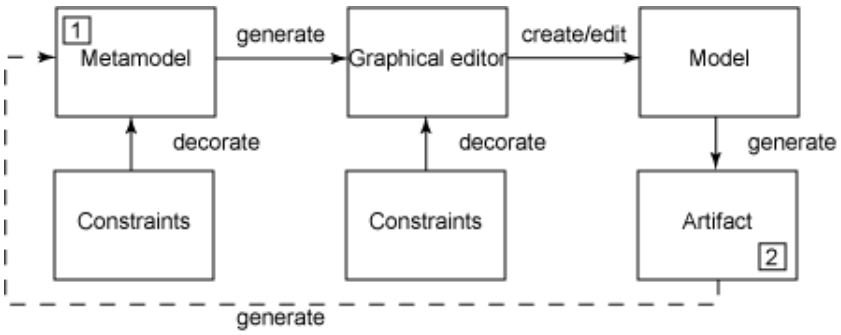


FIGURE 2.5 – Approches de mise en œuvre d'un DSML [Kov07]

45. http ://www.eclipse.org/modeling/
46. Textual Concrete Syntax : http ://www.eclipse.org/gmt/tcs/

Les modèles ne sont pas directement exploitables, dans le sens où on ne peut pas les exécuter directement. D'où la nécessité de fournir des mécanismes qui permettent de passer de ces modèles abstraits à des programmes exécutables. Ces mécanismes sont appelés « des transformations ».

2.2.2.3 Exemple de fabrication d'un DSML

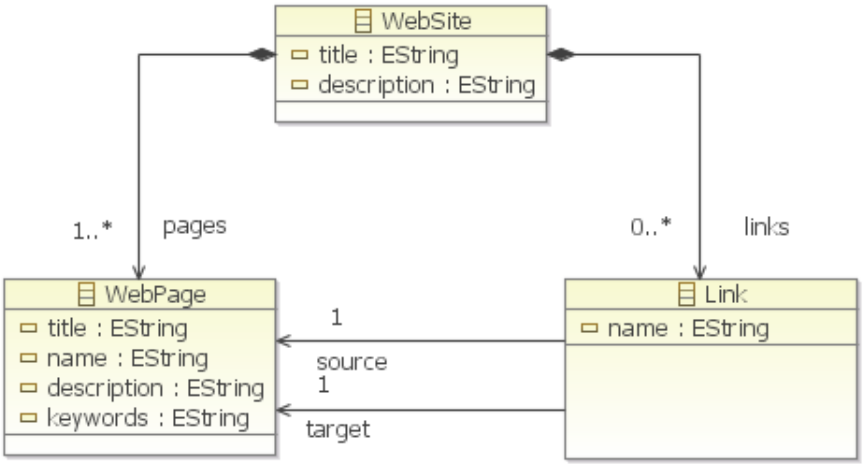


FIGURE 2.6 – Exemple d'un DSML pour la modélisation de sites Web

La figure 2.6 représente un métamodèle Ecore pour la modélisation de sites Internet ordinaires. Le point d'entrée de ce métamodèle est le métatype *WebSite* qui a deux attributs : un titre (*title*) et une description (*description*) qui sont tous les deux de type chaîne de caractères. Les deux losanges noirs sur le rectangle *WebSite* modélisent une relation de composition, c'est-à-dire, une instance de *WebSite* peut contenir une ou plusieurs instances de *WebPage* et 0 ou plusieurs instances de *Link*. Le métatype *WebPage* représente une page Web qui a un titre (*title*), un nom (*name*), une description (*description*) et des mots clés (*keywords*). Le métatype *Link* quant à lui a un seul attribut (*name*). Une instance de type *Link* est liée à deux *Webpage* ; l'une représente la source du lien (flèche de référence "source"), l'autre sa cible (flèche de référence "target"). Au niveau de ce métamodèle, rien n'interdit qu'une instance de *Webpage* soit à la fois la cible et la source d'un lien. Si le concepteur du métamodèle souhaite ajouter une telle contrainte alors il faudra l'exprimer avec la règle OCL décrite ci-dessous.

Contrainte Une *Webpage* ne peut pas être à la fois la cible et la source d'un lien.

```
1 Context Link inv:
2
3   source <> target;
```

Une fois la conception du métamodèle terminée, le concepteur peut modéliser des sites Web en utilisant son métamodèle. Un éditeur par défaut est disponible dans *Eclipse Modeling Framework*. Même si cet éditeur permet de valider la cohérence du métamodèle, il reste néanmoins limité et ne propose pas de représentation graphique pour les modèles, d'où l'interêt d'utiliser la composante *Graphical Modeling Framework* qui permet de créer des modeleurs visuels puissants et plus faciles à utiliser.

2.2.3 Bilan

Nous avons vu que l'adaptation au contexte métier est possible à travers deux principaux mécanismes : les profils UML et les DSML. Les principales différences entre les deux approches sont qu'avec un DSML, le modèle est plus simple à manipuler car il ne dépend pas de tous les concepts UML mais d'un langage de petite taille. Tout modèle est valide par construction car l'utilisateur ne peut pas modéliser quelque chose de non conforme au langage. Plusieurs travaux mettent en évidence l'apport de l'utilisation d'un DSML par rapport à une approche classique [JK09, JLT04, KT08]. Cet apport consiste principalement en un gain en productivité grâce à la simplicité d'utilisation d'un DSML et à la génération automatique du code source et à l'amélioration de la qualité du logiciel en réduisant le nombre de bugs introduits lors du l'implémentation du code source. Toutefois, la définition d'un DSML n'est pas tâche facile et on peut vite tomber dans les travers de cette approche [KP09], *i.e.* langage complexe avec trop de concepts, éditeur difficile à utiliser... Quelques travaux préconisent de bonnes pratiques pour la conception de DSML [KKP+09].

Dans notre cas, nous nous intéresserons surtout à la définition de DSML à travers la métamodélisation.

2.3 Transformation de modèles

La notion de transformation est omniprésente en informatique et elle n'est pas réservée à l'IDM. En effet, la compilation d'un programme C ou C++ est une transformation d'un modèle abstrait (le programme écrit en C/C++) en langage machine compréhensible et optimisé pour la plateforme cible. Sans cette compilation, *i.e.* transformation, le programme en soi n'est pas opérationnel. De la même manière, l'exploitation d'un modèle ne doit pas se limiter à la conception ou la documentation, d'où l'interêt de le rendre opérationnel. L'IDM permet de capitaliser le savoir-faire de modélisation grâce aux transformations de modèles. Ces transformations prennent en entrée un modèle M1 pour produire en sortie un modèle M2. La figure 2.7 illustre le processus de transformation. Si M1 et M2 ont le même métamodèle, alors on parle de transformation endogène

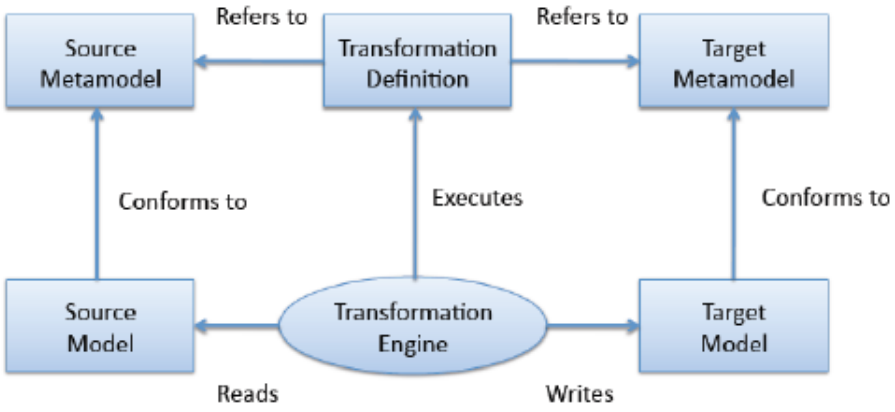


FIGURE 2.7 – Processus de transformation de modèles [CH06]

(exemple : transformer un modèle UML en un autre modèle UML) sinon il s'agit de transformation exogène (exemple : transformer un modèle UML vers du code Java). Dans le cas d'une transformation d'un modèle UML (conforme à un métamodèle source) en code Java (conforme à un métamodèle cible), il s'agit d'une génération de code respectant la syntaxe de la grammaire du langage Java exprimé grâce à l'EBNF (Extended Backus-Naur Form) [ISO96] à partir d'un modèle conforme à un métamodèle décrit par le méta-méta-langage MOF [OMG00].

On parle aussi de transformation M2M (*Model to Model*) et de transformation M2T (*Model to Text*). Les transformations permettent de passer d'un espace technique à un autre (cf. figure 2.1). Ainsi, les transformations permettent de choisir l'espace technique cible et le formalisme le plus adapté à chaque besoin, tout en ayant un espace technologique source unique, *i.e.* les modèles (conformes à un métamodèle).

Il existe plusieurs langages de transformation de modèles, parmi lesquels on peut citer ATL [Ag08], Acceleo [47] qui est une implémentation du standard MTL [Obj08] de l'OMG, JET [48] ou encore QVT [Fav10]. Le choix d'un langage de transformation dépend essentiellement de l'objectif de la transformation.

2.4 Le test et les langages de modélisation

Nous avons évoqué dans les paragraphes précédents la puissance de l'ingénierie dirigée par les modèles et sa capacité à couvrir les différentes phases d'un projet logiciel (conception, génération de code...). Le test ne fait pas exception : en effet l'IDM couvre

47. http ://www.eclipse.org/acceleo/
48. http ://www.eclipse.org/modeling/m2t/ ?project=jet

aussi cette phase.

Le test consiste à détecter des erreurs dans un système, ce qui se traduit par une série d'activités dont le but est d'observer un comportement du système différent de celui attendu. Dans la plupart des cas, le comportement attendu est décrit dans un document de spécification et le système sous test est une implantation qui doit être conforme à cette spécification. Dans le cas de l'ingénierie dirigée par les modèles, la spécification et le système à tester peuvent être tous les deux des modèles. Différentes approches ont vu le jour et permettent d'apporter des solutions aux divers types de test en fonction des objectifs de celui-ci. Néanmoins, le test basé sur les modèles est un domaine relativement récent et ne bénéficie pas encore d'une maturité qui aboutit à la standardisation des diverses approches. Dans cette section nous présentons les principales contributions et nous exposons les différents sujets de discussion.

Un modèle de test, comme un modèle de conception, doit être conforme à un langage de modélisation. Le choix de ce langage dépend des besoins du testeur et des objectifs du test lui-même.

2.4.1 Langage de modélisation vs. langage spécifique de test

La modélisation de cas de test peut se baser soit sur :
- un langage de modélisation classique tel que UML ;
- un langage de modélisation des tests tel que *UML Testing Profile* (UTP) [BDG+08] ;
- des langages spécifique propres au test dans un domaine bien précis.

2.4.1.1 Langage de modélisation

L'utilisation de standards tels que UML ou SysML [FMS08] pour la modélisation de cas de test a un certain nombre d'avantages. Plusieurs outils supportent UML d'autant plus qu'il est devenu un standard dans le monde académique et industriel. Ceci permet donc de trouver relativement facilement des personnes qualifiées à la fois pour modéliser les systèmes et les tests associés. Un autre avantage de l'utilisation d'un langage tel que UML réside dans la facilité de communication entre les testeurs et les développeurs qui seront capables les uns comme les autres de comprendre les modèles de chacun. Ceci est important afin de répondre aux exigences et aux fonctionnalités du système à développer et tester.

Quant à l'inconvénient d'UML, il réside dans l'effort nécessaire pour adapter ce langage générique aux besoins du testeur et à ceux du développeur. D'après [HKO06], il existe trois principales différences entre les besoins de modélisation du point de vue développeur et du point de vue testeur :
- les testeurs n'ont pas besoin de modéliser des détails relatifs à la plateforme cible alors que le développeur est obligé de le faire ;
- les modèles de test sont orientés « expérience utilisateur » alors que les modèles du développement décrivent plutôt le comportement interne détaillé du système à

tester ;
- les testeurs et les développeurs n'ont pas les mêmes compétences techniques. Le développeur maîtrise mieux les langages de modélisation alors que le testeur a plus une vue orientée « utilisateur » et « gestion des exigences ».

Plusieurs travaux ont adopté UML pour la modélisation de test. Parmi ces travaux, nous citons [Pac05] qui a mis en œuvre une solution pour le test de robustesse ou encore [PdKB+09] pour le test de web services.

Dans [LBN09, PBL08, FMW+10] les auteur ont présenté des études de cas industriels réussis où l'utilisation d'UML pour la modélisation des cas de tests a fait ses preuvess.

2.4.1.2 Langages spécifiques de test

L'utilisation d'un langage spécifique à la définition de cas de test peut considérablement faciliter la mise en place d'une approche dirigée par les modèles pour le test. En effet, les testeurs ne sont pas obligés de maîtriser des langages génériques complexes tel que UML pour finalement n'en utiliser qu'une partie.

Il existe plusieurs langages spécifiques pour la modélisation de test. Ces langages sont uniquement utilisables pour le test et souvent pour le test dans un domaine précis à l'exception de UTP [BDG+08] qui lui est générique. *UML Testing Profile*, introduit en 2005 par l'OMG et dont le but est d'enrichir/d'adapter UML aux spécificités du test, fournit des concepts qui visent la modélisation des spécifications du test. En particulier, il contient les concepts couvrant l'architecture de test, le comportement de test, les données de test et la durée du test. Ces concepts définissent un langage de modélisation pour visualiser, spécifier, analyser, construire et documenter les artefacts d'un système en test. Baker et al. [BDG+08] décrit les processus de test basé sur les modèles UML. Ce dernier est utilisé pour spécifier les modèles du système sous test et le profil UTP est utilisé pour décrire l'environnement du système sous test. Nous verrons dans le chapitre 4 que notre approche est relativement différente de UTP étant donné que ce dernier vise à modéliser le comportement du système en test alors que notre démarche vise à modéliser le test dans le but de piloter le système à tester.

Ces langages spécifiques au test permettent à l'utilisateur de bénéficier des mêmes avantages que les langages spécifiques apportent au niveau de la conception, à savoir : l'utilisation de concepts métier haut niveau, la génération automatique du code à partir des modèles, etc. Les travaux dans ce domaine sont nombreux. Prenons à titre d'exemple l'approche décrite dans [JKK+09] ou [FLFL09] pour le test d'applications et de sites mobiles. Les auteurs proposent un DSML pour modéliser le comportement de l'utilisateur du téléphone à un niveau élevé d'abstraction. Ces travaux se basent sur le paradigme de mots clés (*Keyword-based Testing*) pour générer le modèle de test sous forme de machine à état. Ce travail nous a particulièrement interessé vu qu'il s'approche de notre domaine de recherche. En effet MATeL [RBBC10], le DSML que nous présentons dans ce manuscrit est un langage spécifique au test d'applications mobiles. Cependant nos travaux sont

relativement différents et celà à différents niveaux :

- nous ne ciblons pas une plateforme mobile donnée mais nous nous intéressons à toutes les plateformes ;
- nous utilisons différentes plateformes d'exécution de test avec des contraintes différentes ;
- nous suivons différentes approches pour modéliser les cas de test.

Dans le cadre de nos travaux, nous avons opté pour un langage spécifique pour le test. Ce choix est principalement justifié par un besoin élevé de concrétisation de test sur un banc de test en ligne. Nous reviendrons sur ce choix dans le chapitre 4.

2.4.2 *Model-Based Testing* vs. *Model-Driven Testing*

Certainement à cause de la non maturité des approches *Model-* Testing*, très peu de travaux définissent clairement la différence entre l'approche *Model-Based Testing* et l'approche *Model-Driven Testing*. D'ailleurs souvent on utilise les deux terminaisons pour évoquer les mêmes concepts. Dans ce paragraphe, nous présentons ces deux approches ainsi que notre point de vue sur la différence entre les deux.

2.4.2.1 *Model-Based Testing*

La définition de l'approche *Model-Based Testing* est quasiment la même dans tous les travaux. On parle souvent de *Model-Based Testing* ou de MBT comme étant une solution de génération de cas de test à partir de modèles. Ces derniers décrivent le comportement attendu du système sous test ou *System Under Test* (SUT) en anglais. D'après [UL07], un processus MBT se découpe en cinq étapes (cf. figure 2.8) :

1. la modélisation - définition d'un modèle abstrait pour le test du système ;
2. la production - génération de cas de test à partir du modèle abstrait ;
3. la concrétisation - transformation des cas de tests abstraits en cas de tests exécutables sur le système en test ;
4. l'exécution - lancement des tests sur le système ;
5. l'analyse - étude des résultats obtenus.

Il existe plusieurs solutions autour du *Model-Based Testing*. Ces solutions se distinguent principalement par les outils mis en place pour chacune des cinq phases énumérées plut haut. Dans [UPL06], les auteurs offrent un aperçu des différentes approches MBT. A titre d'exemple, pour la phase « exécution », on distingue deux types de génération de tests. Cette distinction est basée sur l'intervalle de temps entre la génération de tests et leur exécution. Dans le cas où l'outil de MBT se connecte automatiquement au système à tester (SUT) pour exécuter les tests dynamiquement, on parle de *On-Line Testing*. Ceci signifie que le test est construit dynamiquement en fonction des réponses du SUT. Ce type de test est particulièrement adapté aux systèmes non-déterministes.

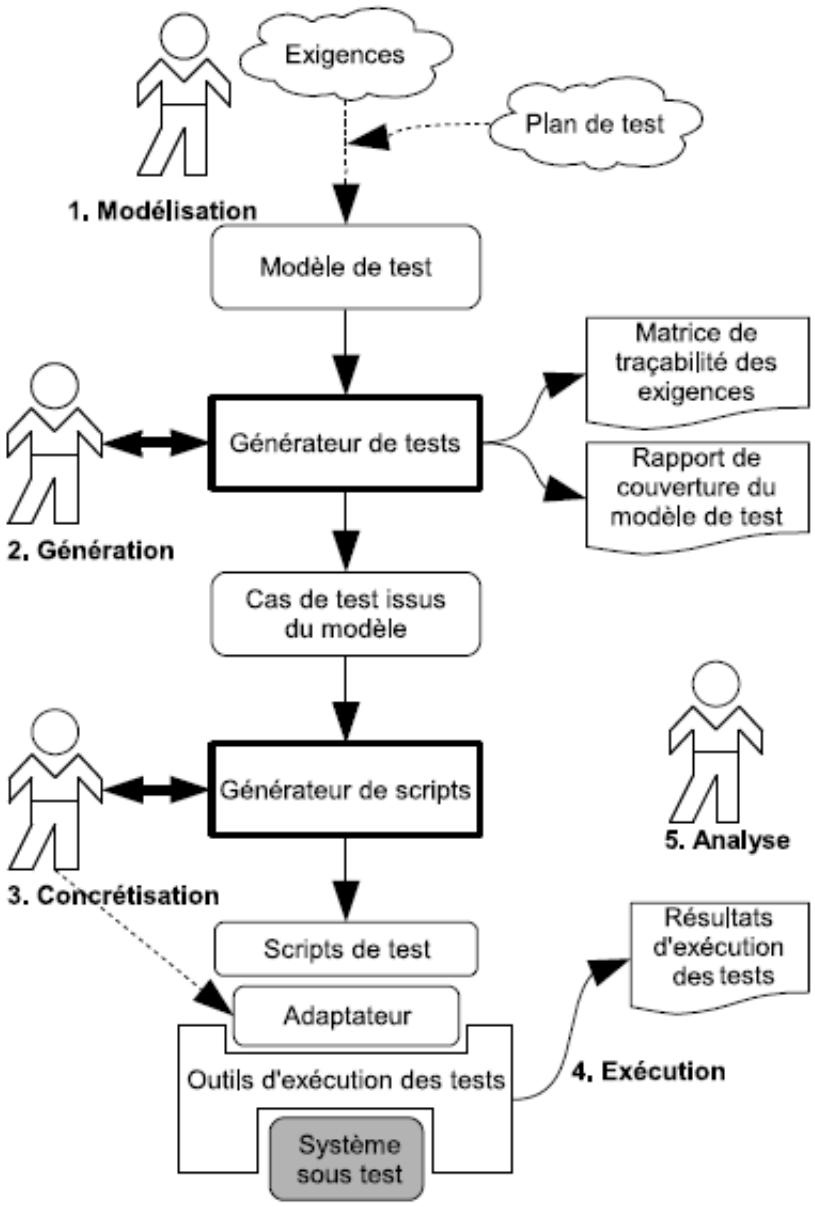


FIGURE 2.8 – Les différentes étapes dans un processus Model-Based Testing

Par opposition, on trouve le *Off-Line Testing* où l'on génère le test indépendamment du SUT. Il existe deux types de tests :

– tests exécutables automatiquement - l'outil de MBT génère des programmes ou scripts de tests exécutables, c'est à la charge du testeur de les déployer. Il s'agit

souvent de programmes écrits en langage de script tels que Perl ou Python ;
– tests à deployer manuellement - l'outil de MBT génère un document (sous forme de texte ou tableau) qui contient les étapes du test à produire sur le système. C'est donc à la charge du testeur de les produire sur le système sous test.

2.4.2.2 *Model-Driven Testing*

Souvent la terminologie *Model-Driven Testing* ou MDT est utilisée pour parler de *Model-Based Testing*. Dans [HKO06], les auteurs font l'analogie avec l'approche MDA [Sol00] définit par l'OMG. D'après eux, le terme *Model-Driven Testing* se réfère à un style particulier de tests basés sur les modèles, inspirés de l'approche MDA et des principes sous-jacents de celle-ci telles que la séparation entre la plateforme spécifique (PSM) et la plateforme indépendante (PIM) ou encore l'utilisation des transformations automatiques pour passer entre ces différents niveaux d'abstraction. Par analogie, le MDT est donc la conception de modèle de test indépendant de la plateforme de test. Grâce à des transformations, ce test indépendant devient spécifique à la plateforme technologique ciblée.

Afin de clarifier notre point de vue, nous proposons, ci-dessous, des définitions pour les approches MBT et MDT.

MBT : nous avons décidé de garder la définition répandue à savoir, une approche de génération de cas de test à partir d'un modèle. Ce dernier décrit le comportement attendu du système sous test. Un processus MBT doit contenir toutes les étapes décrites dans la figure 2.8.

MDT : nous utilisons cette terminaison pour désigner toute approche basée sur l'utilisation de modèle décrivant le processus de manipulation du système sous test (actions à effectuer par l'utilisateur).

En nous basant sur ces deux définitions, nous positionnons nos travaux [RBBC10] dans le cadre d'une approche MDT.

2.5 Synthèse

Nous avons vu que l'IDM est un domaine technologique et scientifique très large qui permet de couvrir tout le cycle de développement logiciel [JGB06]. Dans ce chapitre, nous nous sommes principalement intéressés aux avantages de l'adoption d'une approche dirigée par les modèles (abstraction par la métamodélisation, transformations, différents outils open source...). Nous avons présenté la possibilité de définir de nouveaux métamodèles afin de proposer des langages de modélisation spécifiques à un domaine (DSML). Ces langages ont l'avantage de mettre à la disposition de l'utilisateur les concepts et les notions du système à modéliser. Ceci améliore la lisibilité des modèles, les rend accessibles pour des utilisateurs non spécialistes [KT08] et surtout capitalise le savoir-faire au

niveau de la modélisation. Le monde de l'IDM étant de plus en plus adopté, de nombreux outils ont été développés. Le plus répandu, Eclipse, dispose du projet *Eclipse Modeling Project*[49] et inclut un nombre important de composants open source couvrant la métamodélisation (syntaxe abstraite), la génération de code ou encore la génération d'éditeur graphique (syntaxe concrète).

Nous nous positionnons dans une approche *Model-Driven Testing* avec deux apports majeurs par rapport aux approches classiques. Ces apports résident dans la concrétisation des cas de test abstraits en cas de tests réels exécutables et dans la gestion de la variabilité entre différentes plateformes d'exécution. Pour nos travaux, nous souhaitons bénéficier des avantages offerts par l'IDM et par les outils qui y sont associés pour proposer un métamodèle *i.e.* un langage de modélisation qui permettra la définition de scénarios de test propres au test d'applications mobiles. Le chapitre suivant présente la notion de lignes de produits logiciels que nous avons fortement intègrée dans notre métamodèle. Nous montrons que, paradoxalement, la problématique de la fragmentation mobile touche aussi, mais à une autre échelle, les fabricants de téléphones mobiles. En effet, ces fabricants sont obligés de sortir assez régulièrement des modèles différents au niveau logiciel et matériel afin de répondre aux besoins du marché. Ce qui implique des adaptations du code source des systèmes d'exploitation pour chaque modèle.

49. http ://www.eclipse.org/modeling/

Chapitre 3

Les lignes de produits logiciels

Sommaire

3.1 Introduction 52
3.2 Définitions 52
3.3 L'ingénierie de domaine et d'application 53
3.3.1 Ingénierie de domaine 53
3.3.2 Ingénierie d'application 54
3.4 Les contraintes de dépendance 54
3.5 Expression de la variabilité 55
3.5.1 Au niveau du code 55
3.5.2 Au niveau des modèles 57
3.6 Mise en œuvre dans l'industrie 58
3.6.1 L'exemple Renault 58
3.6.2 L'exemple Nokia 59
3.7 Le test dans une ligne de produits logiciels 59
3.8 Synthèse 61

A software product line (SPL) is a a set of software intensive systems sharing a common, managed set of features which satisfy the specific needs of a particular market segment or mission and which are developed from a common set of core assets in a prescribed way. [CN02]

3.1 Introduction

Le principe de l'approche « lignes de produits logiciels - LdP » a vu le jour pour éviter le gaspillage de temps et d'argent en développant le même logiciel n fois. Effectivement, l'objectif des LdP est de minimiser les coûts de construction de logiciels dans un domaine d'application particulier en ne développant plus chaque logiciel séparément, mais plutôt en le concevant à partir d'éléments réutilisables. Ceci permet d'augmenter les gains grâce à une productivité à grande échelle, minimiser le temps de mise sur le marché des logiciels, améliorer la qualité des logiciels et réduire les coûts de production. Prenons à titre d'exemple la fabrication de photocopieurs. Ces derniers sont fabriqués à partir d'un ensemble de fonctionnalités communes (imprimer, communiquer avec un ordinateur, etc.) mais peuvent comporter certaines caractéristiques qui les différencient (fonctionnalités supplémentaires : fax, scanner, type de connexion réseau, etc.). Dans le monde logiciel, les différences peuvent apparaître en fonction de choix techniques (utilisation d'un type particulier de matériel), commerciaux (création d'une version limitée) ou encore régionaux (produits destinés à plusieurs pays).

3.2 Définitions

Même si les approches de mise en œuvre d'une LdP et les domaines de leur application varient, souvent les mêmes définitions sont utilisées dans la littérature [MMYJ10]. Nous donnons ici les définitions des principaux concepts des LdP.

Une ligne de produits logiciels est un ensemble de systèmes partageant un ensemble de propriétés communes et satisfaisant des besoins spécifiques pour un domaine particulier.

Un domaine est un secteur de métier ou de technologies ou des connaissances caractérisé par un ensemble de concepts et de terminologies compréhensibles par les utilisateurs de ce secteur.

La variabilité regroupe l'ensemble des hypothèses montrant comment les produits membres de la ligne de produits diffèrent.

La commonalité regroupe l'ensemble des hypothèses qui sont vraies pour tous les produits appartenant à la même ligne de produits.

Un *asset* [50] est un élément qui permet de développer un logiciel (un document de spécification, modèle, code, etc.) qui sera réutilisé pour la construction de produits dans la même LdP.

Un point de variation identifie un ou plusieurs emplacements auxquels la variation peut se produire. Un point de variation peut être vu comme un point de décision avec plusieurs choix possibles appelés **variants**.

La première difficulté liée à l'approche LdP réside dans la conception d'une architecture permettant de définir plusieurs produits. Il s'agit d'une architecture générique pour tous les produits de la même LdP. Les membres d'une ligne de produits sont caractérisés par leurs points communs, mais aussi par leurs variabilités. La gestion de cette variabilité est l'une des activités clés des lignes de produits. Une autre difficulté de l'utilisation d'une ligne de produits concerne la construction d'un produit logiciel (on parle aussi de dérivation de produit) qui consiste à faire certains choix vis-à-vis de la variabilité définie dans la ligne de produits. Certains choix sont incompatibles entre eux. De la même manière, un choix particulier lors de la dérivation d'un logiciel peut exclure certaines variantes. Une ligne de produits doit donc aussi intégrer des contraintes de cohérence permettant de faciliter les choix lors de la dérivation.

Le but de ce chapitre est de présenter l'approche LdP, ses différents concepts et enfin le test logiciel dans le cas d'une telle approche.

3.3 L'ingénierie de domaine et d'application

Souvent dans la littérature [PBvdL05], l'ingénierie des lignes de produits logiciels se découpe en deux niveaux. La figure 3.1 illustre ce découpage : « ingénierie de domaine » et « ingénierie d'application ».

3.3.1 Ingénierie de domaine

L'efficacité d'une approche basée sur une ligne de produits logiciels dépend directement de la façon dont la variabilité est gérée à partir de l'analyse et jusqu'à l'implémentation et la maintenance d'une LdP. Les points communs, ainsi que la souplesse nécessaire pour s'adapter aux exigences de produits différents, construisent les *assets*. Ces derniers sont créés lors de l'ingénierie de domaine. Effectivement, le but de l'analyse du domaine est d'étudier le domaine de la ligne de produits et d'identifier les commonalités et les variabilités entre les produits. Il existe plusieurs méthodes pour l'analyse de domaine, la plus connue est FODA (*Feature Oriented Domain Analysis Expressions*) [KCH+90] qui décrit le domaine dans un modèle de caractéristiques (une caractéristique est appelée *feature*). Dans la figure 3.2 apparaît un *Feature Model* réalisé pour la fabrication

50. Nous n'avons trouvé aucune traduction officielle en français. Nous proposons l'expression « capital logiciel ».

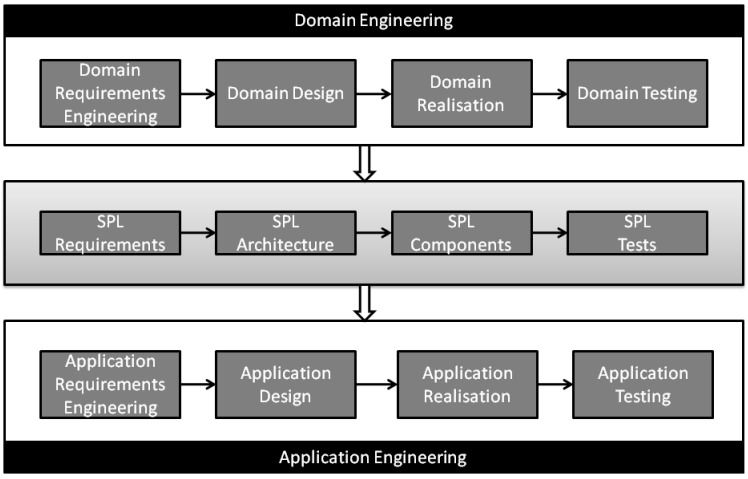


FIGURE 3.1 – L'ingénierie de lignes de produits logiciels

d'un téléphone mobile. Nous pouvons, à titre d'exemple, constater que la fonctionnalité « Auto Focus » est optionnelle alors que la résolution est un point de variabilité dont les variants sont 2MP, 5MP, 8MP ou encore 12 MP.

Une LdP reflète l'expérience acquise par la création de nombreuses applications qui partagent un ensemble de caractéristiques communes, généralement parce qu'elles appartiennent au même domaine. Cet ensemble de caractéristiques communes peut être paramétré et peut être factorisé pour représenter la variabilité dans le domaine. Ceci permet de développer et construire les *assets*, éléments qui permettent de développer un logiciel (document de spécification, modèle, code, etc.) qui seront ensuite réutilisés pour la construction de produits dans chaque nouvelle demande dans le même domaine.

3.3.2 Ingénierie d'application

L'ingénierie d'application consiste à utiliser les résultats de l'ingénierie de domaine pour la construction (dérivation) d'un produit particulier. Les résultats de l'ingénierie de domaine contiennent de la variabilité. La dérivation d'un produit particulier a donc besoin de décisions (ou de choix) associées à ces points de variation. A titre d'exemple, choisir la résolution « 5MP » pour l'appareil photo d'un mobile dans le *Feature Model* de la figure 3.2 constitue une décision.

3.4 Les contraintes de dépendance

Les lignes de produits logiciels sont caractérisées par des contraintes de dépendance entre les points de variation. En effet, la résolution d'un point de variation peut influencer la résolution d'autres points de variation. FODA [KCH+90] permet de décrire deux types

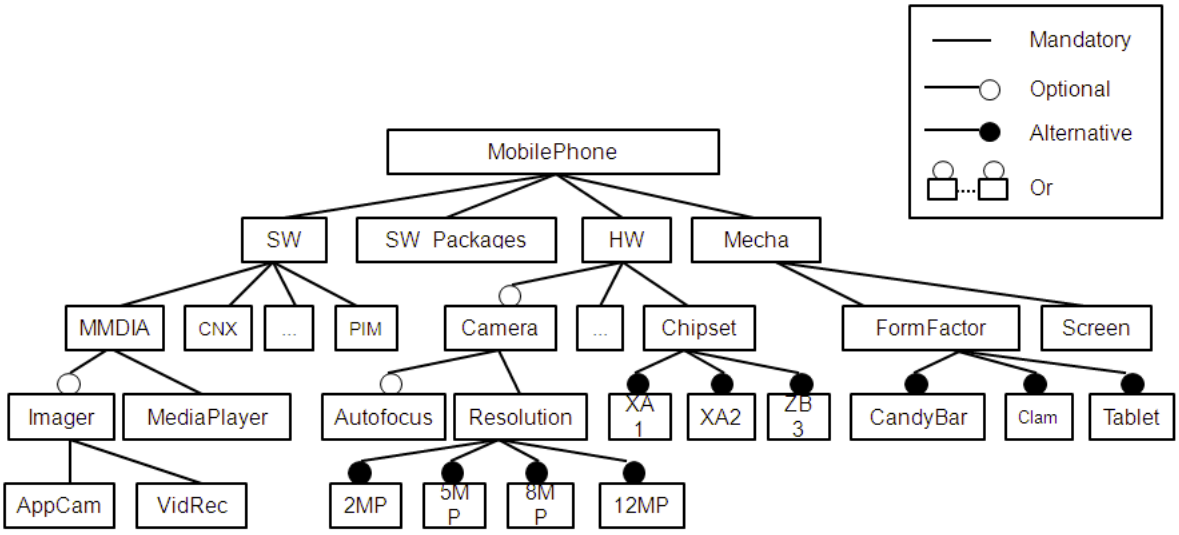


FIGURE 3.2 – Exemple d'un *Feature Model* pour la fabrication d'un téléphone mobile [LH10]

de règles : l'une de présence, l'autre d'exclusion. La règle de présence spécifie que le choix d'une caractéristique optionnelle ou variante exige la présence d'une autre caractéristique optionnelle ou variante (cf. exemple du GPS dans 3.6.1). La règle d'exclusion entre deux caractéristiques spécifie qu'elles ne peuvent pas être présentes dans le même produit (cf. exemple de l'essuie glace dans 3.6.1). Le langage OCL (*Object Constraint Language*) peut être utilisé pour exprimer de telles contraintes [Zia04, AG10].

3.5 Expression de la variabilité

La commonalité et la variabilité sont les concepts centraux dans les lignes de produits logiciels. La gestion de la variabilité est l'activité la plus compliquée à mettre en œuvre. Elle concerne l'identification, la conception et la mise en pratique des points de variabilité. La gestion de la variabilité des *feature* a souvent un large impact sur tout le cycle de vie d'un produit dans une ligne de produits logiciels. Dans [SB00, Bos00, Har01], les auteurs citent quelques techniques pour la gestion de la variabilité au niveau du code source mais aussi au niveau des modèles.

3.5.1 Au niveau du code

Plusieurs techniques permettent de gérer la variabilité au niveau du code source d'une application. Parmi ces techniques on peut retenir les techniques de compilation qui permettent la dérivation d'un produit pendant la phase de compilation. La com-

pilation conditionnelle et le chargement dynamique de bibliothèques sont des exemples de ces techniques. Elles sont utiles si la variabilité concerne les parties de code et les bibliothèques à inclure ou à exclure. Dans le paragraphe 1.1.1 du chapitre 1, nous avons montré un exemple de code Java où cette technique est utilisée pour inclure ou exclure des portions de code en fonction de variabilité liée à des propriétés matérielles de la plateforme du système sous test, en l'occurence sa taille d'écran.

Une autre technique est liée aux langages de programmation à objets qui apportent quelques mécanismes pour implémenter la variabilité. Parmi ces mécanismes, nous citons l'abstraction à travers la notion d'héritage et la redéfinition de fonctions associées au polymorphisme. Les points de variation peuvent être définis comme abstraits et redéfinis par chaque variant d'une manière spécifique. La figure 3.3 illustre un exemple de mise en œuvre d'une telle approche à travers un héritage entre classes.

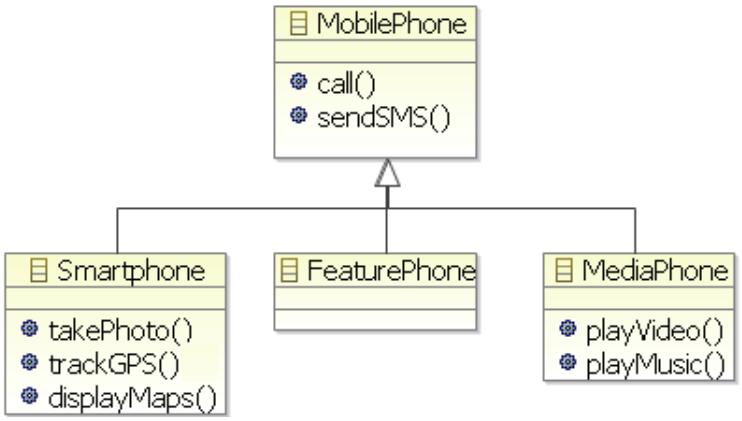


FIGURE 3.3 – Exemple de mise en œuvre de l'héritage

D'autres moyens plus récents de gestion de la variabilité existent. Par exemple, depuis la version 5.0 de Java, la notion d'annotation [51] a été introduite. Il s'agit d'une sorte de commentaires ou méta-données qu'on peut insérer dans le code Java. Certaines de ces annotations sont à la destination du compilateur Java, d'autres sont exploitables directement à l'exécution du programme grâce à la réflexivité.

Même si ces mécanismes se basent sur des technologies et des techniques qui ont connu une réussite remarquable dans le domaine du génie logiciel, la maîtrise du code devient de plus en plus difficile à gérer avec la croissance exponentielle de marqueurs de variabilité dans le code (syndrome des macro instructions C et C++ comme ♯*pragma* par exemple).

51. http ://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

3.5.2 Au niveau des modèles

La modélisation de la variabilité est destinée à capter les éléments nécessaires pour mettre en evidence la façon dont un produit est semblable, mais aussi différent d'un autre. Les produits qui sont suffisamment semblables sont souvent définis pour former une ligne de produits. La modélisation des lignes de produits est différente de la modélisation d'un produit singulier. Un modèle de ligne de produits doit contenir des informations sur les variantes de produits et leurs dépendances. En outre, la modélisation d'une ligne de produits implique des compétences autres que celles d'un spécialiste du génie logiciel. Typiquement, des experts du domaine prennent part à la modélisation des lignes de produits et également à la configuration des modèles de produit comme élément singulier. La compétence des experts du domaine sur une des lignes de produits (ou plus) peut être synthétisée dans un vocabulaire spécifique *Domain-Specific Modeling Language*. Comme nous l'avons expliqué dans le chapitre 2, un DSML contient des concepts spécifiques à un domaine, ce qui implique souvent une spécification plus efficace.

La gestion de la variabilité dans un langage de modélisation a été étudiée dans plusieurs travaux tels que [TZJ03, CJ01, ACL+11, ZJF03, PDH+10] qui proposent une approche basée sur UML pour la gestion de la variabilité dans la conception d'une application. Les auteurs ont formalisé un ensemble d'extensions sous forme de différents profils UML pour la modélisation de la variabilité dans les LdP.

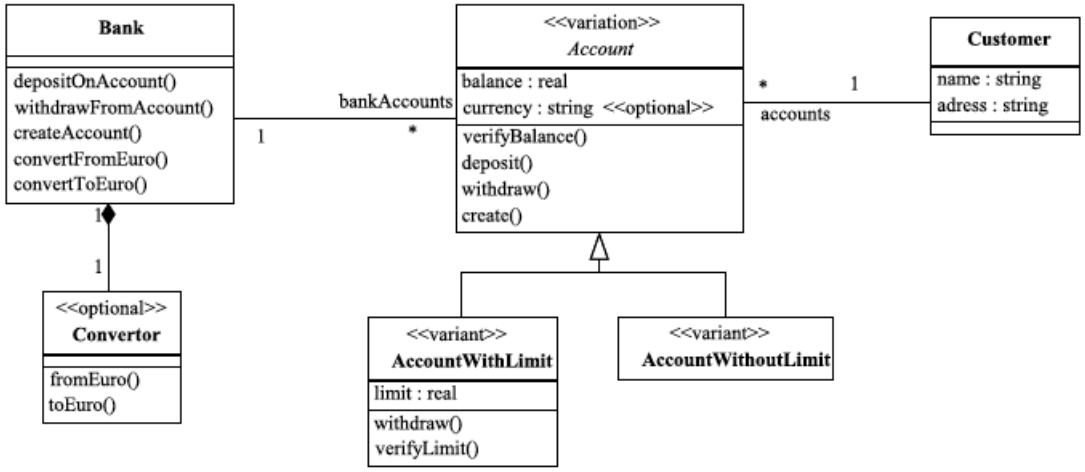


FIGURE 3.4 – Diagramme de classe d'une LdP basée sur l'utilisation d'un profil UML [TZJ03]

La figure 3.4 illustre la mise en œuvre d'une LdP en utilisant un diagramme de classe auquel des extensions ont été introduites sous forme de stéréotypes UML (« optional » pour spécifier l'optionnalité des éléments d'un diagramme de classes, « variation » et

« variant » pour spécifier l'alternative dans le choix de classes).

Grâce à une phase de dérivation, le concepteur de la LdP peut générer tous les produits possibles. Cette dérivation est souvent un ensemble de transformations appliquées au modèle contenant la variabilité. Ces transformations permettent de dériver tous les produits d'une LdP.

3.6 Mise en œuvre dans l'industrie

Motivée par la diversité des facteurs de variation des logiciels dans certains domaines, l'approche ligne de produits a été adoptée depuis sa naissance dans l'industrie. Dans [MMYJ10,PBvdL05,CN02], les auteurs exposent plusieurs expériences industrielles prouvant sa réussite. De la même manière, le site officiel de la conférence dédiée au LdP (Software Product Line Conferences - SPLC), contient une page [52] « hall of fame » qui liste quelques réussites de mise en œuvre de LdP dans divers domaines tels que l'aéronautique (Boeing [53]), les systèmes médicaux (Philips [54]) ou encore les télécommunications (Lucent [55]).

3.6.1 L'exemple Renault

Lors de la journée lignes de produits « Maîtriser la diversité [56] » organisée par l'université de Paris 1 en octobre 2010, une équipe du constructeur automobile Renault [57] a présenté la ligne de produits mise en place pour la spécification et la configuration de la ligne de produits véhicule de Renault. Plusieurs exemples ont été montrés pour illustrer la présence forte de contraintes de dépendance lors de la configuration d'une voiture. En effet, toutes les combinaisons ne sont pas possibles. Il y a des contraintes entre les options. A titre d'exemple, sur un utilitaire, on ne peut pas mettre un essuie-glace arrière sur une porte arrière tôlée ou encore l'option GPS implique forcément un autoradio avec lecteur CD (pour la mise à jour des cartes).

Il existe, par exemple, 10^{21} combinaisons possibles pour le modèle « Renault Traffic ». Quand une commande est prise, il faut fabriquer la voiture ; il n'est plus temps de se demander comment on va la fabriquer, avec quelles pièces. Vu de l'usine, à tout moment, il faut être capable de fabriquer n'importe lequel de ces 10^{21} véhicules virtuels.

52. http ://www.splc.net/fame.html
53. http ://www.boeing.com/
54. http ://www.healthcare.philips.com/
55. http ://www.alcatel-lucent.com/
56. http ://sites.google.com/site/journeespl/
57. http ://www.renault.fr/

3.6.2 L'exemple Nokia

La société Nokia a été parmi les premiers fabricants de téléphones mobiles à adopter une approche ligne de produits pour gérer la diversité des logiciels des téléphones mobiles [MT00, Bos05]. Souvent la terminaison « famille de produits » ou encore « family products » est utilisée pour désigner une LdP. Nokia a mis en place une stratégie qui lui permet de répondre à la fois aux besoins commerciaux et aux besoins techniques. D'après Juha Savolainen (Principal Member of Research Staff), le défi est d'offrir la personnalisation des produits et services pour des clients individuels à un prix de production de masse [MS10].

Chaque année, Nokia introduit sur le marché différents modèles de mobiles ; ceci rend leur production à partir de zéro très difficile. Ces téléphones sont équipés de l'un des deux systèmes d'exploitation adoptés par Nokia : Symbian OS (avec ses différentes versions) et récemment Windows Phone. De plus, Nokia doit aussi répondre à plusieurs facteurs de variation entre ses produits tel que la langue de l'interface utilisateur, chaque langue ayant ses propres particularités (sens d'écriture, connexion des lettres...). Les mobiles Nokia doivent aussi être compatibles avec les différents standard de communication et les différents réseaux de télécommunication partenaires (cf. paragraphe 1.2.1 du chapitre 1).

Dans [LH10], l'auteur a considéré le marché des systèmes d'exploitation des *smartphones* comme exemple pour expliquer comment certaines entreprises (telle que Apple) ont réussi à contourner la problématique de la variabilité en l'externalisant. Effectivement, ce sont les développeurs tiers qui développent des fonctionnalités supplémentaires (applications et jeux) alors que les équipes de développement d'Apple se concentrent sur les fonctions essentielles. Ceci a l'avantage de réduire les coûts et d'accélerer le « time-to-market ». Selon [Bos09], l'externalisation de la gestion de la variabilité en dehors d'une structure LdP transforme la LdP en un « écosystème logiciel ».

3.7 Le test dans une ligne de produits logiciels

L'utilisation des lignes de produits logiciels vise à atteindre un certain nombre d'objectifs, notamment la réduction des coûts, la réduction du temps de mise sur le marché, l'amélioration de la qualité des produits appartenant à la ligne de produits, etc. Ces objectifs ne seront atteints que si un ensemble complet d'activités de test et de validation est mis en place.

Dans [LUV09b], les auteurs présentent une analyse des principales approches proposées pour le test dans le cadre d'une LdP. Ces approches couvrent différents types de test : tests unitaires, tests d'intégration, tests fonctionnels... Le plus grand nombre des travaux se sont intéressés aux tests fonctionnels en proposant des techniques de dérivation des cas d'utilisation à partir de la LdP. Cette dérivation peut être automatisée ou manuelle. A titre d'exemple, dans [NFTJ03], les auteurs proposent d'enrichir les dia-

grammes de cas d'utilisation UML avec des valeurs marquées et des pre/post conditions afin d'exprimer les contraintes de dépendances dans une LdP. D'autres approches telle que celle présentée dans [PSK+10] propose l'utilisation des *feature diagram* pour la génération automatique de cas de tests.

Dans [McG07], l'auteur présente un processus complet pour le test dans le cadre de ligne de produits. Ce processus contient les mêmes types de tests classiques utilisés dans un contexte hors LdP. Ce processus est basé sur le test des *assets*, le test des produits dérivés et enfin le test des interactions entre les deux. Cette approche est basée sur la distinction des rôles pour chaque phase de test. L'auteur présente de bonnes pratiques et préconise par exemple que l'équipe qui a développé un *asset* donné doit être responsable du test de tout composant incluant cet *asset* ou une variante de celui-ci. L'auteur met en place une relation de dérivation entre les produits et les tests ; exemple : si un produit X dérive d'un produit Y alors les tests de ce dernier peuvent être dérivés de ceux de X (cf. figure 3.5). Le grand nombre de variations définies dans l'architecture de la ligne de produits implique un grand nombre de produits à tester. Charge au concepteur du test de réduire le nombre de cas de tests nécessaires pour assurer la couverture adéquate.

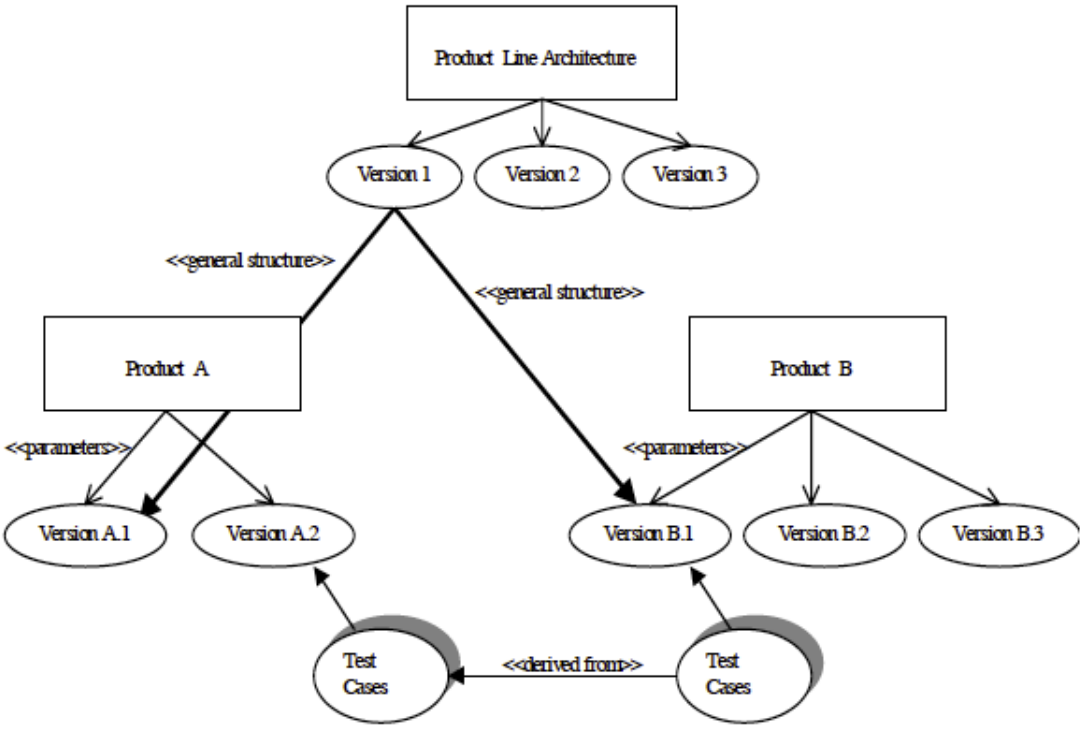


FIGURE 3.5 – Relation entre produits et tests [McG07]

Certains travaux comme [LUV09a] ont proposé d'enrichir *UML Testing Profile* afin

de pouvoir exprimer la variabilité dans le test. Ceci en étendant le métamodèle UTP. Cette approche est relativement similaire à celle présentée par [Zia04] étant donné qu'elle consiste à ajouter des stéréotypes spécifiques tels que « variant » et « variationPoint » pour exprimer les points de variation ainsi que les choix associés. Les auteurs proposent d'utiliser les diagrammes de séquence UML afin de modéliser les interactions du système sous test.

3.8 Synthèse

Dans ce chapitre, nous avons présenté les concepts de l'approche ligne de produits logiciels. Cette approche se base sur trois activités : la modélisation de la variabilité, la gestion des contraintes et la dérivation des produits. Nous avons vu aussi la difficulté liée au test dans le cadre d'une LdP à cause du nombre important de produits possibles (pour n variants, le nombre de produits est de 2^n) [58].

Nous avons vu dans le chapitre 1 les difficultés liées au développement d'applications mobiles, notamment à cause de la fragmentation. Des solutions telles que NeoMAD [59] ou encore [Vir05] ont adopté des approches qui peuvent être assimilées à des LdP au niveau code source en utilisant des techniques de compilation. D'autres travaux comme [CM05, WS08] proposent des solutions différentes basées sur la mise en place de LdP en suivant une approche dirigée par les modèles. Enfin, dans [You05, VG07, CCSC07] les auteurs proposent une démarche basée sur la programmation par aspect. Les applications mobiles produites avec ces outils doivent être testées afin de valider leur bon fonctionnement sur les téléphones cibles. Ces applications font partie d'une ligne de produits logiciels qui partagent un ensemble commun d'*asset* techniques, avec des extensions et des variations pour s'adapter aux téléphones mobiles, aux réseaux mobiles et éventuellement pour répondre aux besoins de clients spécifiques ou des segments de marché. Dans le chapitre suivant, nous présentons une approche basée sur l'utilisation d'un *Domain-Specific Modeling Language* (DSML) pour la description de scénarios de test (plans de test) dans lesquels le concepteur du test est capable d'exprimer des points de variabilité selons plusieurs critères et d'y associer différentes variants. Grâce à un processus de dérivation automatisé, des scénarios dérivés seront générés pour chaque association produit/mobile cible.

58. En pratique, le nombre de produits est sensiblement plus faible à cause des règles d'exclusion entre *feature*

59. www.neomades.com

Troisième partie

MATeL : un DSML pour la définition de scénarios de test d'applications mobiles

Chapitre 4

MATeL

Sommaire

4.1 Introduction 66
4.1.1 MATeL : un langage spécifique 66
4.1.2 Objectifs de MATeL 67
4.2 Le métamodèle MATeL 67
4.2.1 Noyau du métamodèle 68
4.2.2 Gestion des interruptions 76
4.2.3 Gestion des résultats 80
4.2.4 Gestion de la variabilité 81
4.3 Conception de l'éditeur MATeL 87
4.4 Génération automatique de scénarios MATeL 88
4.5 Synthèse 93

4.1 Introduction

Nous avons vu dans les chapitres précédents les difficultés liées au test d'applications mobiles, notamment à cause de la fragmentation. Ce syndrome oblige les développeurs à tester leurs applications sur chacun des téléphones à cibler.

Nous souhaitons apporter une solution d'automatisation à ces tests en proposant d'enrichir l'approche *Cloud testing* (cf. chapitre 1) par un *Domain-Specific Modeling Language* (DSML) propre au test d'applications mobiles. Dans la partie précédente, nous avons vu l'importance de l'ingénierie dirigée par les modèles (IDM) pour mettre en pratique de telles approches. L'IDM permet, en effet, de décrire des langages de modélisation spécifiques et d'automatiser le traitement des modèles qui en sont issus. Ce chapitre regroupe l'ensemble des choix que nous avons réalisés et leurs implications pour atteindre nos objectifs.

4.1.1 MATeL : un langage spécifique

Nous présentons dans ce chapitre MATeL (*Mobile Application Testing Language*) [RBBC10], un langage de modélisation pour la description de scénarios de test. Ce langage permet de modéliser tout ce qu'un utilisateur est capable de faire avec un téléphone entre les mains. Lors de la conception de MATeL, nous avons décidé de définir un DSML qui permet d'écrire des tests avec une orientation « utilisateur », *i.e.* le test est une suite d'actions que l'utilisateur doit réaliser sur son téléphone. La syntaxe concrète de notre DSML est inspirée de celle des diagrammes de séquences UML [UML09] mais nous ne basons pas sur le métamodèle de ce type de diagramme. Effectivement, MATeL est un langage spécifique avec un métamodèle qui lui est propre. Dans nos travaux, nous avons opté pour un DSML parce que nous pensons qu'étant un langage de modélisation générique, UML s'éloigne des préoccupations métier des scénarios de test à modéliser. La sémantique que l'on veut définir s'éloigne de celle exprimée dans UML. De ce fait, nous avons évité la profusion des profils et stéréotypes (cf. chapitre 2) et privilégié un DSML. En effet, le mécanisme de profil UML [UML09] permet de sélectionner et de stéréotyper (étiqueter) les concepts UML afin qu'ils s'adaptent aux domaines à modéliser. En contrepartie, cela implique un certain nombre de compromis ainsi qu'un couplage étroit avec UML. Par définition, un ensemble de stéréotypes et de valeurs marquées ne constitue pas une sémantique alors qu'un métamodèle en constitue une même si elle n'est pas formelle. Bien qu'il soit séduisant de se conformer à un standard supporté par un grand nombre d'outils, nous pensons que, dans le cadre du développement de langages de modélisation très précis, une approche spécifique au domaine peut être profitable [JK09]. En effet, il n'est pas toujours possible de se conformer à UML et d'en définir une extension pour prendre en charge des besoins caractéristiques du domaine. En outre, faire le choix de définir un DSML, ce n'est pas renoncer aux standards. Le processus de définition d'un profil UML est sensiblement identique à celui de la création d'un DSML. D'ailleurs un profil est un DSML, à ceci près qu'il emprunte sa syntaxe et une partie de sa sémantique à UML. Toutefois, dans les deux cas, une analyse fine du domaine est nécessaire, elle ne consti-

tue pas un surcoût pour l'approche « *Domain-Specific* ». En effet, ces deux approches ne diffèrent que sur le choix de la réalisation technique. Il s'agit d'une part de définir un métamodèle à l'aide d'un langage de métamodélisation, par exemple Ecore [SBPM09] d'autre part, de définir une extension ou une restriction du métamodèle d'UML.

Plusieurs travaux mettent en évidence l'apport de l'utilisation d'un DSML par rapport à une approche classique [JK09, JLT04, KT08]. Cet apport se présente principalement en un gain de productivité grâce à la simplicité d'utilisation d'un DSML. Toutefois, la définition d'un DSML n'est pas tâche facile et on peut vite tomber dans les travers de cette approche [KP09], *i.e.* langage complexe avec trop de concepts, un éditeur difficile à utiliser, etc. Quelques travaux préconisent de bonnes pratiques pour la conception de DSML [KKP+09].

4.1.2 Objectifs de MATeL

MATeL est un langage de modélisation dont l'objectif est de permettre à un utilisateur de décrire des scénarios de test pour applications mobiles. Ces scénarios contiennent les actions qu'un testeur manuel doit effectuer et répéter sur tous les téléphones à tester (appuyer sur des touches, valider les résultats, configurer le téléphone, prendre des captures de quelques menus...). Un modèle MATeL n'est pas en interaction avec l'application à tester mais avec le téléphone en test. Ceci revient à dire que le scénario n'a aucune information sur les concepts métier, sur l'IHM ou encore sur le workflow de l'application. Il sait par contre comment exécuter l'application et naviguer dans tous ses menus. En effet, le scénario MATeL est une description des actions qu'un testeur doit faire à la main. Ainsi, MATeL a pour but d'aider le testeur manuel à se concentrer sur la conception de son scénario et non plus à refaire les mêmes tests sur tous les téléphones. Ceci est nettement plus valorisant pour lui ; on passe du testeur « presse boutons » au testeur « concepteur de test ». Afin de pouvoir modéliser un scénario, le concepteur du test doit comprendre les étapes nécessaires à l'utilisation de l'application et les résultats attendus à chacune de ces étapes.

Comme nous l'avons mentionné dans le paragraphe précédent, la syntaxe concrète de MATeL est inspirée de celle des diagrammes de séquence UML qui permettent de décrire les interactions entre objets. Plusieurs travaux tels que [XL06, SKM07] se sont basés sur ce type de diagrammes UML pour la description de cas d'utilisation dans le but de tester les systèmes modélisés. Nous avons opté pour cette représentation graphique principalement parce qu'elle nous permet de modéliser d'une manière simple et intuitive le type de scénarios de test que l'on souhaite obtenir.

4.2 Le métamodèle MATeL

Les langages de modélisation spécifiques permettent de fournir un moyen d'abstraction adapté à un domaine précis (le test d'applications mobiles dans notre cas). Ces langages sont basés sur des métamodèles qui permettent de décrire les concepts précis d'un domaine et de caractériser les relations entre ces concepts. Ils apportent en plus

une garantie quant au respect de la structure qu'ils définissent au sein des modèles qui s'y conforment. Les modèles MATeL (scénarios de test) ont pour but de montrer la séquence des interactions entre testeur et mobiles sur un axe de temps donné. Un modèle MATeL est constitué de deux dimensions. La dimension verticale montre la séquence des messages dans le temps. La dimension horizontale montre les instances des objets auxquelles sont envoyés les messages (testeur et mobiles dans notre cas). Le temps est représenté du haut vers le bas le long des lignes de vie ("LifeTime Line" en anglais) [60]. Les messages sont représentés par des flèches d'un acteur vers un autre. Nous identifions les interactions entre « testeur », « mobile en test » et « mobile secondaire ». On définit les messages entre les entités comme l'exécution de l'application, l'appui sur une touche, la capture d'une image du mobile ou encore l'envoi d'une interruption de type SMS.

L'une des valeurs ajoutées de notre approche réside dans la capacité de ce DSML à intégrer des points de variabilité (dans l'esprit des lignes de produits logiciels) pour exprimer des variations qui peuvent modifier le scénario d'un ensemble de mobiles à un autre. Ceci permet ainsi de concevoir un seul scénario de test pour des téléphones ayant des caractéristiques matérielles et logicielles différentes.

MATeL se base sur un métamodèle Ecore [SBPM09]. Les langages de métamodélisation tels que Ecore permettent d'exprimer une première forme de contraintes de construction par le biais de multiplicités (cardinalités). Celles-ci permettent, par exemple, d'exprimer qu'un concept ne peut être présent qu'une seule fois dans un modèle. Cependant, ce type de contraintes reste limité ; l'OMG a proposé au travers de son langage OCL (*Object Constraint Language*) [OMG06] le moyen d'exprimer des règles plus complexes sur les modèles et d'en vérifier la validité. Ce langage permet de manipuler des ensembles et des expressions de logique tout en exprimant la navigation au sein d'un modèle. OCL est utilisable tant au niveau modèle que métamodèle. De cette façon, il exprime des règles complexes de bonne formation, au niveau d'un métamodèle, pour en vérifier le respect sur les modèles qui s'y conforment. C'est dans cette logique que nous avons integré un ensemble de règles OCL simples à notre métamodèle.

Tous les métatypes MATeL sont liés au domaine du test d'applications mobiles. Dans cette section, nous proposons de décrire ces métatypes ainsi que leurs relations et les règles OCL qui garantissent la cohérence des modèles MATeL.

4.2.1 Noyau du métamodèle

Les éléments du métamodèle sont regroupés en fonction des points spécifiques du test qu'ils modélisent. Techniquement parlant, ce métamodèle a été réalisé grâce au langage de métamodélisation Ecore du *plugin* EMF (*Eclipse Modeling Framework*) d'Eclipse. Ceci, nous permet de bénéficier de nombreux outils pour décrire la syntaxe abstraite du langage correspondant. En outre, il permet une représentation standard des modèles

60. Nous utilisons ici les terminaisons définis pour les diagrammes de séquence UML

sous la forme de fichiers XMI ([61]) garant de l'interopérabilité au sein des *plugins* orientés modèle de la plateforme Eclipse.

La figure 4.1 présente le noyau du métamodèle MATeL. Ce noyau est composé par les principaux métatypes nécessaires pour la description d'un scénario de test :

- un testeur unique (cardinalité 1) modélisé par le métatype ***Tester*** ;
- un ou deux mobiles (cardinalité 1..2) modélisés par le métatype ***Mobile*** ;
- des messages (cardinalité 1..*) modélisés par le métatype abstrait ***Message*** ;
- des interruptions (cardinalité 0..*) modélisées par le métatype abstrait ***Interruption*** ;
- des points de variations (cardinalité 0..*) modélisés par le métatype abstrait ***VariabilityFragment*** et le métatype ***VariabilityAlternative*** ;
- des sous-séquences (cardinalité 0..*) modélisées par le métatype ***Subsequence***.

Un modèle MATeL (une instance de ce métamodèle), *i.e.* un scénario de test doit obligatoirement contenir un testeur, un mobile et un ou plusieurs messages échangés entre eux.

Tous les métatypes présentés ci-dessus, à savoir *Tester*, *Mobile*, *Message*, *Interruption*, *VariabilityFragment*, *VariabilityAlternative* et *Subsequence*, sont attachés au métatype *Scenario* par une relation de composition (losange noir). Cela signifie que toutes les instances de tous ces métatypes sont toujours contenues dans une instance du métatype *Scenario*.

61. XML Metadata Interchange : http ://www.omg.org/spec/XMI/

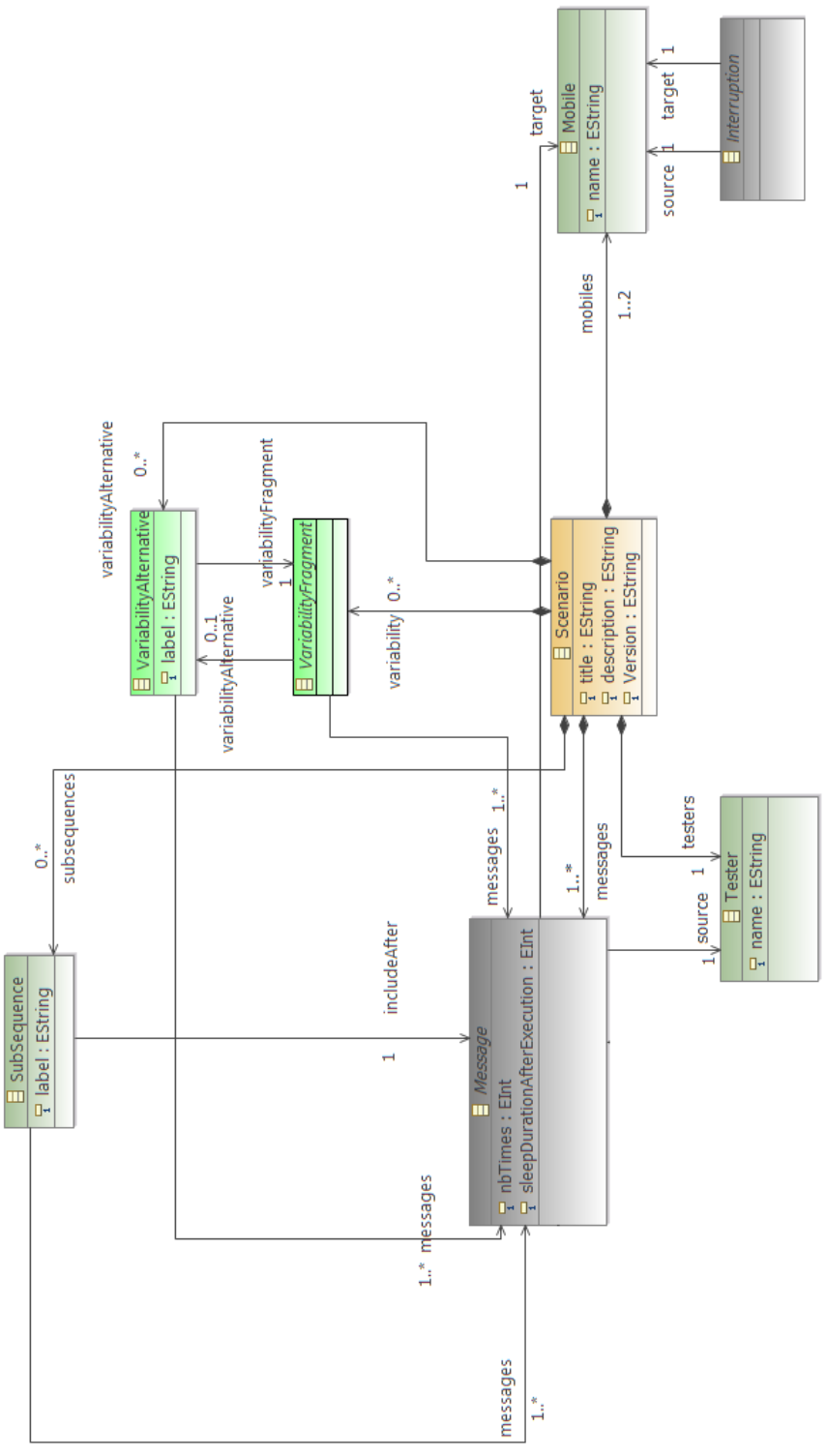


FIGURE 4.1 – Principaux concepts du métamodèle MATeL

4.2.1.1 Le métatype *Tester*

Le testeur, représenté par une ligne verticale, est obligatoire (pas plus d'un seul testeur dans un scénario). Il désigne l'utilisateur qui exécute des actions sur les téléphones mobiles en test. MATeL fournit une liste de toutes les actions qui peuvent être effectuées (toutes héritent du métatype abstrait *Message*). Ces actions peuvent être « atomiques », par exemple l'appui sur une touche, ou « composées » par exemple la saisie d'un texte. Grâce à ces actions, le concepteur du test est capable de décrire tout ce qu'un utilisateur pourrait faire avec un mobile entre les mains. Nous reviendrons plus tard sur ces différentes actions.

4.2.1.2 Le métatype *Mobile*

Dans un scénario MATeL, il est possible d'avoir deux instances du métatype *Mobile*. La première est une ligne verticale obligatoire et correspond à une représentation du ou des mobiles en test. La seconde est une ligne optionnelle qui représente un téléphone permettant la génération d'interruptions (appel entrant, SMS, etc.). La notion d'interruption est détaillée plus tard dans le chapitre. Afin de pouvoir distinguer les deux téléphones, nous avons défini la règle OCL suivante :

Contrainte Le téléphone en test et le téléphone secondaire doivent avoir des noms différents.

```
1 Context Scenario inv:
2
3   mobiles->forAll(m1, m2 | m1 <> m2 implies
4                   m1.name <> m2.name);
```

Cette règle peut aussi être exprimée ainsi :

```
1 Context Scenario inv:
2
3   mobiles->isUnique(name);
```

4.2.1.3 Le métatype *Message*

MATeL offre le métatype *Message* comme un moyen d'interaction entre le testeur et le mobile pour exprimer la façon dont l'application est manipulée (et donc testée) grâce à des stimuli externes (actions de l'utilisateur). Le métatype *Message* est abstrait, donc pas directement instanciable par l'utilisateur mais tous les métatypes qui en héritent le sont. Cette abstraction apporte plus de lisibilité au métamodèle et permet de l'étendre sans trop de difficulté. Comme tous les métatypes de MATeL, chaque instance de *Message* possède différentes propriétés qui doivent être réglées par l'utilisateur grâce à l'éditeur de modèles MATeL. Par exemple, si le testeur choisit la fonction *PressKey*, il doit alors définir avec précision la touche sur laquelle il faut appuyer (nous proposons une liste des touches communes à la plupart des téléphones mobiles qui sont fréquemment utilisées

dans les applications). Le concepteur du test peut ajouter dans son scénario autant de messages qu'il le souhaite.

Nous avons défini quelques contraintes OCL sur le métatype *Message*. Ces contraintes assurent que les instances de ce métatype sont cohérentes et correctement paramétrées pour pouvoir les exécuter sur le banc de test. Le métatype *Message* possède deux attributs :

- ***sleepDurationAfterExecution*** désigne la durée d'attente (en secondes) une fois que le message en question exécuté. Ceci est utile dès lorsqu'après une action donnée, le téléphone a besoin de quelques secondes pour passer d'un état à un autre (connexion, accès au système de fichiers...). Cette durée exprimée en seconde doit donc être positive. Cette contrainte est exprimée à la ligne 3 du listing 4.1 ;
- ***nbTimes*** désigne le nombre de fois que le message doit être répété. Ce nombre doit être supérieur ou égal à 1, ce qui correspond à au moins une exécution. Cette contrainte est exprimée par la règle OCL de la ligne 4 du listing 4.1.

Listing 4.1 – Contraintes sur le métatype Message

```
1 Context Message inv :
2
3     sleepDurationAfterExecution >= 0;
4     nbTimes >= 1;
5
6 Conext Scenario inv :
7
8     messages->forAll(m1, m2 | m1 <> m2 implies m1.target = m2.target);
```

La dernière règle (ligne 5) impose au testeur de ne communiquer qu'avec un seul mobile, donc avec le/les mobiles en test. Ceci évitera que le testeur communique avec le second mobile utilisé pour la génération d'interruptions. Ce qui génèrerait des scénarios ambigus non exécutables par le banc de test.

La manipulation d'un téléphone mobile passe principalement par l'appui sur les touches du clavier ou par l'appui sur l'écran tactile. Ceci dépend des caractéristiques matérielles du téléphone. MATeL propose ces deux types de manipulation.

Utilisation du clavier Le métatype abstrait *ManageKeyPad* hérite du métatype *Message* (cf. figure 4.2). Ainsi, le testeur peut envoyer vers le mobile en test des actions de ce type. Nous avons défini plusieurs types possibles :

- ***PressKey*** - cela correspond à l'appui simple sur une touche du clavier du mobile (attribut *KeyCode*) à spécifier par le concepteur du test. Il est possible de spécifier plusieurs touches. Dans ce cas, cela sera interprété comme une succession d'appuis avec des touches différentes ;
- ***PressDoubleKey*** - sur certains téléphones, il est nécessaire d'appuyer simultanément sur deux touches pour effectuer une action. Cette fonction permet de le

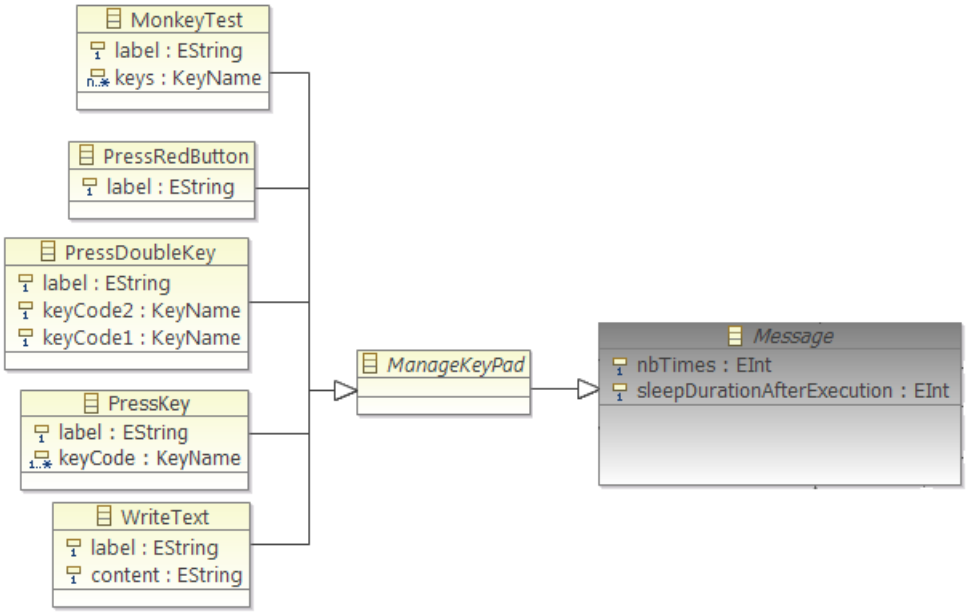


FIGURE 4.2 – Extrait du métamodèle pour la gestion des entrées clavier

réaliser en spécifiant les codes des deux touches à utiliser. La contrainte ci-dessous impose que les deux touches soient différentes pour que l'action reste cohérente ;

Contrainte Pour la fonction *PressDoubleKey*, les touches doivent être différentes.

```
1 Context PressDoubleKey inv:
2
3   keyCode1 <> keyCode2;
```

- ***PressRedButton*** - dans les procédures de test, il est nécessaire de s'assurer que l'application réagit « proprement » à l'appui sur la touche « raccrocher ». Cette touche arrête brutalement l'application, il y a donc un risque de perte de données ;
- ***WriteText*** - la saisie d'un texte, dans un formulaire par exemple, nécessite plusieurs appuis sur le clavier. Afin de ne pas surcharger les scénarios de test et de faciliter la saisie pour le concepteur du test, nous proposons une fonction qui prend en paramètre uniquement le texte à saisir. Ce texte est ensuite transformé en une suite d'actions de type *PressKey* en utilisant automatiquement le code de touche adéquat propre à chaque téléphone. Ceci est possible grâce à une base de données liée au banc de test et qui contient toutes les informations sur chaque mobile connecté dans le banc de test (schéma du clavier, taille d'écran ou encore fonctionnalités supportées telles que le Bluetooth). La fonction *WriteText* permet aussi l'injection automatique de données de test, notamment pour le remplissage

de formulaire ;
- ***MonkeyTest*** - il s'agit de plusieurs appuis successifs sur des touches différentes. Ce test sert à vérifier la robustesse de l'application et sa capacité à gérer des entrées inattendues.

Pour les fonctions *PressKey*, *PressDoubleKey* et *MonkeyTest*, le concepteur du test a la possibilité de spécifier la ou les touches à utiliser en les sélectionnant directement à partir d'une liste prédéfinie dans MATeL.

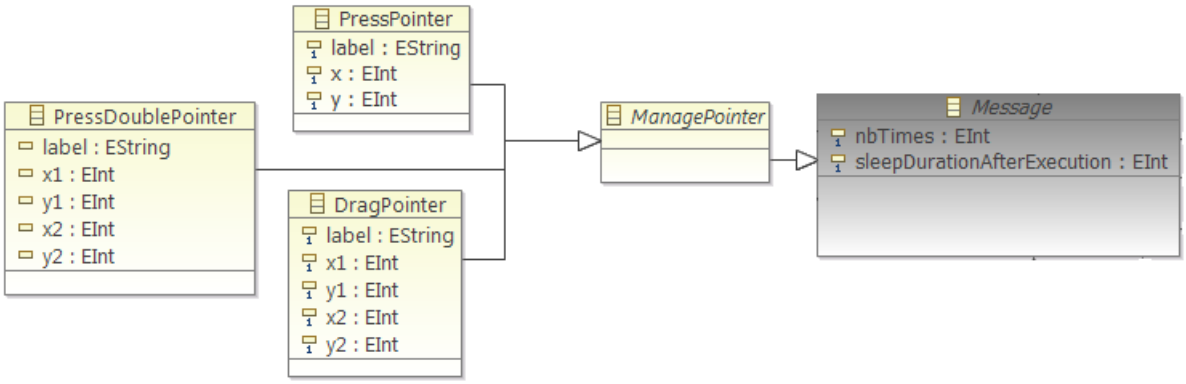


FIGURE 4.3 – Extrait du métamodèle pour la gestion des entrées écran

Utilisation de l'écran tactile Sur les téléphones disposant d'écran tactile, le test d'une application est principalement lié à la gestion du pointeur (cliquer sur une zone, glisser le pointeur, toucher deux zones simultanément, etc.). MATeL propose trois fonctions qui héritent du métatype abstrait *ManagePointer* (cf. figure 4.3) :

- ***PressPointer*** - cette fonction permet de cliquer sur un point donné de l'écran. Ce point est défini par l'utilisateur en spécifiant ses coordonnées (x,y) dans un repère cartésien. Ces coordonnées doivent toujours être positives, d'où l'interêt de la règle OCL ci-après.

  Contrainte Les attributs x et y d'une instance du métatype *PressPointer* doivent toujours être positifs.

```
1 Context PressPointer inv:
2
3   x >= 0 ;
4   y >= 0 ;
```

- ***PressDoublePointer*** - certains téléphones disposent d'un écran multi-touches qui permet de détecter deux voire plusieurs appuis simultanés. La fonction *PressDoublePointer* permet à l'utilisateur de bénéficier de cette fonctionnalité. Cette

fonction n'est cohérente que si les deux points sont correctement définis et différents l'un de l'autre. La règle OCL suivante permet d'assurer ces conditions :

Contrainte Règles sur les atributs du métatype *PressDoublePointer*.

```
1  Context PressDoublePointer inv:
2
3    x1 >= 0;
4    y1 >= 0;
5    x2 >= 0;
6    y2 >= 0;
7
8    x1 <> x2 or y1 <> y2;
```

– ***DragPointer*** - cette fonction permet de dessiner une ligne entre deux points. Ce qui correspond à un glissement de doigt sur l'écran du mobile. La cohérence de cette fonction est vérifiée grâce à une règle OCL semblable à celle présentée ci-dessus.

Le fait de fournir des coordonnées (x,y) pour les fonctions énumérées ci-dessus nous a été imposé par le robot d'exécution lié au banc de test en ligne qui, lui, a besoin de ces valeurs. Sur certaines applications, ceci pourrait être un handicap pour le concepteur du test qui doit trouver les coordonnées de plusieurs composants graphiques. Afin d'aider le concepteur du test dans cette tâche, nous avons défini une fonctionnalité dans le banc de test qui permet de récupérer les coordonnées (x,y) sur n'importe quel téléphone en le manipulant directement.

Paramétrage du téléphone Le test effectif d'une application commence une fois celle-ci lancée. Cependant le testeur est souvent obligé d'effectuer plusieurs paramétrages avant et après ce test effectif. Ces paramétrages sont fastidieux et répétitifs. Dans MATeL nous proposons une liste de fonctions afin de les automatiser. Ces fonctions sont représentées par des métatypes héritant d'un métatype générique abstrait *MobileSetting* (cf. figure 4.4).

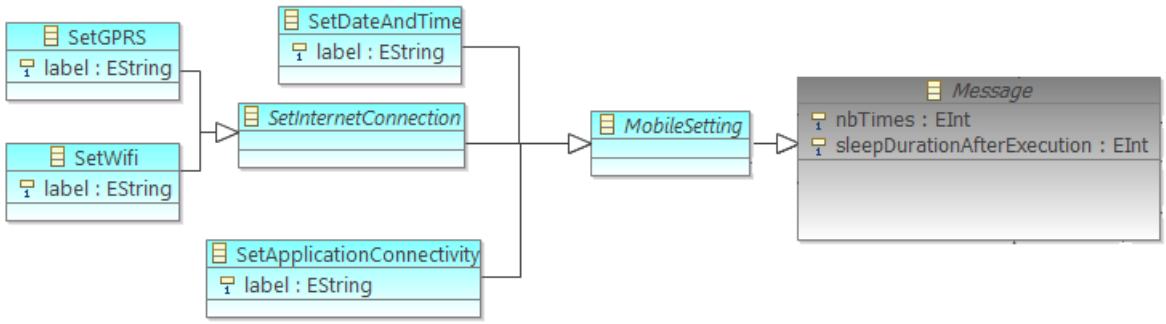


FIGURE 4.4 – Extrait du métamodèle pour la configuration du mobile

Nous avons défini trois métatypes :

- ***SetDateAndTime*** - comme son nom l'indique, cette fonction permet au concepteur du scénario de régler l'heure et la date du téléphone. Le robot de test (banc de test) récupère automatiquement la date et l'heure système pendant l'exécution du test, ainsi il peut paramétrer le téléphone. Ceci est utile principalement pour le test d'applications sécurisées nécessitant des permissions de type connexion HTTPS ;
- ***SetInternetConnection*** - afin de pouvoir télécharger l'application à tester ou encore pour tester des applications connectées à des Web services, il est nécessaire de configurer la connexion Internet du téléphone. Cette configuration nécessite souvent des paramètres en fonction de l'opérateur (*Acces Point Name*, login et mot de passe, proxy...). Grâce à MATeL et au robot de test, le banc est capable, en fonction de l'opérateur de télécommunication auquel est connecté le mobile, de réaliser cette configuration. Deux métatypes permettent la connexion soit au réseau Internet de l'opérateur (*SetGPRS*), soit à un réseau Wi-Fi (*SetWifi*) ;
- ***SetApplicationConnectivity*** - certains téléphones exigent des autorisations particulières pour chaque application en fonction des APIs critiques qu'elles intègrent (connexion réseau, accès au système de fichiers ou encore accès au répertoire des contacts de l'utilisateur). Cette fonction définit automatiquement les autorisations nécessaires pour l'applicaion en test.

Nettoyage du téléphone Etant donné que le principe du *Cloud Testing* repose sur le partage de téléphones entre plusieurs personnes appartenant à des structures différentes, la question de la confidentialité se pose. En effet, chaque utilisateur du banc de test doit pouvoir effacer assez facilement toutes les données relatives à ses tests (adresses Internet, applications installées, données, cookies, etc.). MATeL fournit des métatypes le permettant. Ces métatypes héritent tous du métatype abstrait *CleanPhone* (cf. figure 4.5) :

- ***ClearBrowserHistory***, ***ClearCache***, ***ClearCookies*** - ces fonctions sont utiles pour effacer les traces laissées dans le navigateur utilisé pour télécharger l'application ainsi que toutes les connexions qui y sont associées ;
- ***RemoveApp*** - une fois le test terminé, cette fonction désinstalle automatiquement l'application. Elle parcourt le système de fichiers de chaque mobile testé et lance la désinstallation.

Ces fonctions permettent à l'utilisateur à la fois de gagner un temps précieux et de s'affranchir de toutes les spécificités de chaque mobile, notamment les menus de paramétrages et de configuration.

4.2.2 Gestion des interruptions

Un téléphone portable est un dispositif connecté qui interagit avec le monde extérieur. Pour cette raison, toutes les applications doivent se comporter correctement au regard

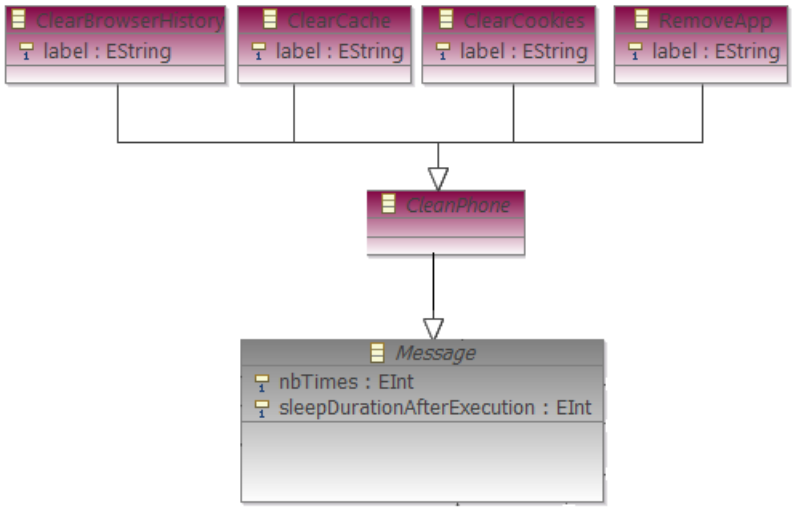


FIGURE 4.5 – Extrait du métamodèle pour le nettoyage du téléphone

de tous les types d'interactions admissibles. Ces dernières sont soit liées à l'utilisateur, soit liées au réseau mobile (GSM et 3G).

4.2.2.1 Interruptions réseaux

MATeL dispose d'un ensemble de métatypes spécifiques pour la gestion des interruptions générées par les communications réseaux (cf. figure 4.6).

En pratique, le concepteur du test, utilise une seconde instance du métatype *Mobile* pour modéliser un second mobile générateur d'interruptions externes. Celles-ci sont définies en trois catégories :

- ***SendMessage*** - afin de s'assurer que la réception d'un SMS ou d'un MMS n'altère pas le bon fonctionnement de l'application mobile (perte de données, blocage...), le concepteur du test peut programmer des envois de messages à n'importe quel moment du test et ainsi tester le comportement de l'application. Les métatypes *SendSMS* et *SendMMS* permettent respectivement l'envoi de SMS et de MMS ;
- ***ConnectBluetooth*** - la réception d'une demande de connexion externe par Bluetooth fait aussi partie de ces interruptions. MATeL offre ainsi la possibilité d'en ajouter une dans le scénario ;
- ***ManageCall*** - le rôle principal d'un mobile étant d'appeler et de recevoir des appels téléphoniques, le développeur d'applications mobiles doit s'assurer que d'une part, l'application ne dégrade pas la qualité de l'appel (exemple : le son joué dans l'application doit être coupé pendant l'appel) et, d'autre part, que l'application continue de fonctionner « proprement » après l'appel. Pour cette raison, nous avons ajouté dans MATeL les métatypes *StartCall* pour lancer un appel, *AcceptCall* pour accepter un appel entrant, *RejectCall* pour refuser un appel entrant et *EndCall*

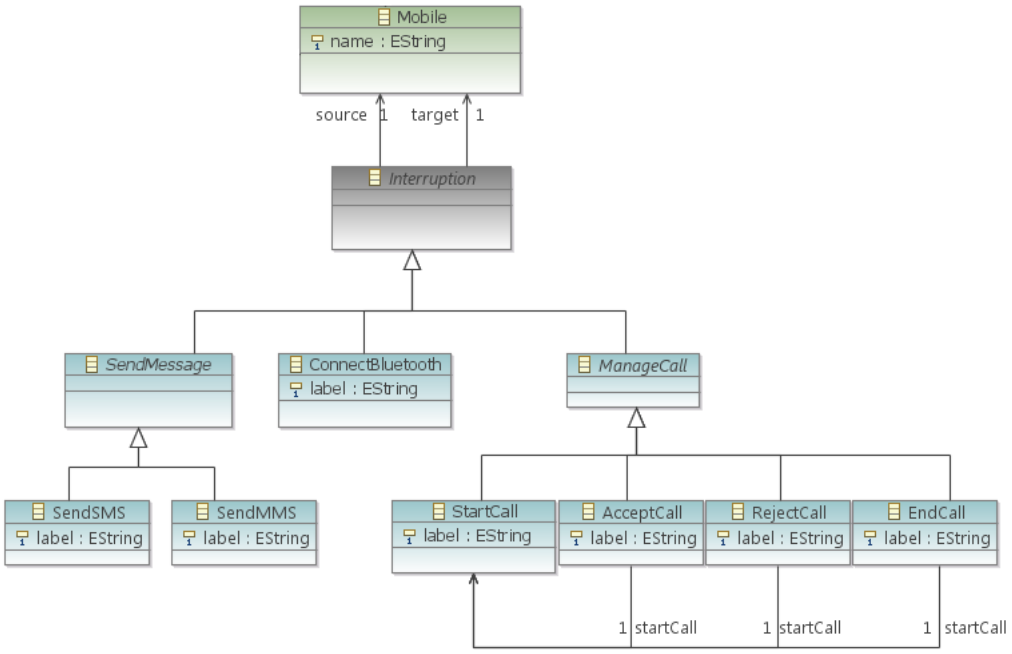


FIGURE 4.6 – Métatypes pour la gestion des interruptions

pour mettre fin à un appel en cours. Avec ces métatypes, le concepteur du test est en mesure de modéliser tous les scénarios d'utilisation possibles entre le mobile en test et le mobile secondaire générateur d'appels.

Afin d'éviter qu'un téléphone en test puisse s'auto-envoyer des interruptions en cours du test, ce qui n'a pas de sens vu qu'un téléphone exécutant une application ne peut pas accèder au menu d'envoi de messages ou d'appels, nous avons défini une règle OCL qui impose qu'une interruption doit être envoyée par un autre téléphone.

Contrainte Un mobile en test ne s'auto-envoie pas des interruptions.

```
1 Context Interruption inv:
2
3   target <> source;
```

4.2.2.2 Interruptions utilisateur

Le second type d'interruptions est généré par l'utilisateur du mobile lui-même. Ces interruptions sont liées à la manipulation physique du mobile en fonction de ses spécificités matérielles. L'application en test doit subir ces interruptions afin de vérifier sa robustesse et sa capacité à continuer de fonctionner correctement. La figure 4.7 est un

extrait du métamodèle MATeL consacré à la gestion de ce genre d'interruptions.

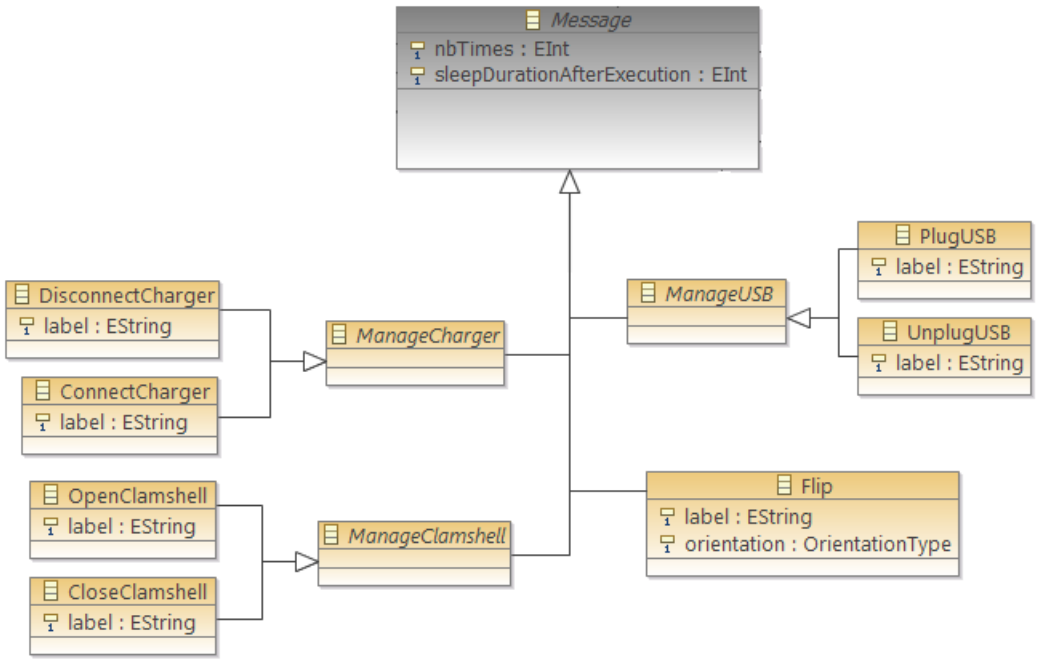


FIGURE 4.7 – Extrait du métamodèle pour la gestion des interruptions utilisateur

Les interruptions sont regroupées en trois catégories définies ci-après.

- ***ManageUSB***, ***ManageCharger*** - ces interruptions sont générées par le branchement et le débranchement du chargeur et/ou d'un câble USB (si le téléphone en dispose). En effet, l'application doit continuer à fonctionner « proprement » si une telle interruption est générée ;
- ***ManageClamShell*** - si le téléphone dispose d'un clavier à clapet ou d'un clavier coulissant, il est souvent nécessaire de tester le comportement de l'application face à un changement d'état de ses claviers (ouverture/fermeture). Ceci est possible grâce aux métatypes *OpenClamShell* et *CloseClamShell* ;
- ***Flip*** - pour les téléphones disposant de capteur de rotation, il est nécessaire de tester l'application dans les deux modes d'affichage (portrait et paysage) afin de s'assurer qu'elle fonctionne correctement dans les deux modes. L'utilisateur est capable de modéliser ceci grâce au métatype *Flip* qui prend en paramètre l'orientation souhaitée (verticale ou horizontale).

Etant donné que les téléphones sont physiquement fixés dans le banc de test, ce type d'interruptions est simulé électroniquement. Sur certains téléphones, à cause de restrictions techniques, il est impossible de les exécuter.

4.2.3 Gestion des résultats

Comme nous l'avions évoqué dans le chapitre 1, le test d'applications mobiles est de type « boîte noire » d'une part parce que le robot de test n'interagit pas directement avec le code source, d'autre part parce qu'aucune interface n'est fournie pour communiquer avec le système en test. Ce qui veut dire que MATeL pilote le téléphone en test pour exécuter et manipuler l'application à tester. Les tests fonctionnels peuvent être réalisés en utilisant des simulateurs. Par exemple, vérifier que la saisie d'un mot de passe erroné est signalée à l'utilisateur (tests avec des paramètres d'entrée pour les cas passants et non passants). En revanche, une fois l'application déployée, les résultats attendus et les résultats obtenus sont souvent des interprétations visuelles et/ou sonores du comportement de l'application sur le téléphone.

MATeL permet au concepteur de test d'ajouter au scénario des points de contrôle. Il s'agit de prendre une capture de l'écran des mobiles en test, de les filmer ou d'enregistrer des séquences audio. Ces fonctionnalités sont intégrées dans MATeL grâce au métatype abstrait *Checkpoint* (cf. figure 4.8) :

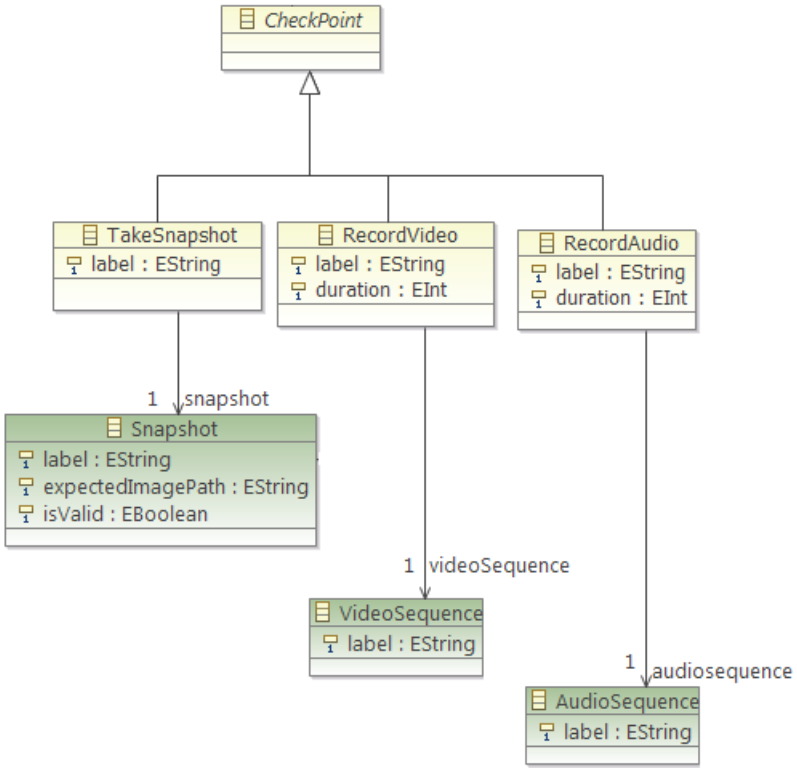


FIGURE 4.8 – Métatypes pour la gestion des résultats

– ***Snapshot*** - si, à un moment donné du test, le testeur souhaite faire une capture

d'écran des mobiles, il doit ajouter dans son scénario un message de type *TakeSnapshot*. A ce message il faut associer une instance de *Snapshot*. Grâce à l'attribut *expectedImagePath*, l'utilisateur peut spécifier le résultat attendu (capture d'écran attendue). Une fois l'exécution du test terminée, il peut qualifier le résultat grâce à l'attribut *isValid* (pass/fail) ;

- ***AudioSequence*** et ***VideoSequence*** - MATeL permet aussi l'enregistrement de séquences vidéo des mobiles en test et/ou des séquences audio du son émis par ces mobiles. Pour cela il suffit d'utiliser les messages de type *RecordVideo* et *RecordAudio* auxquels il faut associer respectivement des instances de *VideoSequence* et *AudioSequence*. A la fin du test, l'utilisateur peut jouer les séquences enregistrées et ainsi vérifier le comportement de l'application.

Afin de pouvoir générer des fichiers images, vidéo ou audio, le concepteur de test doit fournir des noms différents pour chacune des instances du métatype *Checkpoint*. Les contraintes OCL présentées ci-dessous valident cette règle.

Contrainte Les captures d'écran doivent avoir des noms différents.

```
1 Context Snapshot inv :
2
3   Snapshot.allInstances()->forAll (s1 , s2 | s1 <> s2 implies
4                 s1.label <> s2 .label ) ;
```

Contrainte Les séquences audio doivent avoir des noms différents.

```
1 Context AudioSequence inv :
2
3   AudioSequence.allInstances()->forAll (a1 , a2 | a1 <> a2 implies
4                 a1.label <> a2 .label ) ;
```

Contrainte Les séquences vidéo doivent avoir des noms différents.

```
1 Context VideoSequence inv :
2
3   VideoSequence.allInstances()->forAll (v1 , v2 | v1 <> v2 implies
4                 v1.label <> v2 .label ) ;
```

A ce jour, l'oracle du test n'est pas automatisé. C'est à la charge du testeur de valider les résultats obtenus à la fin du test. Nous pourrions imaginer par exemple pour les captures d'écran, des comparaisons automatiques d'images avec un seuil de tolérance donné.

4.2.4 Gestion de la variabilité

Comme nous l'avons indiqué dans l'introduction générale, la grande difficulté du test d'applications mobiles est liée à la répétitivité de la procédure sur des modèles de mobiles tous différents les uns des autres. C'est pour cette raison que nous avons enrichi MATeL

pour pouvoir gérer de telles différences dans le test. Ainsi le concepteur du test n'a besoin que d'un modèle de test MATeL unique pour l'ensemble des téléphones à tester.

Nous distinguons deux types de variabilité, l'une liée aux téléphones ciblés, l'autre aux applications à tester. En effet, la procédure de test peut varier soit parce que les caractéristiques du mobile, appelées aussi « *features* » dans le domaine de l'ingénierie des lignes de produits logiciels, l'imposent (exemple : type de clavier), soit parce que l'application à tester a été adaptée pour une plateforme cible donnée.

4.2.4.1 Variabilité au niveau des applications

Souvent, une même application mobile doit être adaptée en fonction des téléphones ciblés (cf. chapitre 1). Dans l'esprit des lignes de produit logiciel, cinq différents types de variabilité sont définis dans MATeL [62]. Ces cinq types de variabilité sont fournis par des spécialistes en développement d'applications mobiles. Selon eux, une application peut être adaptée principalement en raison de ces variations. En pratique, l'utilisateur est en mesure d'attacher des actions spécifiques à ces points de variabilité. Ces concepts sont disponibles dans MATeL, grâce aux métatypes *ApplicationVersion*, *MobileSpecification*, *MobileModel*, *ScreenResolution* et *MobilePlatforms* qui héritent tous du métatype abstrait *VariabilityFragment* (cf. figure 4.10) :

- ***ApplicationVersion*** - cette variabilité est liée à la version de l'application à tester. Celle-ci est particulièrement utile pour la gestion des modifications ou évolutions entre versions de la même application. Ce métatype prend en paramètre le numéro de version sous forme d'une chaîne de caractères (exemple : « 1.0.0 »). Dans [Zia04], la variabilité a été représentée avec deux dimensions ; l'espace et le temps. La variabilité dans l'espace concerne les différences entre les produits alors que la variabilité dans le temps concerne les différentes versions du même produit (cf. figure 4.9). Le métatype *ApplicationVersion* permet de modéliser cette dimension temps ;
- ***MobileSpecification*** - cette variabilité repose sur les spécificités de chaque mobile. Le métamodèle MATeL en fournit une liste prédéfinie (*isTouchScreen*, *hasGPS*, *hasQWERTYKeyboard*, *hasAZERTYKeyboard*, *hasLandscapeMode* et *hasBluetooth*) à partir de laquelle l'utilisateur peut spécifier un ou plusieurs types. Ce métatype prend en paramètre une ou plusieurs caractéristiques. Afin que cette variabilité soit cohérente, elle ne peut pas avoir à la fois les paramètres *hasQWERTYKeyboard* et *hasAZERTYKeyboard*. La règle OCL ci-après permet d'éviter cela.

  Contrainte Le clavier d'un mobile ne peut pas être à la fois QWERTY et AZERTY :

```
1 Context MobileSpecification inv:
2
3 technicalDetails->includes(TechnicalDetail::hasAZERTYKeyboard) implies
4 technicalDetails->excludes(TechnicalDetail::hasQWERTYKeyboard) and
```

62. Cette liste peut éventuellement être enrichie afin de répondre à des nouveaux types de varaibilités

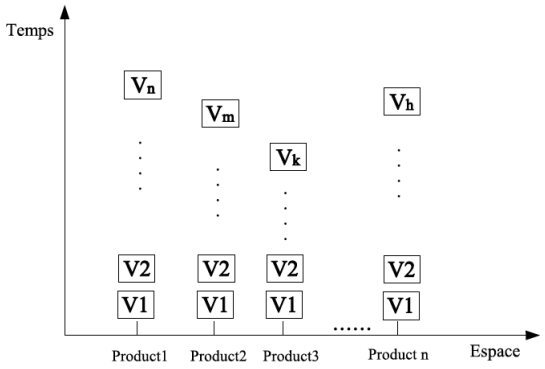


FIGURE 4.9 – Dimensions de la variabilité
[Zia04]

```
technicalDetails->includes(TechnicalDetail::hasQWERTYKeyboard) implies
technicalDetails->excludes(TechnicalDetail::hasAZERTYKeyboard) ;
```

– ***MobileModel*** - cette variabilité est relative aux modèles des mobiles. Par exemple : HTC Hero, iPhone 4, Nokia 3310, etc. Ce métatype prend en paramètre un ou plusieurs modèles de mobiles sous la forme d'une chaîne de caractères ;
– ***ScreenResolution*** - cette variabilité est dédiée aux résolutions d'écran. L'utilisateur peut spécifier la résolution qu'il souhaite. Ceci est possible en fournissant la largeur et la hauteur de l'écran du mobile grâce aux attribut *width* et *height*. Afin de s'assurer que le testeur fournisse des valeurs cohérentes, nous avons défini la règle OCL suivante qui impose des résolutions d'écran de 120 x 160 minimum. Ceci correspond à la plus petite taille d'écan de mobile disponible dans le banc de test ;

Contrainte Le concepteur du scénario doit saisir des valeurs cohérentes.

```
Context ScreenResolution inv:

  width >= 120;
  height >= 160;
```

– ***MobilePlatforms*** - la dernière variabilité est liée à la plateforme mobile (Android, Java ME, iOS, BlackBerry, Windows Phone, etc.). Une liste prédéfinie dans le métamodèle permet au concepteur du test de choisir la plateforme.

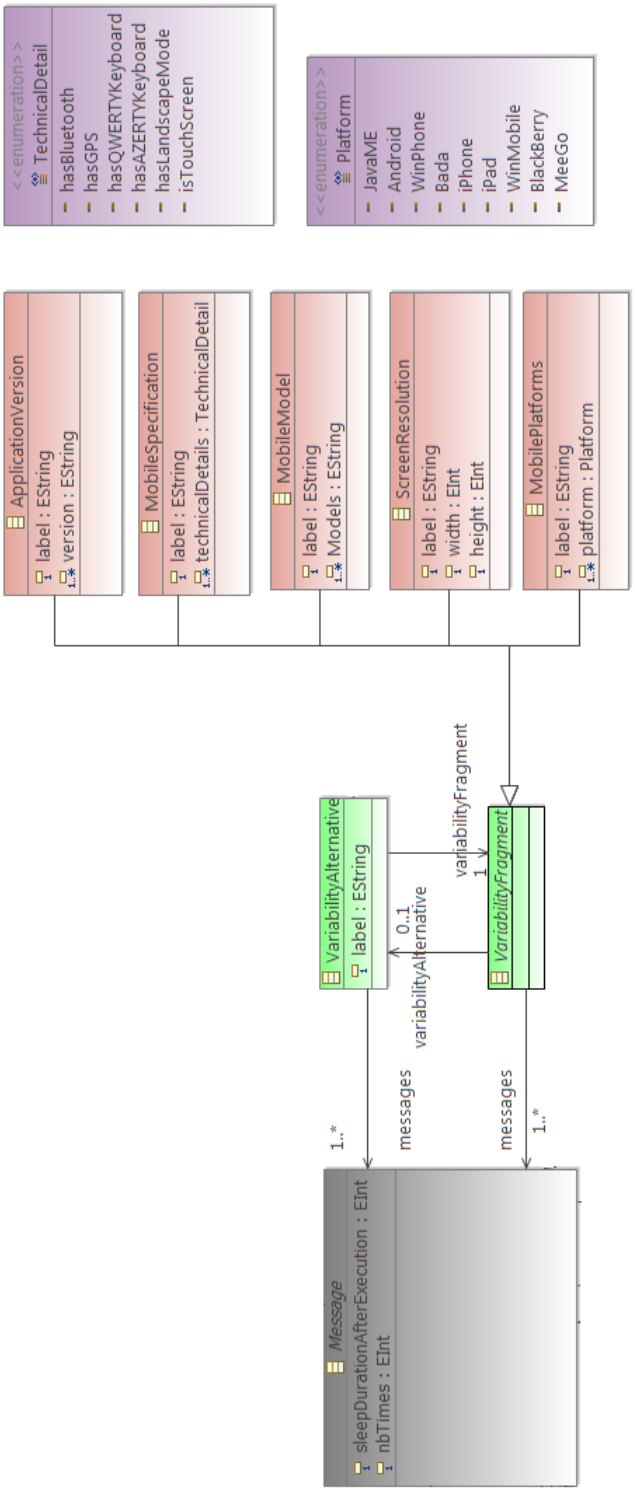


FIGURE 4.10 – Métatypes pour la gestion de la variabilité

L'utilisateur peut manipuler directement ces concepts dans un scénario de test. Un ou plusieurs messages peuvent être assignés à une instance de *VariabilityFragment*. Par exemple, considérons une application dans laquelle le développeur a décidé d'ajouter un écran de publicité pour inciter les utilisateurs à télécharger d'autres applications. Cet écran ne sera ajouté que pour les téléphones mobiles ayant une résolution de 320 x 240 pixels. Cela signifie que, lors du test, une action (pour annuler ou valider cet écran) doit être omise pour les téléphones mobiles ayant une résolution différente de 320 x 240 pixels. Grâce à la gestion de la variabilité fournie par MATeL, c'est assez simple de concevoir un tel point de variabilité.

Dans la figure 4.11, nous voyons une variabilité appelée *Screen Resolutions Variability* (rectangle bleu) à laquelle nous avons attribué une action de type *PressKey*.

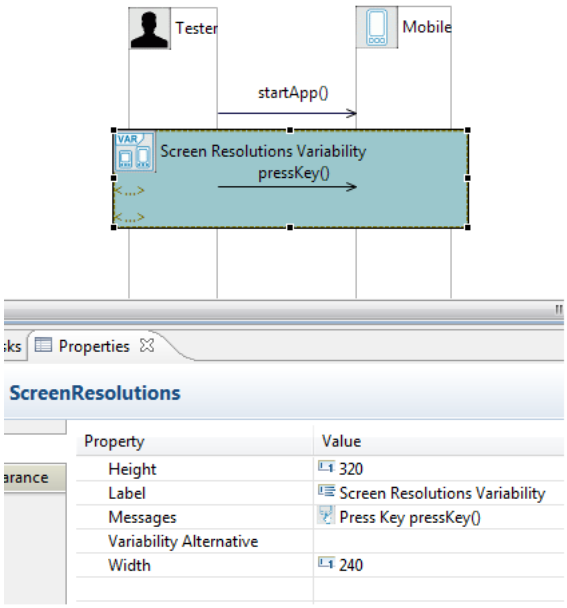


FIGURE 4.11 – Exemple de modélisation de variabilité

Cette variabilité est une instance du métatype *ScreenResolution* paramétrée par une résolution de 320 x 240 pixels. Grâce à la base de données, le banc de test est capable de réaliser cette action uniquement sur les mobiles ayant la bonne résolution d'écran. En pratique, l'action *pressKey()* est attachée à la variabilité *Screen Resolutions Variability* qui est une instance du métatype *ScreenResolution*. En utilisant l'onglet « Propriétés » de l'éditeur de modèles MATeL (cf. figure 4.11, partie basse), nous pouvons spécifier la hauteur et la largeur de l'écran. Lors de l'exécution du scénario sur le banc de test, il est relativement facile d'exécuter toutes les séquences communes en prenant en compte les points de variabilité. Ainsi, cette action est exécutée uniquement sur les téléphones appropriés. Si le concepteur du test souhaite spécifier des actions différentes sur d'autres

mobiles (dans l'esprit d'une condition if/else) alors le métatype *VariabilityAlternative* (cf. figure 4.10) peut être utilisé. Pour cela, il suffit d'ajouter une instance du *VariabilityAlternative*, de lui associer les messages adéquats et enfin de l'attacher au métatype *Screen Resolutions Variability*.

Dans le cas où le concepteur du test a besoin de spécifier plusieurs points de variabilité imbriqués (if/else if/else if...), il pourra utiliser plusieurs instances du métatype *VariabilityFragment*. Si, par exemple, l'utilisateur souhaiterait enrichir l'exemple présenté dans la figure 4.11 pour ajouter des traitements spécifiques aux téléphones ayant une résolution de 800 x 480, il suffit d'ajouter au scénario une nouvelle instance du métatype *ScreenResolution*.

4.2.4.2 Variabilité au niveau des téléphones

Selon le modèle du mobile à tester, les actions nécessaires pour accomplir ces différentes étapes varient (menus différents, touches différentes...). C'est la raison pour laquelle MATeL fournit des « fonctions automatisées » qui permettent de réduire le temps nécessaire avant de démarrer le test effectif des applications en automatisant toutes les phases annexes liées à la préparation du test ou sa terminaison. Par exemple, nous fournissons la fonction *SendSMS* où le testeur ne fournit que le texte qui sera envoyé. Grâce à la base de données renseignée avec toutes les caractéristiques de chaque téléphone du banc (propriétés de l'écran, type de clavier, correspondances de claviers, entrées/sorties disponibles, capteurs (mode paysage, rotation...), les menus de navigation, etc.), nous sommes en mesure d'adapter la fonction d'envoi de SMS à chaque mobile.

Cette variabilité est avérée lorsque l'utilisateur essaie de naviguer dans les menus du téléphone, lorsqu'il essaie de le configurer ou lorsqu'il essaie de saisir un texte. Pour cette raison, nous proposons une liste de fonctions incontournables pour le test. Nous avons détaillé le rôle de ces fonctions en présentant le noyau du métamodèle (cf. paragraphe 4.2.1.3) :

- ***SetWiFi*** ;
- ***SetGPRS*** ;
- ***SetDateAndTime*** ;
- ***SetApplicationsConnectivity*** ;
- ***SendSMS*** ou ***SendMMS***, ***WriteText*** ou ***Call*** ;
- ***ConnectBluetooth*** ;
- ***RemoveApp*** ;
- ***ClearBrowserHistory***, ***ClearCookies***, ***ClearCache*** ;
- ***StartApp***.

La dernière fonction (*StartApp*) explore le système de fichiers de chaque mobile en test pour trouver l'application installée et la démarrer. Ceci est possible principalement grâce, d'une part, à une base de données qui contient la suite des actions à effectuer pour accèder au répertoire dans lequel l'application a été installée, d'autre part à la sauvegarde de tous les téléphones dans un état connu après chaque utilisation, ce qui

nous permet de savoir quel est l'emplacement exact de l'application installée.

En résumé, MATeL fournit plusieurs métatypes permettant à l'utilisateur de spécifier des points de variation dans son scénario. Ces points de variation sont par la suite pris en compte lors de l'exécution du scénario. Un autre type de variabilité dans le test est quant à lui géré automatiquement via la base de données qui contient des informations sur les téléphones. Dans ce cas, le testeur n'est pas obligé de spécifier explicitement certaines variations qui n'ont pas d'impact sur le déroulement du test. Par exemple, si le testeur ajoute une fonction de type *PressRedButton* alors il n'est pas obligé de lui associé une variabilité spécifique parce que le robot de test sait sur quel téléphone il faudra l'ignorer, en l'occurence ceux qui ne disposent pas de la touche « raccrocher », principalement les téléphones équipés d'écran tactile.

Grâce aux fonctionnalités de MATeL, le testeur conçoit un scénario de test unique pour tous les mobiles ciblés ; le banc de test l'exécute correctement sur chaque appareil (cf. chapitre 5). De plus, les scénarios peuvent être réutilisés pour des applications similaires.

4.3 Conception de l'éditeur MATeL

Le langage MATeL est supporté par un éditeur graphique intégré à Eclipse sous forme d'un *plugin*. Ceci permet de bénéficier des capacités d'interopérabilité du format standard fourni par Ecore (XMI). Le développement de cet éditeur a également suivi un processus dirigé par les modèles. Les aspects graphiques (formes géométriques 2D, canvas, boutons, actions, etc.) sont gérés par le *plugin* GEF [63]. Ce dernier met en pratique le patron de conception MVC (Modèle Vue Contrôleur) afin de maintenir un couplage faible du modèle (s'appuyant sur Ecore et le *plugin* EMF) vis-à-vis de sa représentation graphique. Les deux *plugins* EMF et GEF sont mis en relation grâce à un troisième outil appelé GMF [64]. Il permet de générer de manière plus ou moins automatique un *plugin* d'édition graphique se basant sur GEF et utilisant les résultats de la métamodélisation en Ecore. L'approche GMF est intégralement dirigée par les modèles. Il s'agit de décrire tous les aspects de l'éditeur souhaité sous la forme de modèles conformes à différents métamodèles. Ainsi, le modèle du domaine (ou métamodèle) décrit en Ecore est associé au modèle d'outillage (décrivant la palette graphique et les actions de création et suppression possibles - fichier gmftool) et au modèle de représentation graphique (qui définit les différentes figures et leur propriétés de formes, tailles, etc. - fichier gmfgraph) par un quatrième modèle de mapping (fichier gmfmap). Ce dernier met en relation un concept du métamodèle avec sa représentation graphique et les outils qui lui sont associés et permet la création d'un modèle de génération paramétrable (fichier gmfgen). Ce modèle permet de personnaliser les paramètres de génération du code du *plugin*. Une fois ceux-ci complétés, GMF génère le code du *plugin* correspondant aux modèles définis (domaine, graphique, outillage et mapping) grâce à une transformation de modèles. La figure 4.12

63. Graphical Editing Framework : http ://www.eclipse.org/gef/
64. Graphical Modeling Framework : http ://www.eclipse.org/gmf/

décrit le déroulement du processus de génération d'un éditeur grâce à GMF. Le *plugin* généré ne propose cependant que des fonctionnalités de base. GMF est encore en cours de développement et propose régulièrement de nouvelles capacités de génération. De ce fait, nous avons dû étendre cette première version du *plugin* pour offrir des fonctionnalités supplémentaires et améliorer la présentation des divers éléments de modélisation.

Ainsi, nous avons ajouté à l'éditeur MATeL des extensions permettant la connexion au banc de test, la génération des scénarios en prenant en compte les points de variabilité. De même, nous avons ajouté plusieurs vues, notamment pour la gestion des résultats. Enfin, nous avons proposé à l'utilisateur diverses fonctionnalités pour l'assister dans la manipulation du banc de test via notre *plugin* (cf. chapitre 5).

En termes de résultats, ce travail nous a permis de distinguer les limites d'une approche générative dans le cas des éditeurs de diagrammes. Elle permet de mettre en place relativement rapidement un éditeur graphique fonctionnel mais nécessite un travail d'extension dès lors que des fonctionnalités non triviales sont requises. Quoi qu'il en soit nous avons obtenu un *plugin* répondant à nos attentes permettant de décrire graphiquement des modèles de scénario de test conformes à notre métamodèle et aux règles OCL décrites.

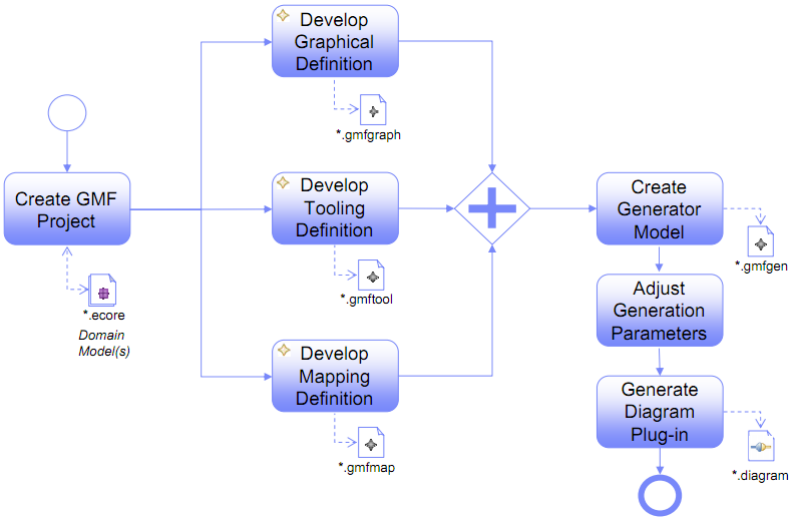


FIGURE 4.12 – Processus de génération d'un modeleur avec GMF

4.4 Génération automatique de scénarios MATeL

La génération automatique de scénarios de test pour applications standard (PC et Web) est un domaine de recherche vaste qui a été étudié dans divers travaux [Pac05, Zdu07] sur plusieurs types de test tels que les tests de conformité ou encore les tests de robustesse. Ces travaux ont abouti à la production de divers outils et à la défini-

tion de plusieurs approches. A notre connaissance, aujourd'hui, aucun outil ne permet de générer des scénarios de test pour applications mobiles multi-plateformes dans un cadre industriel. Afin de faciliter la description de scénarios MATeL, nous nous sommes penchés sur l'idée de les générer automatiquement. Pour cela, il faut avoir un minimum d'informations sur l'architecture de l'application à tester. Une approche dirigée par les modèles nous paraît adéquate pour ceci. En effet, nous proposons de définir un autre DSML appelé MobiAM pour *Mobile Application Modeler* qui permet de concevoir des applications mobiles multi-plateformes dans le but de les tester avec MATeL et le banc de test en ligne. Très peu de travaux proposent la conception d'applications mobiles en utilisant une approche dirigée par les modèles [DB07, GHKV08, BFH08]. Le projet Netbeans [65] offre la composante « Visual Mobile Designer » (VMD) qui permet de modéliser une application Java ME. La solution « App Studio » [66] propose aussi un environnement de ce type. Tous ces outils/travaux proposent des outils de modélisation basés sur des métamodèles propres aux applications mobiles. Nous utiliserons une approche similaire avec comme objectif de pouvoir générer les modèles MATeL automatiquement.

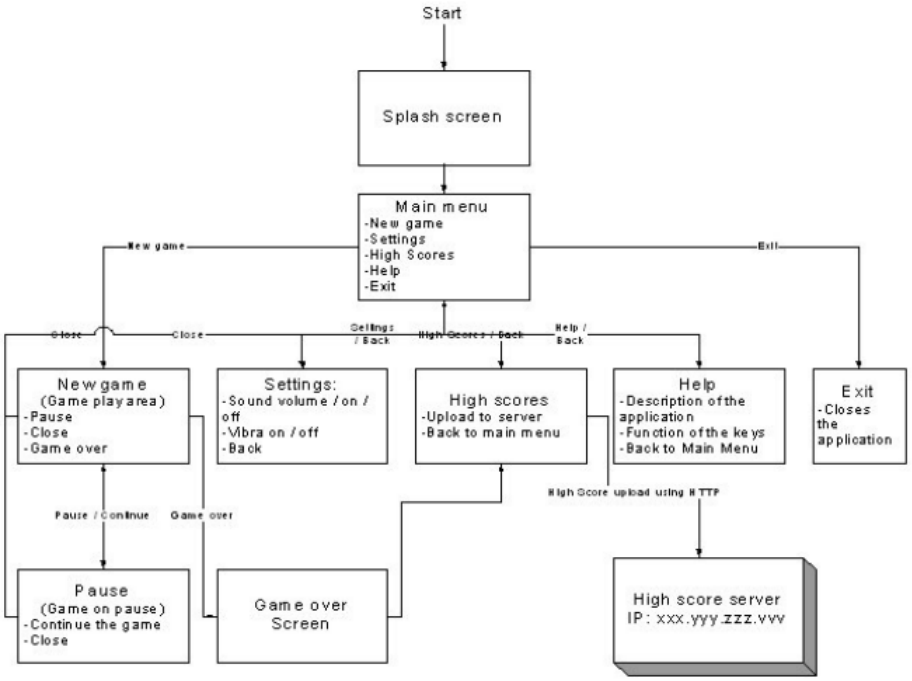


FIGURE 4.13 – Vue globale d'une application mobile [67]

En observant l'architecture générale des applications mobiles (cf. figure 4.13), nous constatons qu'il s'agit principalement d'un ensemble d'écrans avec des éléments gra-

65. http ://www.netbeans.org
66. http ://pyxismobile.com

phiques (boutons, zone de texte...) et des transitions entre ces écrans. Une transition peut être attachée à un ou plusieurs événements comme par exemple l'appui sur une touche. La figure 4.13 représente un exemple d'une vue globale d'une application mobile. Chaque rectangle représente un écran et les flèches représentent les transitions d'un écran à un autre. Quasiment toutes les applications mobiles peuvent être représentées sous cette forme à l'exception des scènes de jeux.

Le test d'une application mobile est un enchaînement d'actions sur le téléphone pour naviguer dans les différents menus et sous-menus dans le but de couvrir tous les chemins possibles. Cet enchaînement peut varier d'un téléphone à un autre. Prenons un exemple : supposons que nous avons une application dont le menu principal est composé de 7 éléments rangés en liste verticale et que chaque élément est dessiné dans un rectangle dont la taille est de 240 x 30 pixels (largeur x hauteur). Sur un téléphone dont la hauteur de l'écran est supérieure ou égale à 7 * 30 pixels, tous les éléments du menu sont accessibles directement alors que sur un téléphone avec un écran de taille inférieure, il faudra effectuer un scroll pour accéder aux éléments non affichés. Ainsi le test est variable en fonction du téléphone. **Si une telle variabilité est exprimée dans l'application alors nous pouvons la déduire dans le test**.

4.4.0.3 Le métamodèle MobiAM

Nous proposons un environnement qui permet de modéliser l'application sous la forme d'un workflow en exprimant la variabilité qui impactera le scénario de test. Les raisons qui peuvent impacter le scénario de test sont exactement les mêmes que celles décrites dans MATeL : résolution d'écran, spécificités du mobile, plateformes mobiles, modèles de téléphones particuliers et versions de l'application.

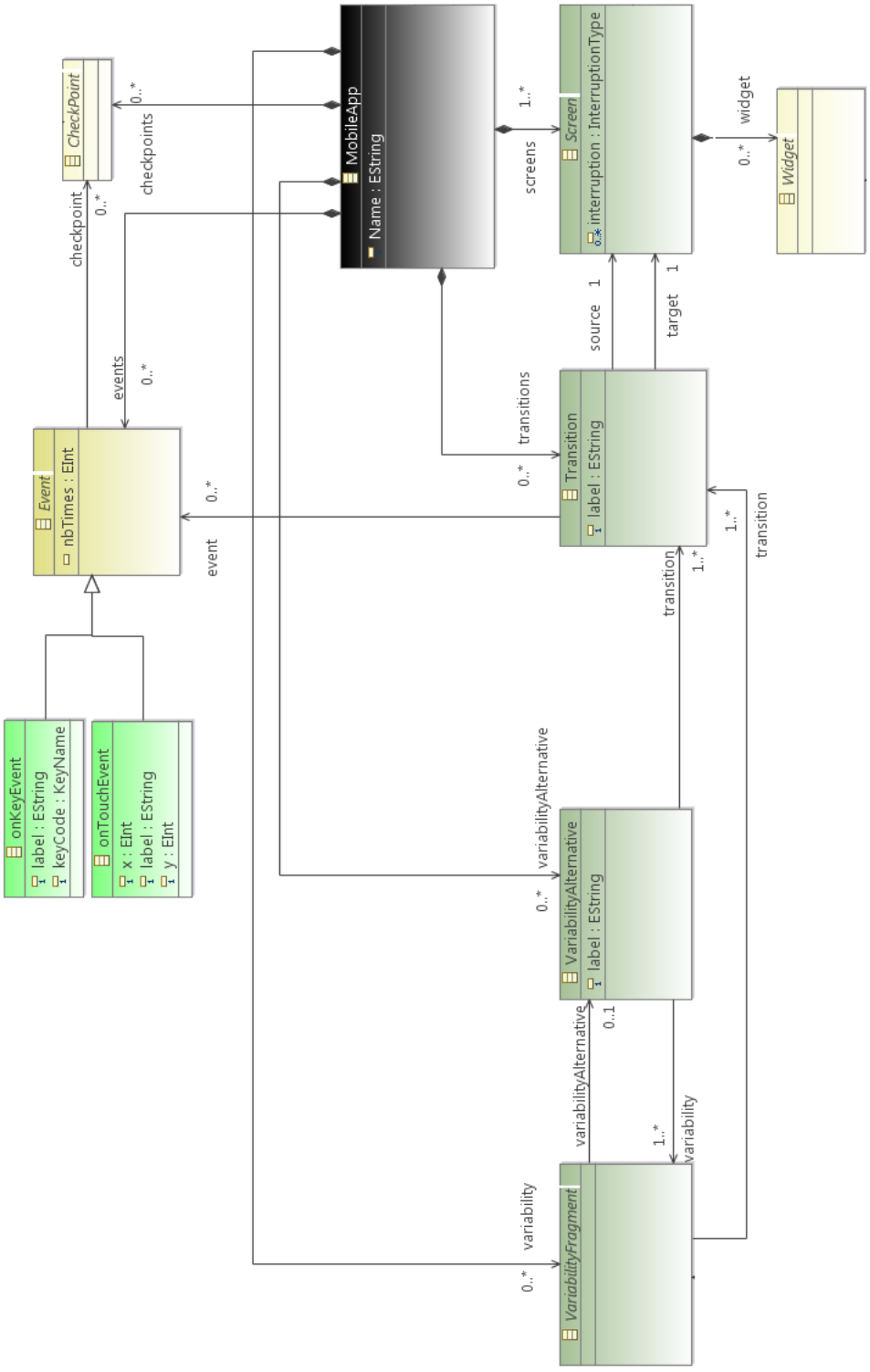


FIGURE 4.14 – Métamodèle MobiAM

Le métamodèle MobiAM offre les concepts suivants :
- ***Screen*** pour ajouter des écrans à l'application ;
- ***Transitions*** - une transition peut être externe (d'un écran à un autre) ou interne (changement dans le même écran). Chaque écran est associé à une ou plusieurs transitions. L'écran peut être la source ou la destination de cette transition ;
- ***Widgets*** - composants graphiques que l'on peut inclure dans un écran tels que des boutons, des checkbox, des zones de texte... ;
- ***Event*** - l'événement déclencheur de la transition. Chaque transition est générée par un événement de type *KeyEvent* ou de type *TouchEvent*. Chaque événement de type *KeyEvent* est attaché à une touche spécifique choisie parmi une liste (LEFT, RIGHT...) ;
- ***VariabilityFragment*** et ***VariabilityFragment***- identiques aux deux métatypes de MATeL ;
- ***Checkpoint*** - un événement peut être attaché à un ou plusieurs points de contrôle. Il s'agit des mêmes métatypes utilisés dans MATeL ;
- ***Interruption*** - le concepteur de l'application peut associer une interruption à un écran.

Grâce à ces différents métatypes, l'utilisateur est capable de modéliser son application et de spécifier la variabilité qui affectera le test.

4.4.0.4 Gestion de la variabilité dans MobiAM

La variabilité exprimée dans l'application permet de générer un scénario de test qui prend en compte cette même variabilité. Celle-ci peut être attachée soit aux écrans, soit aux transitions, soit aux événements. Nous avons décidé de l'attacher aux transitions. Ce choix est justifié par les raisons suivantes :

- si un écran est variable alors la transition l'est aussi. Exemple : si un écran X est présent uniquement sur certains mobiles, ceci veut dire que la transition dont la cible est cet écran est variable à son tour ;
- si un événement est variable alors la transition l'est aussi. Pour passer d'un écran X à un écran Y, l'événement peut être différent selon le mobile. Exemple : appuyer sur une touche ou cliquer sur l'écran s'il est tactile.

La déduction de la variation sur un écran ou un événement est implicite si la transition à laquelle ils s'attachent est variable. Au contraire, une variation sur un événement ne permet pas de déduire une variation sur un écran ou sur une transition. De la même manière, une variation sur un écran ne permet pas de déduire une variation sur un événement (ou une transition). Ainsi, en ajoutant une variabilité à une transition, l'événèment la génèrant est variable et donc l'écran cible l'est aussi. Pour générer un scénario MATeL à partir d'une architecture MobiAM, nous scrutons toutes les transitions sortantes d'un écran. S'il y a des points de variation sur certaines transitions alors nous ajoutons un point de variabilité de ce type dans le test et nous lui associons l'événement de cette

transition. S'il y a plusieurs transitions différentes avec le même événement qui aboutissent sur différents écrans alors il suffit de créer un seul point de variabilité auquel seront associées toutes les transitions.

Si l'écran est différent d'un téléphone à un autre alors nous créons des écrans différents et nous mettons des points de variabilité sur les différentes transitions. Dans le cas contraire, *i.e.* si nous avons le même écran mais des transitions différentes (événements différents) nous créons alors un écran unique, une transition unique et nous mettons le point de variabilité sur cette transition. Le métatype *VariabilityAlternative* est à utiliser uniquement dans deux cas exclusifs (if/else) : le premier respecte la variabilité et le deuxième respecte la *VariabilityAlternative*.
Le passage d'un modèle MobiAM à un modèle MATeL se fait grâce à une transformation de type *Model 2 Model*.

Indépendamment de MATeL, un tel modeleur d'applications mobiles présente certains avantages :

- fournir une vue globale de l'application (storyboard) ;
- générer automatiquement le scénario de test MATeL (complet ou partiel), ce qui s'avère plus simple et plus rapide que de le construire manuellement ;
- profiter des apports de l'ingénierie des modèles en termes de transformation de modèle en code afin de générer du code mobile multi-plateformes (Java ME, Bada, Android, iPhone...) ;
- intégrer l'ensemble dans un environnement complet (IDE, conception, génération et exécution de test).

Afin de valider notre approche nous avons développé un éditeur élémentaire pour permettre la modélisation d'applications et la génération automatique de scénarios MATeL. Les premiers résultats sont satisfaisants et nous ont permis de valider cette approche.

4.5 Synthèse

Notre objectif est de fournir une méthodologie outillée pour concevoir des scénarios de test pour applications mobiles. Nous avons présenté MATeL, un langage qui permet de décrire de tels scénarios et d'exprimer les points de variabilité qui ont un impact considérable sur la facilitation de la procédure de test. Connecté au banc de test et à la base de données qui lui est associée, MATeL permet aussi l'exécution automatique de tels scénarios. L'interprétation du modèle MATeL est une représentation similaire à une machine à état (FSM pour *Finite State Machine* en anglais). Ainsi, le test est une exécution de tous les chemins possibles. La complexité du test est alors proportionnelle au nombre de chemins à explorer. Le concepteur du test n'est pas obligé de tout tester (perte en productivité) mais il doit définir les éléments importants à vérifier. Il faut essayer d'arriver à un ratio couverture de test / coût optimal.

MATeL permet de modéliser le scénario avec toutes les variabilités fournies. Le banc

de test se charge de l'exécution de tels scénarios (cf. chapitre 5). Ainsi, MATeL aide le testeur manuel à passer du modèle de test pas à pas, avec des points à vérifier et des feuilles de tableau à remplir, à une démarche basée sur une suite logicielle intégrée dans le processus de développement. Nous verrons dans les chapitres suivants la mise en œuvre de notre approche et son intégration dans le cadre d'une solution de *Cloud testing*.

Chapitre 5

Architecture logicielle et matérielle de MATeL

Sommaire

5.1 Introduction 96
5.2 Architecture du banc de test 96
5.3 Editeur MATeL 100
5.3.1 Interface avec le banc de test 100
5.3.2 Métatypes et icônes 103
5.4 Synthèse 106

5.1 Introduction

Comme nous l'avons écrit dans l'introduction générale de ce manuscrit, nos travaux de recherche sont réalisés dans le cadre d'une thèse CIFRE avec la société Neomades. Dans ce chapitre, nous présentons le travail réalisé pour valider MATeL et les outils associés.

Afin de résoudre la problématique d'acquisition des téléphones mobiles et de suivre l'évolution effrénée du marché, certaines entreprises ont lancé des plateformes de test à distance (bancs de test en ligne). L'idée générale est de fournir un service Internet qui permet d'accéder et de manipuler de vrais téléphones mobiles via des services Web. De manière générale, l'approche *Cloud Testing* offre un accès à distance à un grand nombre de téléphones et de tablettes réels connectés à des réseaux de télécommunication du monde entier. Ceci est possible en déployant des bancs de test dans plusieurs pays. La connexion aux réseaux mobiles permet de tester des applications connectées mais également d'analyser le comportement des applications lors de la réception d'événements externes comme les appels entrants, SMS, MMS, etc.

Dans ce chapitre, nous présentons l'architecture globale de la plateforme développée dans le cadre de cette thèse ainsi que son intégration avec MATeL.

5.2 Architecture du banc de test

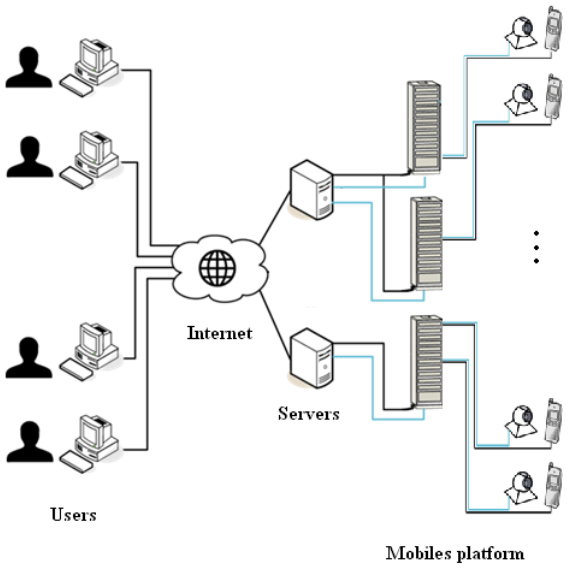


FIGURE 5.1 – Architecture globale d'une plateforme pour le test à distance

La figure 5.1 montre l'architecture globale d'une solution de test en ligne. L'idée est basée sur une infrastructure logicielle dédiée (*middleware*) avec des téléphones réels connectés à des serveurs Web via des bancs de test. Une caméra vidéo est placée en face de chaque appareil pour filmer son écran ; les kits mains-libres des mobiles sont utilisés pour capturer les sorties audio ; les touches sont électroniquement câblées afin de pouvoir effectuer des appuis (cf. figure 5.2). Une application connectée aux serveurs Web permet de se connecter au banc de test et d'utiliser n'importe quel téléphone disponible.

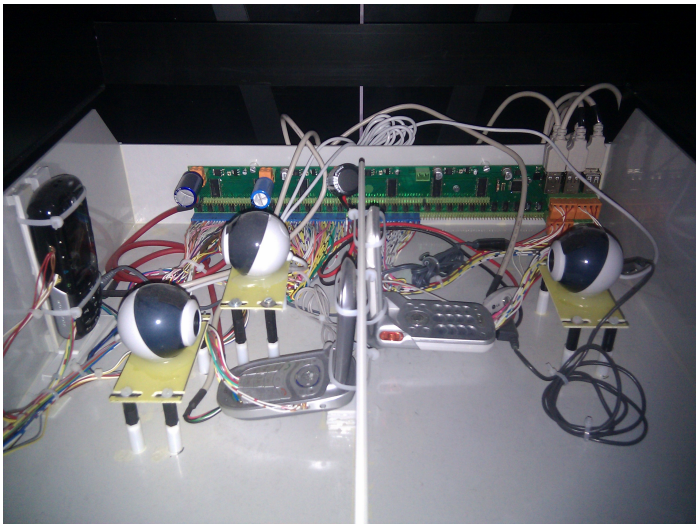


FIGURE 5.2 – Câblage de quelques téléphones

L'architecture matérielle du banc de test comporte certains choix technologiques faits par son constructeur. Dans ce chapitre, nous présenterons le prototype de banc réalisé par Neomades[68] afin de valider MATeL (cf. figure 5.3). Cette armoire contient potentiellement une douzaine de modèles de téléphones différents qui sont électroniquement connectés à une carte de contrôle électronique. Ces téléphones sont branchés électroniquement afin de permettre de gérer :

- **les entrées** - en fonction du téléphone, l'utilisateur peut appuyer sur des touches ou cliquer sur l'écran du téléphone. Il peut aussi utiliser son microphone pour transmettre sa voix sur les kits mains-libres des mobiles ;
- **les sorties** - en retour, le banc de test renvoie la vidéo de la caméra qui filme l'écran du mobile et les sons émis par celui-ci (sonnerie, musique ou encore voix).

Des emplacements de cartes SIM sont aussi disponibles dans ce banc de test afin de pouvoir connecter les mobiles sur différents réseaux mobiles. Le protocole utilisé pour communiquer avec le banc est le standard Modbus[69] RTU RS485, 9600 bauds, 8 bits,

68. www.neomades.com
69. http ://www.modbus.org/

FIGURE 5.3 – Banc de test développé par Neomades

parité paire, 1 start, 1 stop. Ce protocole passe par une connexion série et permet de coupler un téléphone et une carte SIM puis d'envoyer des commandes à ce téléphone. Par exemple pour mettre sous tension le téléphone $n°21$, il faut envoyer la commande hexadécimale ModBus décrite ci-dessous.

Listing 5.1 – Exemple de commande Modbus

```
1 15 06 01 00 00 01 4A E2
```

Le tableau 5.1 décrit la signification de chaque valeur.

TABLE 5.1 – Détails de la commande Modbus

Code	**Taille**	**Signification**
0x15	1 octet	Adresse Modbus du téléphone $n°21$
0x06	1 octet	Code de la fonction de mise sous tension
0x00FF	2 octets	Adresse du registre
0x0001	2 octets	Valeur à envoyer (1 = ON, 0 = OFF)
0x4A 0x00E2	2 octets	CRC (Contrôle de redondance cyclique)

L'armoire fournit en sortie deux câbles USB, l'un pour la vidéo, l'autre pour l'audio, ainsi qu'un câble série RS32 pour l'envoi des commandes et la récupération des codes d'exécution. Afin de pouvoir exploiter cette armoire, nous avons développé un *middleware* qui permet de faire abstraction des commandes ModBus et d'utiliser les connexions USB et série. Ce middleware est hébergé sur un serveur auquel nous avons connecté le banc de test. Ce serveur est accessible à distance via la technologie Java RMI [70].

La figure 5.4 montre le rôle du middleware et son mode de fonctionnement.

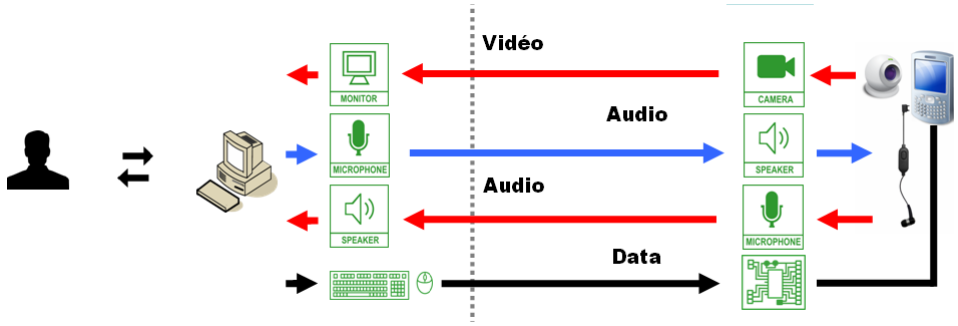


FIGURE 5.4 – Architecture globale du banc de test

Un serveur de streaming temps réel a été implémenté sur le *middleware*. Ce serveur envoie en temps réel la vidéo (capturée via la webcam) et l'audio (capturé via le kit mains-libres). Nous avons aussi développé une application PC standard qui permet à l'utilisateur de se connecter au banc via le serveur. Une fois la connexion au serveur du banc de test établie, cette application affiche à l'utilisateur le téléphone mobile et joue, en temps réel, l'audio et la vidéo transférés à partir du banc de test. L'utilisateur est capable d'envoyer des commandes (manipuler le clavier, toucher un écran tactile, brancher/débrancher le chargeur ou un câble USB...) vers le téléphone mobile en utilisant la souris et le clavier de son ordinateur.

En fournissant une telle architecture, l'utilisateur du banc de test est capable d'utiliser le téléphone comme s'il l'avait entre les mains. Aujourd'hui, avec les nouveaux systèmes d'exploitation mobiles, on peut tout à fait profiter des possibilités qu'offre la virtualisation pour pouvoir manipuler les téléphones directement en utilisant des câbles USB, sans passer par des branchements éléctroniques.

70. Remote Method Invocation : http ://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

5.3 Editeur MATeL

Nous avons développé un éditeur graphique pour la modélisation de scénarios MATeL. Cet éditeur est basé sur Eclipse (cf. figure 5.5). Ainsi il permet à l'utilisateur de bénéficier de tous les avantages de cet environnement mais il lui permet aussi de bénéficier des fonctionnalités suivantes :

- un environnement complet qui permet de dessiner un scénario en utilisant le « drag & drop » de tous les métatypes disponibles ;
- un système de validation de la conformité des modèles dessinés par rapport au métamodèle et aux règles OCL exprimées ;
- une vue qui permet d'afficher et de qualifier les résultats du test ;
- un ensemble d'IHM pour la configuration du test et de son exécution automatique sur le banc de test.

5.3.1 Interface avec le banc de test

Afin de connecter l'éditeur MATeL au banc de test, nous avons développé un *plugin* Eclipse qui permet à l'utilisateur de valider son scénario, de sélectionner les téléphones disponibles, de se connecter au banc de test et de lancer l'exécution de son scénario.

Quelle que soit la technique mise en œuvre pour passer du modèle d'un système à une application logicielle exécutable, il est nécessaire que la sémantique du modèle à exécuter soit clairement définie de manière unique, formelle et surtout non ambiguë. De plus la richesse d'expression fournie à l'utilisateur pour décrire un modèle (des modèles complets et exhaustifs ou encore des éléments nécessaires à l'exécution de celui-ci) va influer directement le niveau d'exécution atteignable par le scénario.

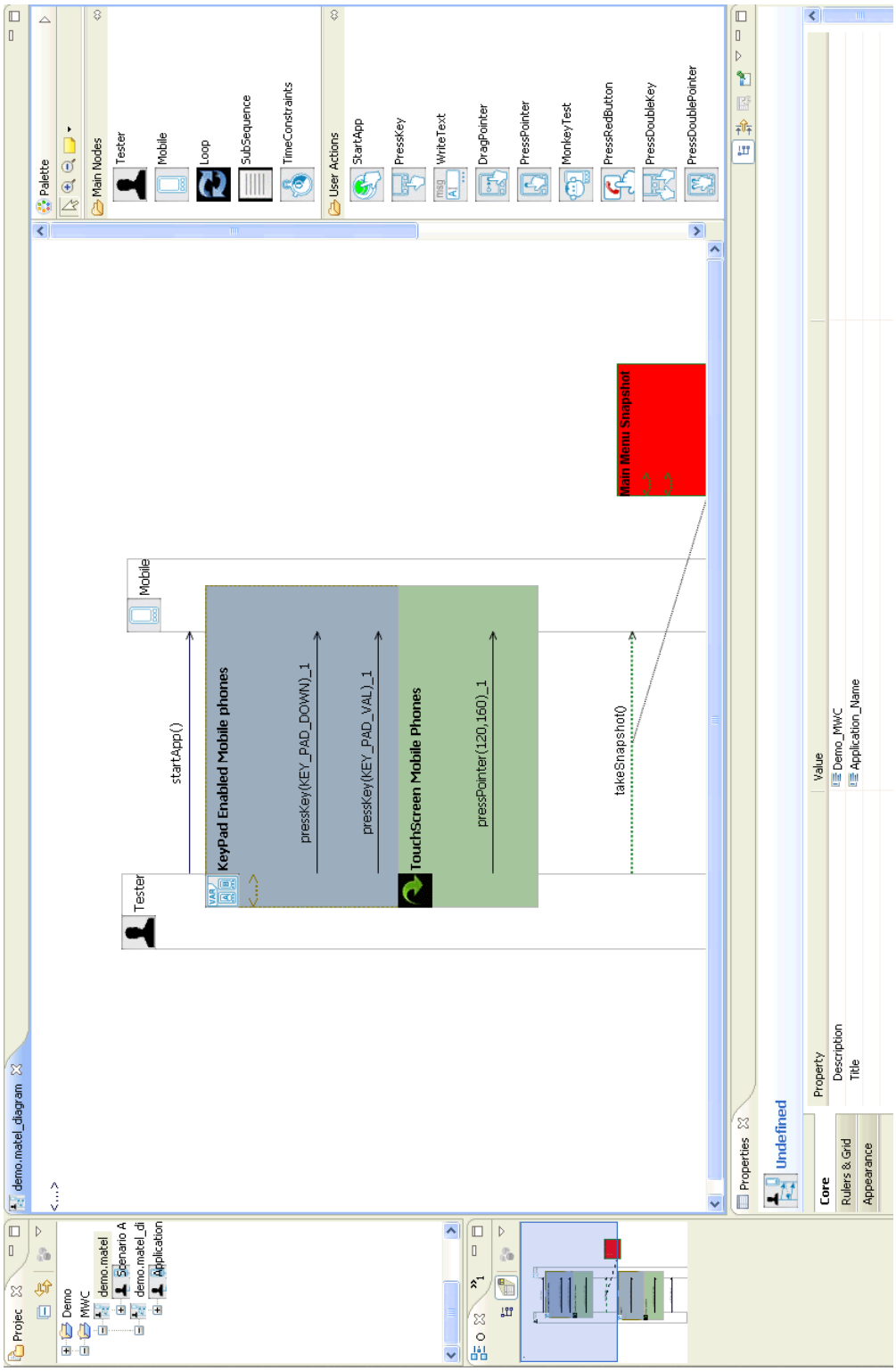


FIGURE 5.5 – Editeur MATeL

Dans notre cas, il s'agit d'une transformation qui permettra le passage d'un modèle décrit avec le langage MATeL à un langage exécutable compréhensible par le banc de test. Le *middleware* pilotant le banc de test a été développé en Java. Afin de pouvoir exécuter les modèles MATeL, nous devons les transformer en programme Java en s'appuyant sur les APIs du banc. Il s'agit ici d'une transformation où les métamodèles cible et source n'appartiennent pas au même espace technique. En effet, à partir d'un modèle conforme à un métamodèle décrit par le méta-méta-langage Ecore, il faut générer du code dont la syntaxe de la grammaire est décrite grâce à l'EBNF (Extended Backus-Naur Form). De nombreux langages sont à ce jour disponibles pour écrire des transformations de modèle. On retrouve les langages qui s'appuient directement sur la représentation abstraite du modèle. On citera par exemple l'API EMF [SBPM09] qui, couplée au langage Java, permet de manipuler un modèle. Dans ce cas, la charge reviendra au programmeur de faire la recherche d'information dans le modèle, d'expliciter l'ordre d'application des règles, de gérer les éléments cibles construits, etc. Afin de mieux coller aux limitations et aux interfaces fournies par notre banc de test, nous avons opté pour cette solution, *i.e.* à partir d'un modèle MATeL, nous fabriquons plusieurs scénarios exécutables sur le banc en prenant en compte les points de variabilité définis par le concepteur du scénario. Cette transformation est réalisée par le « *derivation engine* » (cf. figure 8.2) qui, en se connectant à une base de données, récupère toutes les informations nécessaires sur tous les téléphones du banc et peut ainsi adapter le scénario MATeL générique en plusieurs scénarios spécifiques à un ou plusieurs mobiles du banc.

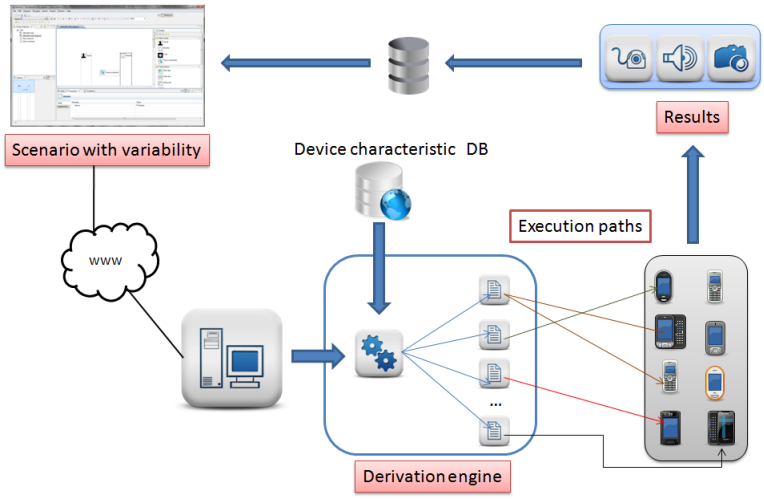


FIGURE 5.6 – Du modèle au test effectif

L'éditeur MATeL dispose d'une vue Eclipse qui, à la fin de l'exécution du test, affiche tous les résultats obtenus (Snapshot...). Ainsi le concepteur du scénario peut valider les résultats obtenus en les comparant aux résultats attendus.

5.3.2 Métatypes et icônes

Le tableau 5.2 présente tous les métatypes manipulables par le concepteur du test, la relation que chacun représente ainsi que le nombre d'instances par modèle. Voici quelques exemples.

Call - En pratique ce métatype est un message entre deux mobiles. Le concepteur du scénario peut en insérer un ou plusieurs en utilisant l'éditeur MATeL. Lors de l'exécution du scénario, ce message déclenchera un appel entrant pour le mobile sous test ;

Mobile - Il est possible d'avoir un ou deux mobiles par scénario de test. L'un est obligatoire et représente les mobiles en test. L'autre permet de générer des interruptions de type « appel entrant » ;

WriteText - Ce métatype est un message dont la source est un testeur et la cible est un mobile. Le concepteur du test doit fournir, en paramètre (attribut *Content*), le texte à écrire (cf. figure 5.7).

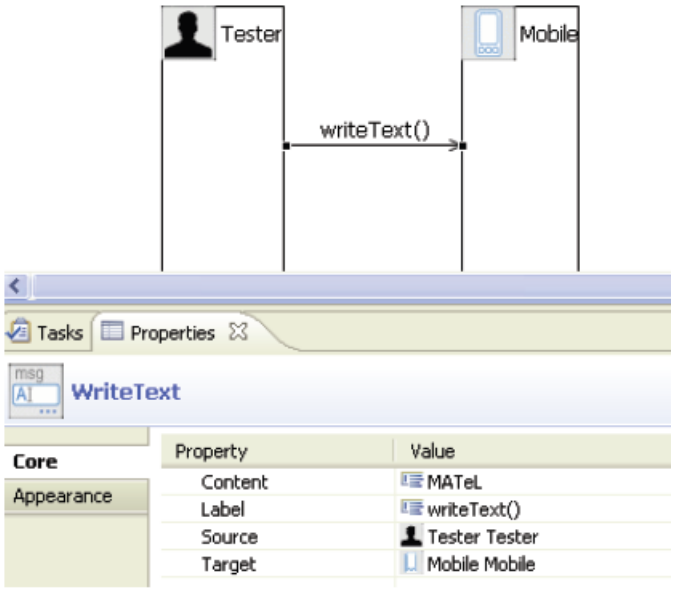


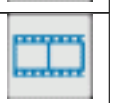
FIGURE 5.7 – Paramètres du métatypes *WriteText*

Le "-" dans la colonne « Relation » du tableau 5.2 signifie que le métatype n'est pas une relation entre deux métatypes mais un noeud dans le modèle.

TABLE 5.2 – Différents métatypes utilisables dans l'éditeur MATeL

Métatype	**Relation**	**Nombre par scénario**	**Icône**
AcceptCall	Mobile - Mobile	0..*	
AudioSequence	-	0..*	
Call	Mobile - Mobile	0..*	
CloseClamshell	Testeur - Mobile	0..*	
ConnectBluetooth	Mobile - Mobile	0..*	
ConnectCharger	Testeur - Mobile	0..*	
DisconnectCharger	Testeur - Mobile	0..*	
DragPointer	Testeur - Mobile	0..*	
EndCall	Mobile - Mobile	0..*	
Flip	Testeur - Mobile	0..*	
Mobile	-	1..2	
MobileModels	-	0..*	
MobileSpecifications	-	0..*	

MonkeyTest	Testeur - Mobile	0..*	
OpenClamshell	Testeur - Mobile	0..*	
PressDoubleKey	Testeur - Mobile	0..*	
PressDoublePointer	Testeur - Mobile	0..*	
PressKey	Testeur - Mobile	0..*	
PressPointer	Testeur - Mobile	0..*	
PressRedButton	Testeur - Mobile	0..*	
RecordAudio	Testeur - Mobile	0..*	
RecordVideo	Testeur - Mobile	0..*	
RejectCall	Mobile - Mobile	0..*	
SendMMS	Mobile - Mobile	0..*	
SendSMS	Mobile - Mobile	0..*	
SetApplicationConnectivity	Testeur - Mobile	0..*	
SetDateAndTime	Testeur - Mobile	0..*	
SetGPRS	Testeur - Mobile	0..*	

SetWifi	Testeur - Mobile	0..*	
Snapshot	-	0..*	
StartApp	Testeur - Mobile	0..*	
StartCall	Mobile - Mobile	0..*	
TakeSnapshot	Testeur - Mobile	0..*	
Tester	-	1	
VideoSequence	-	0..*	
WriteText	Testeur - Mobile	0..*	

5.4 Synthèse

Dans ce chapitre, nous avons présenté l'éditeur MATeL et le prototype du banc de test en ligne. La mise en place de ce dernier a eu lieu pendant la première année de thèse. Ce travail s'est fait en collaboration avec des électroniciens qui ont fabriqué l'armoire. Nous avons implémenté toute la partie logicielle pour y accèder à distance, la commander via la connexion série (RS 32) et enfin récupérer et transférer en temps réel la vidéo et l'audio. Ce dernier point a été particulièrement compliqué à gérer, vu qu'il fallait d'abord contrôler la caméra et ensuite transférer le flux capturé. La première difficulté était liée au pilote logiciel de la webcam qui n'était pas fourni pour notre type de serveur. Il nous a fallu effectuer plusieurs tests afin de trouver une webcam (avec un rapport qualité/prix acceptable) dont le driver est disponible. La deuxième difficulté était quant à elle liée au transfert en temps réel des flux vidéo et audio. Effectivement, plusieurs implémentations ont abouti à des résultats insatisfaisants en termes de rapidité et de temps de réponse (entre 3 et 5 secondes de latence entre la vidéo capturée et la vidéo reçue dans un réseau local). Ceci n'est évidemment pas acceptable pour pouvoir manipuler un mobile. Nous avons tout de même réussi à finir ce prototype et à établir le lien entre l'éditeur MATeL et ce banc de test. Ceci nous a permis de valider notre approche et de mettre le doigt sur certaines limitations que nous détaillons dans le chapitre suivant.

Chapitre 6

Evaluation

Sommaire

6.1 Introduction 108
6.2 Etude de cas 108
6.2.1 Description de l'application 108
6.2.2 Description des mobiles ciblés 109
6.2.3 Description du scénario MATeL 110
6.2.4 Gestion des résultats de test 114
6.3 Apports et limitations de MATeL 116
6.3.1 Apports 116
6.3.2 Limitations 117
6.4 Synthèse 118

6.1 Introduction

Afin de proposer un environnement de modélisation complet et ainsi faciliter la conception de scénarios MATeL, nous avons développé un ensemble de *plugins* Eclipse. Ces *plugins* aident à la modélisation, à la vérification et à l'exécution de scénarios MATeL sur le banc de test en ligne. Ils facilitent aussi la visualisation et la vérification des résultats de test. MATeL peut être considéré à la fois comme un langage de modélisation de test outillé mais aussi comme une méthodologie rigoureuse de conception de scénarios exécutables. Effectivement, un scénario MATeL doit être conforme aux règles de construction que nous avons définies dans le métamodèle à la fois au niveau des métatypes et de leurs relations ainsi qu'au niveau des règles OCL.

Dans ce chapitre nous présentons un exemple concret de la mise en œuvre de MATeL et du banc de test. Nous mettons aussi en évidence les apports de notre approche ainsi que ses limitations.

6.2 Etude de cas

Nous présentons dans cette section l'application mobile à tester, les mobiles ciblés ainsi que le scénario de test défini.

6.2.1 Description de l'application

Nous avons développé une application Java ME compatible avec les téléphones que nous ciblons. Nous nous limitions à 2 téléphones à cause de contraintes techniques liées au prototype du banc de test en ligne.

L'application est composée de trois écrans (cf. figure 6.1 - écran 1, 3 et 4) qui contiennent des boutons, un menu sous forme de liste avec trois entrées et une zone de texte. La figure 6.1 montre les différents écrans de l'application pendant le déroulement du test. Ce dernier consiste à naviguer entre les différents écrans en appuyant sur des boutons, en choisissant une entrée dans le menu et en saisissant un texte dans la zone de texte. Le test comporte les étapes suivantes :

- choisir la deuxième entrée dans le menu ;
- valider ce choix ;
- valider l'écran 3 (bouton OK en bas à gauche de l'écran) ;
- saisir le texte « demo ».

L'application que nous présentons est relativement simple. Néanmoins, elle contient des éléments disponibles dans la majorité des applications mobiles : une liste, une zone de texte et des boutons.

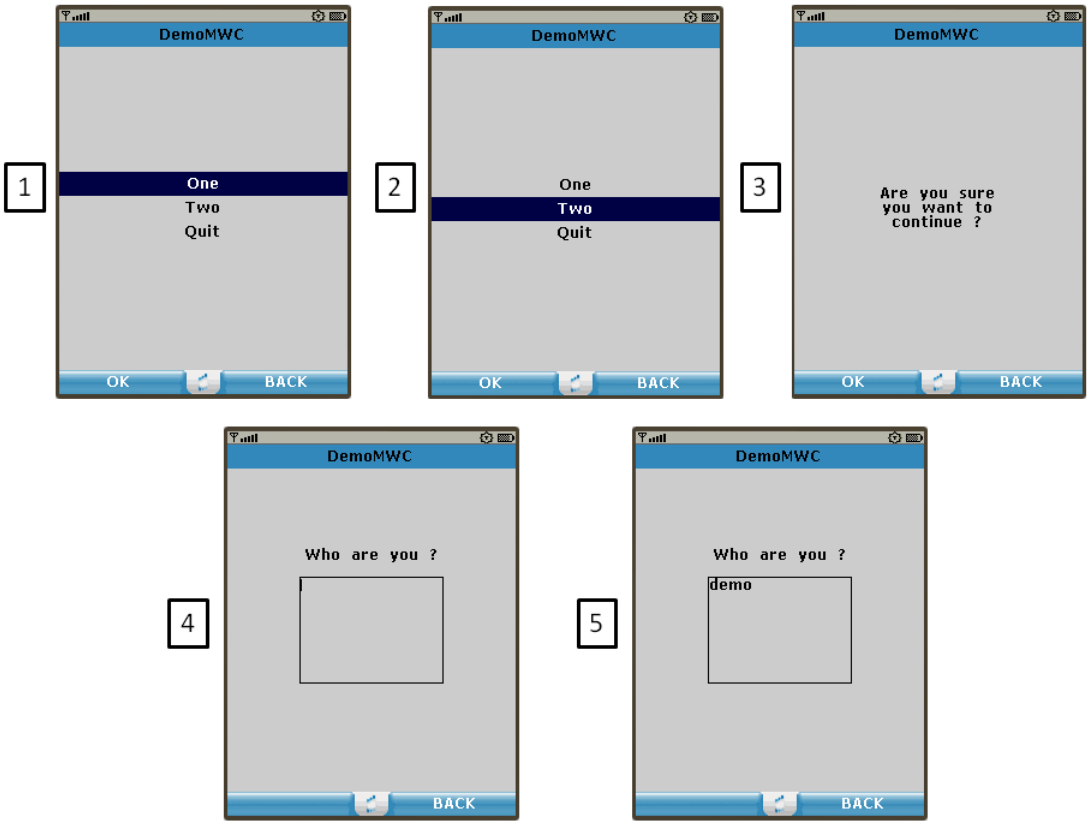


FIGURE 6.1 – Ecrans de l'application à tester

6.2.2 Description des mobiles ciblés

Parmi la douzaine de mobiles disponibles dans notre banc de test, nous avons choisi de tester deux téléphones assez différents l'un de l'autre. Il s'agit du Samsung D800 et du Sony Ericsson P990i. Le tableau 6.1 liste les principales différences entre ces deux modèles. Ces différences ont un impact important sur la description du scénario de test.

TABLE 6.1 – comparaison du Samsung D800 et du Sony Ericsson P990i

Samsung D800	**Sony Ericsson P990i**
	
Ecran classique	Ecran tactile
240 x 320 pixels	240 x 320 (clapet ouvert) et 240 x 256 (clapet fermé)
Keypad numérique coulissant	KeyPad numérique + clavier AZERTY ou QWERTY à clapet

6.2.3 Description du scénario MATeL

La figure 6.2 est une représentation graphique du scénario de test que nous avons conçu pour notre application. Nous avons développé cet éditeur afin de faciliter la création, la visualisation et l'édition de scénarios MATeL.

Le scénario contient deux lignes verticales pour représenter d'une part le testeur (*Tester*) et d'une autre part les mobiles en test (*Mobile*), à savoir le Samsung D800 et le Sony Ericsson P990i. Ces deux lignes verticales représentent aussi le temps de haut en bas. Le scénario contient deux types d'instructions envoyées du testeur vers les mobiles. Le premier type contient tous les messages communs aux deux mobiles, le second contient les messages propres soit au Samsung D800 soit au Sony Ericsson P990i (rectangles bleu et vert).

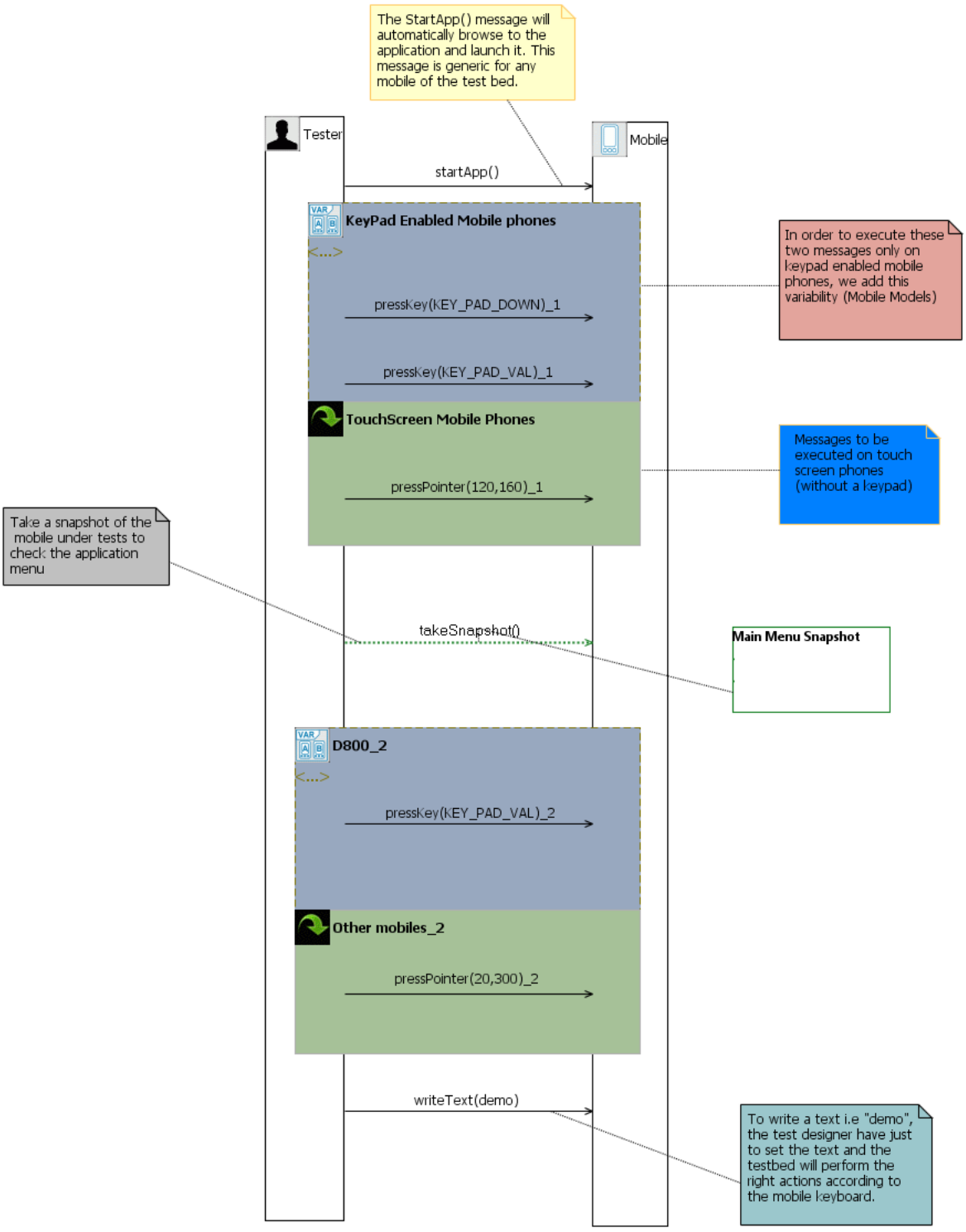


FIGURE 6.2 – Exemple de scénario de test

6.2.3.1 Modélisation du scénario

En utilisant le modeleur MATeL, la conception du scénario se fait principalement en « drag & drop ». Le concepteur du test ajoute les composantes de son test à partir d'une liste prédéfinie. Ce scénario commence par un message de type *startApp()*, cette fonction explore le système de fichiers de chaque mobile pour trouver l'application installée et la démarrer. Ce message est commun aux deux téléphones, c'est le banc de test qui se charge d'effectuer les bonnes actions sur chacun des mobiles.

Nous avons défini, par la suite, un point de variabilité appelé *KeyPad Enabled Mobile Phones*, cette variabilité est de type *MobileModels* (cf. figure 4.10) c'est-à-dire que le concepteur doit spécifier les modèles de mobiles concernés. Dans notre exemple, il s'agit du Samsung D800. A cette variabilité, nous affectons les deux messages « pressKey(KEY_PAD_DOWN)_1 » et « pressKey(KEY_PAD_VAL)_1 ». Ils correspondent respectivement à un appui sur la touche directionnelle « bas » et la touche centrale de validation (cf. tableau 6.1). En effet, pour ce test, nous avons choisi de sélectionner la deuxième entrée du menu (cf. figure 6.1). Au lancement de l'application, le curseur est positionné sur la première entrée du menu. En appuyant une fois sur la touche « bas », le curseur se déplace sur la deuxième entrée (écran 2 de la figure 6.1). Enfin, en appuyant sur la touche de validation, l'application passe de l'écran 2 à l'écran 3 (cf. figure 6.1).

Afin de prendre en compte dans le test les spécificités des téléphones tactiles, en l'occurence le Sony Ericsson P990i, une instance de *VariabilityAlternative* appelée *TouchScreen Mobile Phones* est attachée à la variabilité *KeyPad Enabled Mobile Phones*. Le message « pressPointer(120,160)_1 » a été attaché à cette *VariabilityAlternative*. Ce message sera exécuté sur tous les téléphones sélectionnés pour le test et ne faisant pas partie de la variabilité *KeyPad Enabled Mobile Phones*. Dans notre cas, cela sera le Sony Ericsson P990i. Ce message permet à l'application de passer de l'écran 1 à l'écran 3 de l'application (cf. figure 6.1). Etant donné que ce mobile dispose d'un écran tactile, il est possible de choisir directement l'entrée souhaitée du menu. A ce stade du test, les deux mobiles se trouvent exactement au même écran (écran 3).

Une deuxième variabilité appelée *D800_2*, est introduite, à laquelle la variabilité *VariabilityAlternative Other mobiles_2* a été attachée. Celle-ci sert à valider l'écran 3 (cf. figure 6.1) de l'application. Enfin, le scénario se termine par l'exécution de l'instruction « WriteText(demo) ». Cette fonction écrira le texte “demo” dans la zone de texte. En utilisant une base de données, le banc de test est capable de faire la correspondance entre la lettre à écrire et la touche sur laquelle il faut appuyer (cf. dernier écran de la figure 6.1).

6.2.3.2 Validation du scénario

Un scénario MATeL est un modèle qui doit être conforme à son métamodèle. Chaque métatype (testeur, mobile, message...) a une liste d'attributs (code de touche, source,

cible...) ; tous ces attributs doivent être renseignés via l'éditeur. Dans le cas où certains attributs ne seraient pas renseignés ou mal renseignés, l'éditeur avertira l'utilisateur lors de la validation du scénario et affichera en rouge les instances de métatypes à corriger. Cette validation est basée d'une part sur la vérification de la conformité du modèle au métamodèle (exemple : la source d'un message est un testeur, sa cible est un mobile (cf. figure 4.1)) et d'autre part sur le respect des règles OCL définies.

6.2.3.3 Exécution du scénario

Une fois la connexion entre l'éditeur MATeL et le banc de test établie, l'exécution du scénario démarre. En pratique, le banc de test a généré, à partir du scénario MATeL modélisé, deux scénarios différents pour chaque mobile en prenant en compte les variabilités exprimées (cf. listing 6.1 et 6.2). Ces scénarios sont, par la suite, traduits en commandes « bas niveau » pour commander le banc de test.

Listing 6.1 – Scénario dérivé pour le Samsung D800

```
1 startApp()
2 pressKey(KEY_PAD_DOWN)_1
3 pressKey(KEY_PAD_VAL)_1
4 takeSnapshot()
5 pressKey(KEY_PAD_VAL)_2
6 writeText(demo)
```

Listing 6.2 – Scénario dérivé pour le Sony Ericsson P990i

```
1 startApp()
2 pressPointer(120,160)_1
3 takeSnapshot()
4 pressPointer(20,300)_2
5 writeText(demo)
```

La dernière instruction des deux scénarios (*writeText(demo)*) est traduite en une suite d'actions de type pressKey() en utilisant les différents codes de touche pour écrire la chaîne de caractère « demo ». Pour le Sony Ericsson P990i disposant d'un clavier Azerty, les lettres seront directement accessibles via des touches uniques. En revanche, pour le Samsung D800, les lettres « d » et « e » se trouvent toutes les deux sur la même touche numéro 3, de même, les lettres « m » et « o » se trouvent sur la même touche numéro 6 (cf. tableau 6.1). Afin de pouvoir produire toutes ces lettres, le banc de test nous permet de spécifier la durée d'appui (en millisecondes) pour chaque *pressKey()*. Ainsi, il suffit d'envoyer 2 appuis rapides sur la touche 3 du Samsung D800 pour obtenir la lettre « e » et 3 appuis rapides sur la touche 6 pour obtenir la lettre « n ».

Il est évident que notre scénario de test peut être exécuté sur d'autres mobiles, notamment ceux dont les caractéristiques sont similaires aux deux téléphones sélectionnés pour l'illustration.

6.2.4 Gestion des résultats de test

Un message de type *TakeSnapshot* est ajouté au milieu du scénario. Ce message est attaché à une instance de type *Snapshot* appelée « Main Menu Snapshot ». Ce message permet des captures d'écran des deux téléphones en test après avoir exécuté les commandes qui le précèdent. Avant de lancer l'exécution du scénario, grâce à une vue Eclipse spécifique (cf. figure 6.3), l'utilisateur peut choisir à partir du système de fichiers de son ordinateur une image qui correspond au résultat attendu. A la fin de l'exécution du test, la même vue Eclipse affiche à la fois le résultat attendu et le résultat obtenu retourné par le banc de test. Charge au testeur, par la suite, de valider ou pas le résultat grâce à l'attribut *isValid* du métatype *Snapshot*. Si le résultat est validé alors le rectangle représentant l'instance de *Snapshot* change de couleur et devient vert sinon il devient rouge. Ce changement de couleur permet au testeur d'avoir une vue globale sur les parties validées ou non de son application.

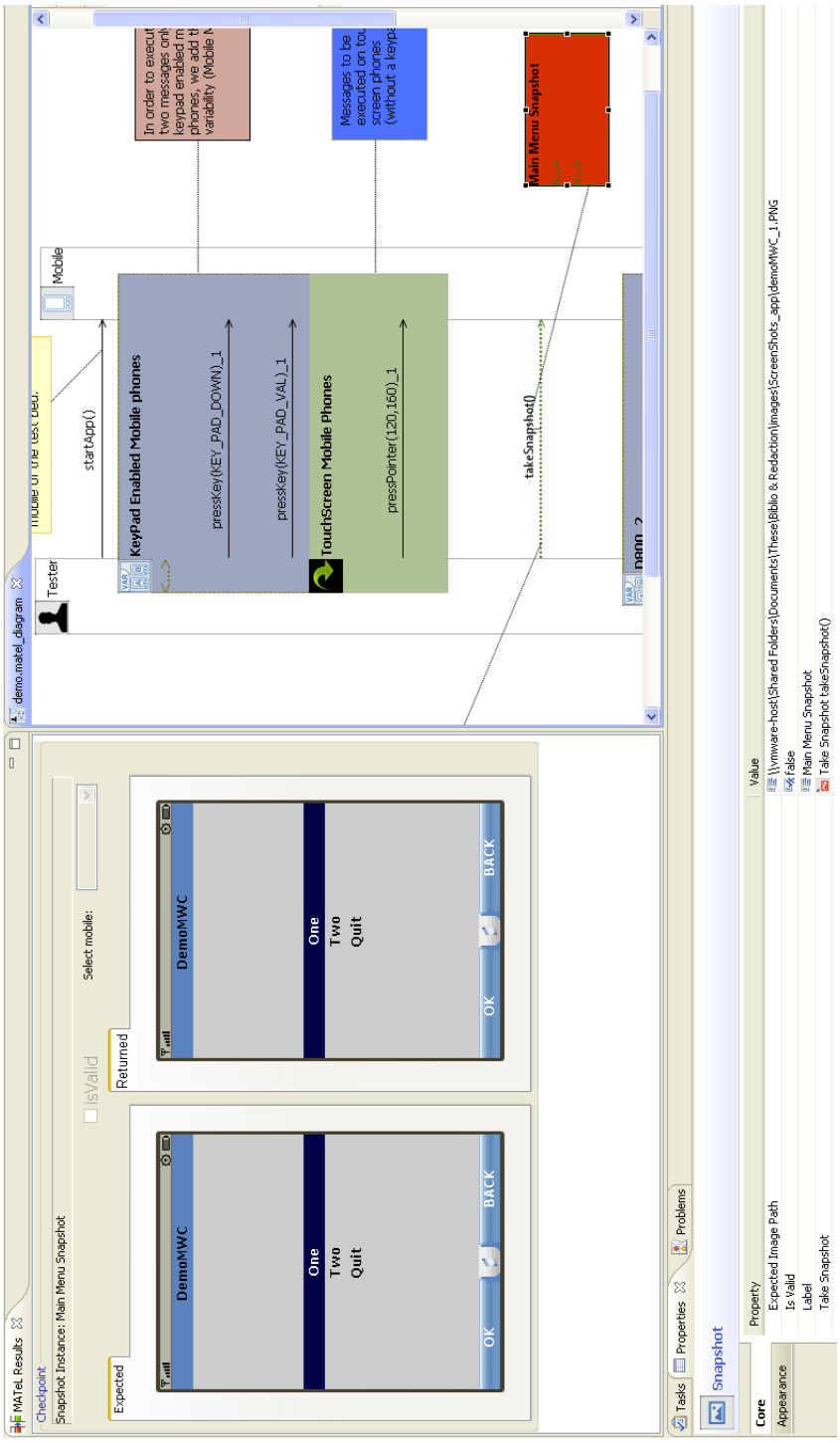


FIGURE 6.3 – Vue Eclipse pour la gestion des résultats

6.3 Apports et limitations de MATeL

6.3.1 Apports

Les plateformes de *Cloud Testing* existantes telles que « Device Anywhere » [71] ou « Perfecto Mobile » [72] permettent aux testeurs d'enregistrer une séquence d'actions en les réalisant sur un téléphone donné. Un scénario basé est alors généré. L'utilisateur ne peut le (re)jouer que sur un téléphone mobile ayant le même type de clavier, la même résolution d'écran, etc. Ceci est justifié par les raisons que nous avons mentionnées dans la première partie, *i.e.* chaque mobile a ses propres caractéristiques matérielles et logicielles. Par exemple, si l'utilisateur enregistre un test sur un téléphone mobile en utilisant les touches de validation gauche et droite, le scénario généré ne fonctionnera pas sur un mobile n'ayant pas ces touches comme par exemple la majorité des mobiles de marque BlackBerry ou encore les téléphones à écran tactile. L'exemple de scénario de test que nous avons étudié confirme cette limitation.

Les principaux points faibles de ces produits industriels sont d'une part l'utilisation de langages propriétaires (loin des standards du marché et des langages de modélisation répandus tel que UML) et d'autre part l'absence de la gestion de la variabilité dans la conception des scénarios alors que le besoin de partager les scénarios entre les mobiles est élevé. C'est principalement en essayant de répondre à ces deux points clés que nous avons conçu MATeL. En effet, l'objectif de MATeL est d'aider le concepteur du test en lui apportant un outil et une méthodologie qui permettent de lancer des campagnes de test à grande échelle et de manière industrielle. L'approche banc de test permet en effet de lancer des exécutions en direct ou planifiées pour une exploitation optimale des téléphones mobiles. L'approche *Cloud Testing* couplée à MATeL permet de réduire les coûts liés à l'achat de matériel (téléphones et tablettes), de réduire les coûts liés à la gestion des équipes de testeurs manuels et des infrastructures nécessaires pour la saisie et le traitements des bugs. L'automatisation de toutes les tâches fastidieuses et répétitives du test permettent aux développeurs d'applications mobiles d'être plus réactifs et de répondre plus rapidement aux besoins et de suivre plus facilement l'évolution du marché. De plus MATeL étant un langage spécifique au test, l'apprentissage est envisageable par des non informaticiens. Ajoutons à cela que l'utilisation de DSMLs basés sur les concepts métiers étroitement liés au domaine d'application est souvent vue comme un gain de 5 à 10 en productivité comparé aux approches classiques [HKO06].

D'après [Hom11], il existe trois phases de test :
- la phase de test technique permet de s'assurer que le système livré fonctionne ;
- la phase de test fonctionnel permet de s'assurer que le système livré correspond aux demandes des utilisateurs ;
- la phase exploitabilité regroupe les tests permettant de s'assurer que le système livré est opérant en conditions normales et reste utilisable en contexte dégradé.

71. www.deviceanywhere.com
72. www.perfectomobile.com

MATeL et le banc de test en ligne permettent de répondre aux première et troisième phases. La deuxième phase quant à elle est réalisable sur simulateur.

Grâce à quelques essais effectués auprès de testeurs d'applications mobiles au sein de la société Neomades, nous avons constaté une première adoption de MATeL principalement pour la simplicité d'utilisation et l'intégration dans l'environnement Eclipse. Nous avons aussi constaté quelques limitations et quelques réticences quant à l'utilisation de telles plateformes.

6.3.2 Limitations

Le débat « automatisable vs. manuel » est complexe et dépend de plusieurs paramètres comme le périmètre de tests que l'on souhaite couvrir et le degré d'automatisation possible en fonction du type de test (fonctionnel, technique, performance...). Certes, l'automatisation permet d'exécuter des tests à un coût marginal faible, après un investissement initial en conception ou en maintenance de scénarios de test. Mais il ne faut pas oublier que les robots de test (le banc de test) ne font jamais de fausse manipulation. Ainsi, ils ne découvrent pas les bugs auxquels nous n'avons pas pensé en avant. Ils ne permettent pas non plus une meilleure vision de la perception client du produit avant commercialisation. Un choix important est donc à faire entre l'adaptabilité de l'humain et la répétabilité de la machine.

Tous les tests ne peuvent pas être automatisés, d'où le besoin d'avoir une équipe de « petites mains » qui exécutera certains tests écrits plus ou moins précisément. Cela est valable pour des interfaces homme-machine (IHM) comme pour des tests de compatibilité avec des environnements extérieurs (Bluetooth, USB, etc.). C'est ce qui est fait par exemple chez les fabricants de téléphones mobiles. En effet, tous les tests ne peuvent pas être conçus avec une précision extrême afin que n'importe quel testeur passant le même test observe exactement le même comportement de la part du mobile. Les tests rédigés permettent de trouver les bugs « classiques », c'est-à-dire ceux que le concepteur du test a envisagés. Toutefois, laisser des marges de liberté au testeur permet aussi de détecter des bugs moins classiques et pas forcément moins critiques.

6.3.2.1 Au niveau de MATeL

Certains tests sont difficilement réalisables sur le banc de test en ligne, notamment ceux qui se basent sur l'utilisation de nouveaux éléments techniques tels que les puces de communication en champs proches ou Near Filed Communication (NFC) en anglais, le GPS, la boussole, l'accéléromètre ou encore la realité augmentée. Ces technologies demandent une interaction physique importante avec le monde extérieur, ce qui n'est pas faisable avec un téléphone fixé dans un banc de test. De la même manière, les jeux basés sur le hasard ne sont pas non plus testables compte tenu que le scénario de test ne s'adapte pas en temps réel à l'évolution dans les niveaux du jeu.

6.3.2.2 Au niveau de l'approche *Cloud Testing*

L'un des problèmes majeurs liés à l'utilisation de l'approche *Cloud Testing* concerne les exceptions qui peuvent avoir lieu pendant l'exécution du test. Ces exceptions sont des erreurs qui se produisent généralement de manière imprévisible, c'est-à-dire suite à une action de l'utilisateur ou de l'environnement (appels entrants, SMS...). Toutes ces exceptions ne peuvent être prévues pendant le développement de l'application. Souvent ces exceptions changent l'état normal du mobile pour le mettre dans un état qui n'a pas été prévu dans le scénario de test (exemple : au lieu de passer d'un écran à un autre, l'application s'arrête inopinément). Ainsi, l'exécution de la suite du test devient inutile voire interminable. Afin d'arrêter le test au cas où ceci se produsait, nous donnons la possibilité au concepteur du scénario de programmer l'arrêt du test au bout de x minutes après son lancement. Par exemple, si, en temps normal, le test dure une dizaine de minutes alors le testeur peut programmer son arrêt au bout de 11 minutes. Cette solution n'est pas optimale étant donné que, pour des tests compliqués et longs, il serait plus intéressant d'arrêter l'exécution du scénario dès qu'une exception est détectée. Ainsi, le testeur peut intervenir plus rapidement et interpréter plus facilement le bug.

Nous avons aussi constaté quelques problèmes liés à l'architecture des bancs de test. Ces derniers étant accessibles via Internet, des problèmes de connexion et/ou de latence peuvent se produire. Ceci complique l'interprétation des tests de performance de l'application. Les problèmes de réseaux peuvent aussi altérer la qualité de la vidéo transmise. Ainsi, il est difficile d'interpréter les résultats obtenus (captures d'écran et séquences vidéo).

Nous pensons que le test basé sur une approche regroupant MATeL et un banc de test en ligne est une approche complémentaire au test manuel classique. Ce dernier doit intervenir en premier sur quelques téléphones afin de détecter les bugs majeurs. Une fois ces téléphones validés, des campagnes de test à grande échelle peuvent être lancées de manière industrielle.

6.4 Synthèse

L'exemple d'application/scénario de test associé que nous avons présenté, montre la mise en œuvre de l'approche que nous avons proposée. Cette approche utilise un DSML pour la description de scénarios de test génériques. Ce DSML basé sur une approche dirigée par les modèles a permis de bénéficier de tous les apports de ce domaine des concepts et des outils. L'utilisation des diagrammes MATeL a montré son intérêt dans la conception des plans de test. Cette utilisation propose en effet au concepteur un niveau d'abstraction et des concepts directement représentatifs de son domaine d'expertise. C'est l'intérêt principal d'une démarche spécifique au domaine, elle permet de porter l'attention du concepteur sur les aspects qui sont particulièrement importants pour le système. Au-delà de ces considérations qui valent pour l'ensemble des approches

spécifiques à un domaine, nous pouvons constater un apport considérable dans la gestion de la variabilité. Grâce au métamodèle MATeL, nous fournissons aux concepteurs des concepts précis, un langage pour décrire, dans un même modèle, les points communs, les points variables mais aussi les résultats de test.

Le développement d'applications mobiles est un domaine assez récent où les bonnes pratiques du génie logiciel ne sont pas toujours appliquées. Dans la majorité des projets mobiles, les développeurs ne donnent pas d'importance à la conception en utilisant des diagrammes UML ou autres types de diagrammes. Le code est au cœur du projet. Ce dernier est dirigé par la conception d'écran. Effectivement, souvent la conception d'une application mobile se résume à la définition de tous les écrans, à la définition des transitions entre ces écrans et enfin à l'écriture du code métier. Il s'agit d'un « développement dirigé par les écrans ».

En effectuant des tests auprès d'informaticiens et de testeurs, nous avons constaté une réticence quant à l'utilisation d'une approche dirigée par les modèles pour modéliser des scénarios de test. Nous pensons que ceci est justifié par l'absence « d'une culture de l'IDM » dans ces équipes. Ajoutons à cela que les approches *Model-Driven Development* ou encore *Model-Based Development* sont relativement jeunes et donc il faut un temps d'adaptation des testeurs pour les adopter.

Quatrième partie

Conclusion et perspectives

Chapitre 7

Conclusions et perspectives

Sommaire

7.1 Bilan 124
7.2 Perspectives 125
7.2.1 Benchmark 125
7.2.2 IDM et Eclipse 125
7.2.3 Métamodèle MATeL 125

7.1 Bilan

Dans cette thèse, nous avons présenté le marché des applications mobiles, sa forte croissance et l'augmentation effrénée du nombre de modèles de téléphones mobiles de nouvelle génération. Ces deux facteurs ont conduit à une montée en complexité et en coût du développement, notamment à cause des coûts de portage et de test. Nous avons aussi présenté des outils permettant de faciliter la phase de développement et de portage et avons exposé la problématique du test. C'est dans cette problématique d'actualité que s'inscrivent nos travaux. Une fois l'application portée et optimisée pours chaque modèle de mobile, elle doit être testée puis validée sur les mobiles cibles.

Nous avons aussi présenté une nouvelle approche basée sur les modèles pour le test d'applications mobiles. Nous avons détaillé un *Domain-Specific Modeling language* "MATeL" qui permet de décrire des modèles de test. L'objectif de ces scénarios est de modéliser la suite d'actions qui manipulent l'application en commençant par son exécution, la navigation dans tous ses menus et sa désinstallation. Nous ne nous intéressons pas à l'implémentation de l'application mais à son fonctionnement du point de vue utilisateur. Il s'agit en effet de vérifier et valider le comportement de l'application dans son environnement d'exécution ; ce type de test est souvent appelé « Acceptance tests » [AD97].

L'utilisation des diagrammes MATeL a montré son intérêt dans la conception de scénarios de test pour applications mobiles, en premier lieu en proposant au concepteur un niveau d'abstraction et des concepts directement représentatifs de son domaine d'expertise, à savoir le test. C'est l'intérêt principal d'une démarche spécifique au domaine : elle permet de porter l'attention du concepteur sur les aspects qui sont particulièrement importants pour le système à tester. Au-delà de ces considérations qui valent pour l'ensemble des approches spécifiques à un domaine, nous pouvons constater des progrès significatifs dans la conception détaillée de scénarios de test qui incluent des points de variabilité. En effet, nos travaux s'articulent autour de la modélisation de la variabilité dans un modèle MATeL et la dérivation de modèles spécifiques. Grâce au métamodèle MATeL nous fournissons aux utilisateurs des concepts précis, un langage pour décrire, dans un même modèle, les parties communes à tous les mobiles et les parties spécifiques à un mobile ou un ensemble de mobiles. Nous avons présenté l'ensemble de ces mécanismes de modélisation dans le chapitre 4.

Un ensemble d'outils a été développé pour faciliter la modélisation de scénarios MATeL. Ces outils sont basés sur la plateforme Eclipse et ses composantes *Eclipse Modeling Framework* et *Eclipse Graphical Framework*. Ces framework permettent d'étendre et d'enrichir le métamodèle MATeL relativement facilement. Un tel environnement rend la représentation graphique des modèles MATeL assez intuitive et donc ne nécessite pas un long processus d'apprentissage par les testeurs concepteurs de scénarios. Notre approche permet aussi de capitaliser le savoir-faire des testeurs en permettant la réutilisation de scénarios MATeL complets ou partiels entre applications.

Nos travaux se sont déroulés dans un contexte industriel (thèse CIFRE) où MATeL a été implémenté au-dessus d'un prototype de banc de test en ligne contenant des téléphones mobiles. Dans le chapitre 5, nous avons présenté ce banc de test ainsi que le processus de dérivation de modèles en se basant sur le banc de test d'une part, sur une base de données contenant toutes les informations sur les mobiles cibles d'autre part. Grâce à cette plateforme matérielle, les modèles MATeL sont automatiquement exécutables sur l'ensemble des téléphones ciblés par le test.

7.2 Perspectives

7.2.1 Benchmark

A cause de quelques limitations techniques liées au prototype de banc de test développé pour valider MATeL, nous n'avons pas pu effectuer de benchmark à grande échelle afin de tester notre approche de manière plus approfondie. Il serait intéressant d'effectuer plusieurs essais sur une plateforme industrielle et auprès de plusieurs testeurs afin d'évaluer MATeL. Ces évaluations nous permettront d'apporter des modifications au niveau du métamodèle afin de l'enrichir et de mieux répondre aux besoins des testeurs.

L'implémentation du modeleur de scénarios MATeL réalisée n'est encore qu'un prototype. Confronter cette implémentation à des études de cas de plus grande taille permettrait de déterminer les parties les plus difficiles à modéliser et sur lesquelles des améliorations seraient souhaitables pour faciliter le passage à l'échelle indstrielle. L'éditeur de modèles MATeL nécessite un travail supplémentaire afin de le rendre plus facile à utiliser.

7.2.2 IDM et Eclipse

Lors de nos travaux, nous avons rencontré quelques difficultés liées à l'utilisation de l'environnement de métamodélisation Eclipse. L'un des problèmes auxquels il faut faire face est lié à la modification des métamodèles. Le développement d'un langage de modélisation et de ses éditeurs implique de nombreuses itérations. Chaque modification du métamodèle liée à l'intégration d'un nouveau concept ou d'une nouvelle relation nécessite la régénération ou la reprise de l'ensemble des outils s'y référant. De même, les modèles précédemment conformes au métamodèle ne peuvent plus être pris en compte par les outils mis à jour pour respecter le nouveau métamodèle. Dans le cadre de nos travaux, notre contribution n'a pas été dans le domaine de l'IDM mais nous pensons que ce point est important à soulever afin que des réponses pratiques soient apportées pour résoudre ses limitations.

7.2.3 Métamodèle MATeL

Souvent les modèles sont utilisés dans le test afin de modéliser le comportement du système à tester. Notre approche est différente étant donné qu'un modèle MATeL est une description des interactions à effectuer pour piloter un mobile et ainsi le tester. Ce

choix est justifié d'une part par le lien étroit entre la plateforme d'exécution (le banc de test) et les particularités de chaque mobile et d'autre part la nature des projets mobiles où l'on remarque notamment l'absence de modélisation. Nous avons présenté, dans le chapitre 4, une approche qui permet de modéliser des applications mobiles et de générer des modèles MATeL grâce aux transformations. Il serait intéressant d'approfondir l'étude d'une telle approche pour faciliter la modélisation d'applications mobiles et la génération automatique de scénarios de test. Le modèle d'application est construit du point de vue utilisateur du système : toutes les interactions que l'utilisateur est susceptible d'effectuer doivent être représentées et ceci à un niveau d'expression relativement bas (appuyer sur des touches...). Dans le cas d'applications compliquées, le modèle de test devient de plus en plus complexe et difficile à comprendre et à faire évoluer, notamment à cause d'un nombre important de variabilités. En modélisant la variabilité au niveau de l'application elle-même, il sera toujours plus facile de comprendre les raisons et motivations à l'origine de chaque point de variabilité. D'autant plus que chaque modification sur l'application impliquera automatiquement un changement dans le scénario. Enfin, ceci permettra aux développeurs d'applications mobiles d'avoir une suite logicielle complète pour la conception, le développement et le test d'applications mobiles multi-plateformes.

Cinquième partie

Annexes

Chapitre 8

Annexes

Sommaire

8.1 Plan de test 130
8.2 Metamodèle complet 134

8.1 Plan de test

Test Case - Technical Tests

Note : All Test Cases are written in English but localized texts must be displayed depending on the language selected

					Ok
					NOk
					No to be tested
A - Incoming call					Results
Steps			Keys	Expected Result	
1	Launch the game		-	-	-
2	Perform an incoming call while **game is loading**		-	The application is suspended and goes into **pause** state	
	a	**Reject** call	Red handset key	Game continues loading normally	
	b	**Accept** call	Green handset key	As soon as you end your call, game **continues loading** normally	
	c	**End call** with second device	Red handset key	Game continues loading normally	
3	Perform an incoming call on **Main Menu**		-	The application is suspended and goes into **pause** state	
	a	**Reject** call	Red handset key	You come back to the **Main Menu**	
	b	**Accept** call	Green handset key	As soon as you end your call, you come back to the **Main Menu**	
	c	**End call** with second device	Red handset key	You come back to the **Main Menu**	
4	Perform an incoming call in **two sub-menus** (options, help, etc.)		-	The application is suspended and goes into **pause** state	
	a	**Reject** call	Red handset key	You come back to the **sub-menu 1**	
	b	**Accept** call	Green handset key	As soon as you end your call, you come back to **sub-menu 1**	
	c	**End call** with second device	Red handset key	You come back to the **sub-menu 1**	
	c	**Reject** call	Red handset key	You come back to the **sub-menu 2**	
	d	**Accept** call	Green handset key	As soon as you end your call, you come back to **sub-menu 2**	
	c	**End call** with second device	Red handset key	You come back to the **sub-menu 2**	
5	Perform an incoming call **in-game**		-	The application is suspended and goes into **pause** state	
	a	**Reject** call	Red handset key	The application presents the user with a **continue option** or **is continued automatically** from the point it was suspended at	
	b	**Accept** call	Green handset key	As soon as you end your call, the application presents the user with a **continue option** or **is continued automatically** from the point it was suspended at	
	c	**End call** with second device	Red handset key	The application presents the user with a **continue option** or **is continued automatically** from the point it was suspended at	
6	Perform an incoming call in-game in **pause menu**		-	The application is suspended and goes into **pause** state	
	a	**Reject** call	Red handset key	You come back to the **pause menu**	

	b	**Accept** call	Green handset key	As soon as you end your call, you come back to the **pause menu**	
	c	**End call** with second device	Red handset key	You come back to the **pause menu**	

B - SMS / MMS				Results
Steps		Keys	Expected Result	
1	Launch the game	-	-	-
2	Send a **SMS / MMS** with other device while game **is loading**	-	You receive a **notification** (sound or icon) that you have received a SMS / MMS and the game **continues normally**	
3	Send a **SMS / MMS** with other device while on **Main Menu**	-	You receive a **notification** (sound or icon) that you have received a SMS / MMS and the game **continues normally**	
4	Send a **SMS / MMS** with other device **in-game**	-	You receive a **notification** (sound or icon) that you have received a SMS / MMS and the game **continues normally**	
4	Send a **SMS / MMS** with other device in-game **in pause menu**	-	You receive a **notification** (sound or icon) that you have received a SMS / MMS and the game **stays on the pause menu**	

C - Red button				Results
Steps		Keys	Expected Result	
1	Launch the game	-	-	-
2	Press **red button** while game is **loading**	Red handset key	Game is **suspended**, and when you resume the application, game **continues loading normally**	
3	Press **red button** while on **Main Menu**	Red handset key	Game is **suspended**, and when you resume the application, you come back to the **Main Menu**	
4	Press **red button in-game**	Red handset key	Game is **suspended** and when you resume the application, it presents the user with a **continue option** or **is continued automatically** from the point it was suspended at	

D - Clam shell				Results
Steps		Keys	Expected Result	
1	Launch the game	-	-	-
2	Close **clam shell** while game is **loading**	-	Game is **suspended**, and when you resume the application, game **continues loading normally**	
3	Close **clam shell** on **Main Menu**	-	Game is **suspended**, and when you resume the application, you come back to the **Main Menu**	
4	Close **clam shell in-game**	-	Game is **suspended** and when you resume the application, it presents the user with a **continue option** or **is continued automatically** from the point it was suspended at	

E - Charging				Results
Steps		Keys	Expected Result	
1	Launch the game **without the charger plugged in**	-	-	-
2	**Start charging** the device while game is **loading**	-	The application does **not display any error message or crash**	
3	**Start charging** the device on **Main Menu**	-	The application does **not display any error message or crash**	

4	**Start charging** the device **in-game**	-	The application does **not display any error message or crash**	
5	**Start charging** the device in-game **in pause menu**	-	The application does **not display any error message or crash**	

F - JAD and Manifest content - All handsets (** *Optional*)				Results
	JAD	**JAR**	**Data type**	
1	**MIDlet-Name**	**MIDlet-Name**	String must be smaller or equal to 16 bytes	
2	**MIDlet-Version**	**MIDlet-Version**	**XX.YY.ZZ** X Y, Z are numbers between 0 and 9	
3	**MIDlet-Vendor**	**MIDlet-Vendor**	**In-Fusio**	
4	**MIDlet-1**	**MIDlet-1**	**Applet Name** (String), **Applet icon** (path/name), **Applet launch Class** (path/name)	
5	**MIDlet-Icon**	**MIDlet-Icon**	Applet icon (path/name)	
6	***MIDlet-Description*** **	***MIDlet-Description***	*String*	
7	***MIDlet Data-Size*** **	***MIDlet Data-Size***	*Integer*	
8	**MIDlet-Jar-Size**		Integer	
9	**MIDlet-Jar-URL**		String	
10		**Manifest-Version**	Constant: **1.0**	
11		**MicroEdition-Configuration**	Constant: **CLDC-1.0**	
12		**MicroEdition-Profile**	Constant: **MIDP1.0** or **MIDP2.0**	
13		**Created - By**	String	

G - JAD and Manifest content - Nokia handset except S60				Results
	JAD	**JAR**	**Data type**	
1	**Nokia-MIDlet-Category**	**Nokia-MIDlet-Category**	Constant: **Game**	

H - JAD and Manifest content - Sharp GX XX *(** Optional)*				Results
	JAD	**JAR**	**Data type**	
1	**MIDlet-Install-Notify**		**http://www.vodafone.net**	
2	**MIDxlet-Network**	**MIDxlet-Network**	**Y** or **N**	
3	***MIDxlet-Application-Resolution*** **	***MIDxlet-Application-Resolution***	***120-240*** *or* ***130-320***	
4	**MIDxlet-API**	**MIDxlet-API**	Sharp GX10: **VSCL-1.0.1** Sharp GX15/GX20/GX25/GX30: **VSCL 1.1.0**	

I - JAD and Manifest content - Panasonic X400 and X60				Results
	JAD	**JAR**	**Data type**	
1	**MIDlet-Install-Notify**		**http://www.vodafone.net**	
2	**MIDxlet-Network**	**MIDxlet-Network**	**Y** or **N**	
3	**MIDxlet-API**	**MIDxlet-API**	**VSCL-1.0.1**	

8.2 Metamodèle complet

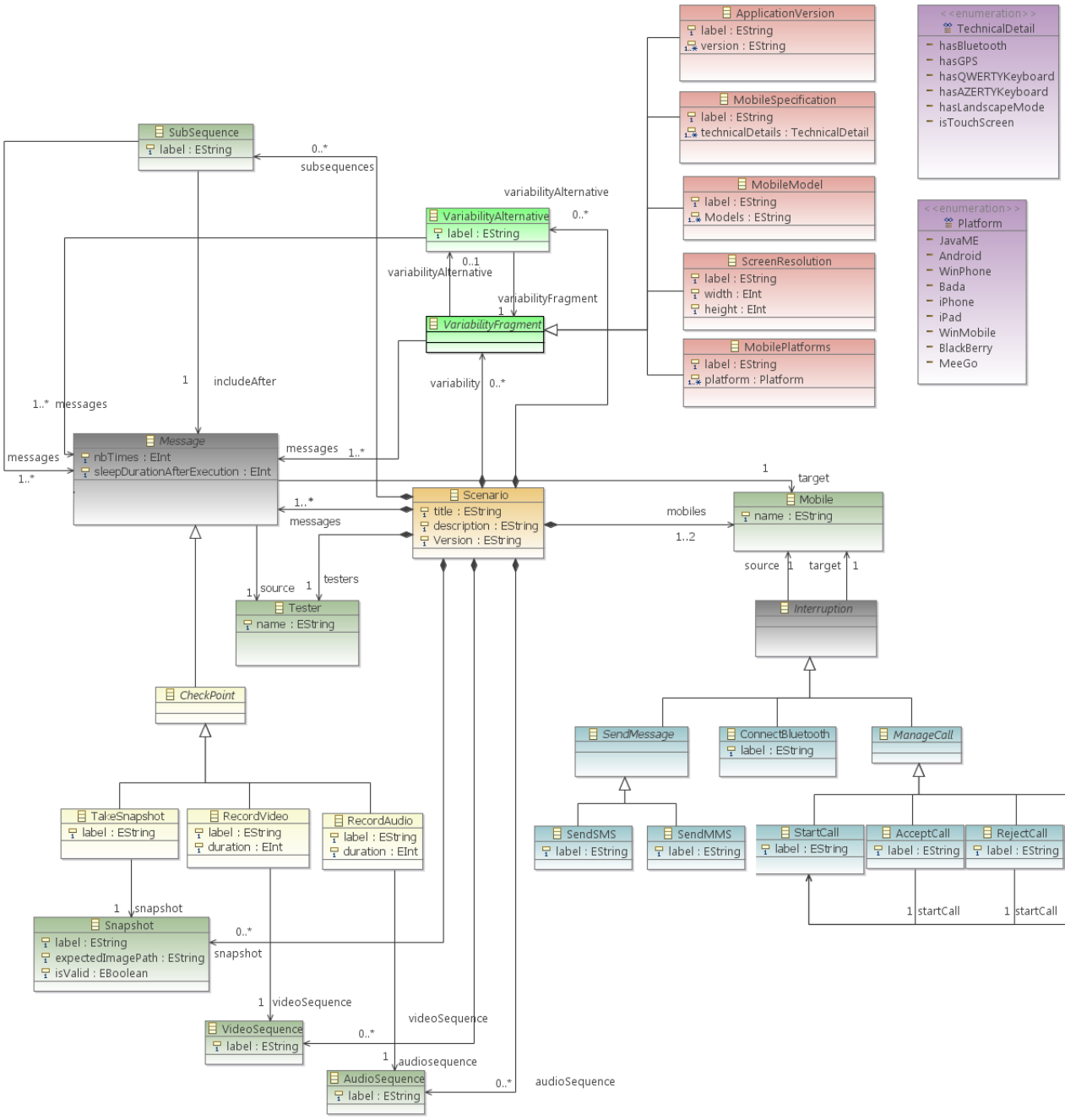


FIGURE 8.1 – Partie 1 du métamodèle MATeL

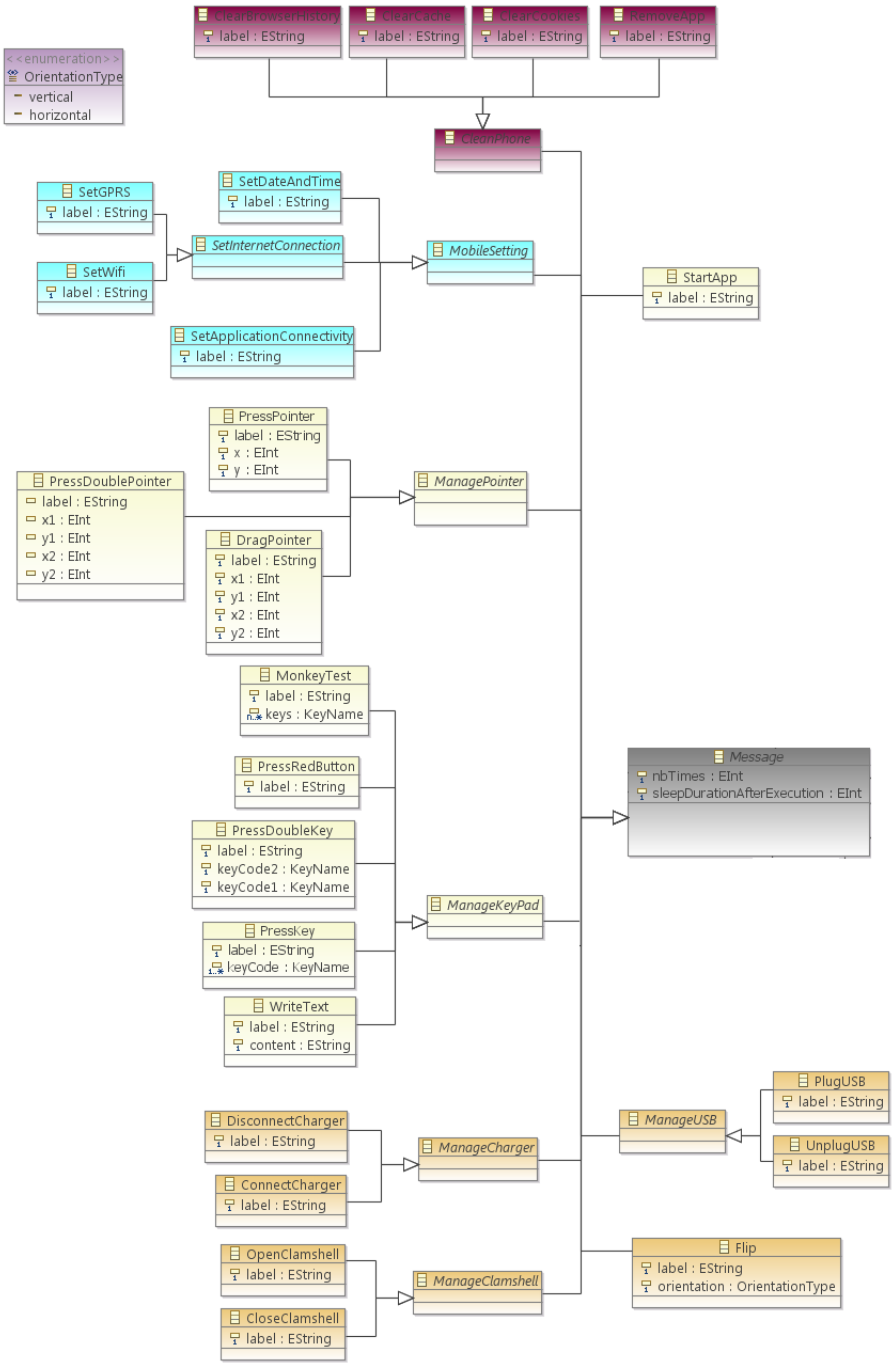


FIGURE 8.2 – Partie 2 du métamodèle MATeL

Bibliographie

[ACL+11] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Variability from Requirements to Runtime. In *16th International Conference on Engineering of Complex Computer Systems(ICECCS'11)*, page 10, Las Vegas, April 2011. Polytech'Nice-Sophia, IEEE.

[ACV+05] Vander Alves, Ivan Cardim, Heitor Vital, Pedro H. M. Sampaio, Alexandre L. G. Damasceno, Paulo Borba, and Geber Ramalho. Comparative Analysis of Porting Strategies in J2ME Games. In *ICSM*, pages 123–132. IEEE Computer Society, 2005.

[AD97] Larry Apfelbaum and John Doyle. Model Based Testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[Ag08] LINA INRIA Nantes ATLAS group. ATLAS transformation language. http://www.eclipse.org/m2m/atl/, April 2008.

[AG10] Isidro Ramos Abel Gómez. Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. Prague, Czech Republic, August 2010. 9th International Conference on Information Systems Development.

[AGL+10] Sarah Allen, Vidal Graupera, Lee Lundrigan, Sarah Allen, Vidal Graupera, and Lee Lundrigan. Phonegap. In *Pro Smartphone Cross-Platform Development*, pages 131–152. Apress, 2010.

[BBG+08] Sören Blom, Matthias Book, Volker Gruhn, Ruslan Hrushchak, and André Köhler. Write Once, Run Anywhere - A Survey of Mobile Runtime Environments. In *GPC Workshops*, pages 132–137. IEEE Computer Society, 2008.

[BDG+08] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Serge Lucio, Eric Samuelsson, Ina Schieferdecker, and Clay E. Williams. The UML 2.0 testing profile, 2008.

[Bec02] Kent Beck. *Test Driven Development : By Example*. Addison-Wesley Professional, November 2002.

[BFH08] F. T. Balagtas-Fernandez and H. Hussmann. Model-Driven Development of Mobile Applications. In *Proceedings of the 2008 23rd IEEE/ACM Inter-*

national Conference on Automated Software Engineering, ASE '08, pages 509–512, Washington, DC, USA, 2008. IEEE Computer Society.

[BM07] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 333–342, New York, NY, USA, 2007. ACM.

[Bos00] Jan Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley Professional, 2000.

[Bos05] Jan Bosch. Software product families in nokia. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 2–6. Springer Berlin / Heidelberg, 2005.

[Bos09] Jan Bosch. From software product lines to software ecosystems. In Dirk Muthig and John D. McGregor, editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 111–119. ACM, 2009.

[Bou09] T.J. Boudreaux. *Programming the iPhone user experience*. O'Reilly Media, 2009.

[CCSC07] Andrew Carton, Siobhan Clarke, Aline Senart, and Vinny Cahill. Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing. In *Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments*, SEPCASE '07, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45 :621–645, July 2006.

[CJ01] Matthias Clauß and Intershop Jena. Modeling variability with UML. In *In GCSE 2001 Young Researchers Workshop*, 2001.

[CM05] Vasian Cepa and Mira Mezini. MobCon : A Generative Middleware Framework for Java Mobile Applications. In *HICSS*. IEEE Computer Society, 2005.

[CN02] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley, 2002.

[DB07] Jürgen Dunkel and Ralf Bruns. Model-Driven Architecture for Mobile Applications. In Witold Abramowicz, editor, *BIS*, volume 4439 of *Lecture Notes in Computer Science*, pages 464–477. Springer, 2007.

[DFV+09] Z. Drey, F. Fleurey, D. Vojtisek, C. Faucher, and Vincent Mahé. *Kermeta Language, Reference Manual*, 2009.

[Dis08] Mobile Distillery. White paper : How to easily manage device fragmentation using standard ide's such as netbeans and eclipse with the celsius collaborative application development framework. an example. Technical report, Mobile Distillery, 2008.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext : implement your language faster than the quick and dirty way. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *SPLASH/OOPSLA Companion*, pages 307–309. ACM, 2010.

[EVP00] J. Eisenstein, J. Vanderdonckt, and A. Puerta. Adapting to mobile contexts with user-interface modeling. *IEEE Workshop on Mobile Computing Systems and Applications*, 0 :83, 2000.

[Fav10] Liliana Favre. Foundations for QVT Transformation. In *Software Engineering Research and Practice*, pages 58–64. CSREA Press, 2010.

[FLFL09] Gail Rahn Frederick, Rajesh Lal, Gail Rahn Frederick, and Rajesh Lal. Testing a mobile web site. In *Beginning Smartphone Web Development*, pages 259–272. Apress, 2009.

[FMS08] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML : The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[FMW+10] Frédéric Fondement, Pierre-Alain Muller, Brice Wittman, Fabrice Ambert, Fabrice Bouquet, Jonathan Lasalle, Emilie Oudot, Fabien Peureux, Bruno Legeard, Marc Alter, and Claude Scherrer. VETESS : IDM, Test et SysML. *Génie Logiciel*, (93) :43–48, June 2010. Selected paper from the 7-th NEPTUNE Workshop.

[GCH+05] Krzysztof Gajos, David B. Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld. Fast and Robust Interface Generation for Ubiquitous Applications. In *Ubicomp*, volume 3660 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2005.

[GE11] Damianos Gavalas and Daphne Economou. Development platforms for mobile applications : Status and trends. *IEEE Software*, 28(1) :77–86, 2011.

[GHKV08] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL : a domain-specific language for dynamic web applications. In Gail E. Harris, editor, *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, pages 779–780. ACM, 2008.

[Gro09] R. C. Gronback. *Eclipse Modeling Project : A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[Har01] Maarit Harsu. A survey of product-line architectures contents. Technical report, Tampere University of Technology, 2001.

[HKO06] Alan Hartman, Mika Katara, and Sergey Olvovsky. Choosing a Test Modeling Language : A Survey. In *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2006.

[HO09] Adrian Holzer and Jan Ondrus. Trends in Mobile Application Development. In *MOBILWARE Workshops*, volume 12 of *Lecture Notes of the Institute*

for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 55–64. Springer, 2009.

[Hom11] Bernard Homès. *Les tests logiciels : fondamentaux*. Lavoisier, 2011.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[ISO96] ISO. Information technology - syntactic metalanguage - extended bnf. Technical report, International Organization for Standardization, 8 1996.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS : a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE*, pages 249–254. ACM, 2006.

[JGB06] Jean-Marc Jézéquel, Sébastien Gérard, and Benoit Baudry. Le génie logiciel et l'IDM : une approche unificatrice par les modèles. In *L'ingénierie dirigée par les modèles*. Lavoisier, Hermes-science, 2006.

[JK09] Steven Kelly Juha Kärnä, Juha-Pekka Tolvanen. Evaluating the use of DSM in Practice. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM 09)*, 2009.

[JKK+09] Antti Jääskeläinen, Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, Tommi Takala, and Heikki Virtanen. Automatic GUI test generation for smartphone applications - an evaluation. In *ICSE Companion*, pages 112–122, 2009.

[JLT04] Steven Kelly Janne Luoma and Juha-Pekka Tolvanen. Defining Domain-Specific Modeling Languages : Collected Experiences. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM 04)*, 2004.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract : The lessons of ariane. *IEEE Computer*, 30(1) :129–130, 1997.

[KÖ6] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5(4) :369–385, December 2006.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KKP+09] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 7–13, 2009.

[KMH07] Andre N. Klingsheim, Veborn Moen, and Kjell J. Hole. Challenges in securing networked j2me applications. *Computer*, 40(2) :24–30, 2007.

[Kov07] Peter Kovari. Explore model-driven development (mdd) and related approaches : Applying domain-specific modeling to model-driven architecture, September 2007.

[KP09] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4) :22–29, 2009.

[Kra00] Paul Krause. Software test automation : Effective use of test execution tools, mark fewster and dorothy graham, addison-wesley, 1999 (book review). *Softw. Test., Verif. Reliab.*, 10(2) :140–142, 2000.

[KSB+10] Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard. Variability modeling and qos analysis of web services orchestrations. In *International Conference on Web Services*, Miami, FL, USA, July 2010. IEEE.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. John Wiley & Sons, 2008.

[Laf04] Pierre Laforcade. *Méta-modélisation UML pour la conception et la mise en oeuvre de situations-problèmes coopératives*. PhD thesis, UPPA, 2004.

[Law02] George Lawton. Moving java into mobile phones. *Computer*, 35 :17–20, June 2002.

[LBN09] Bruno Legeard, Fabrice Bouquet, and Pickaert Natacha. *Industrialiser le test fonctionnel*. Management des systèmes d'information. Dunod, April 2009.

[LH10] Daniel Lucas-Hirtz. Quatre étapes pour la réutilisation du logiciel. In *Journée Ligne de Produits Logiciels "Maîtriser la diversité"*. Université Paris 1, Paris, Octobre 2010.

[LT04] Yves Le Traon. *Contribution au test de logiciels orientés-objet*. PhD thesis, Habilitation à diriger les recherches de l'université de Rennes I, July 2004.

[LUV09a] Beatriz Pérez Lamancha, Macario Polo Usaola, and Mario Piattini Velthius. Towards an Automated Testing Framework to Manage Variability Using the UML Testing Profile. In *AST*, pages 10–17, 2009.

[LUV09b] Beatriz Pérez Lamancha, Macario Polo Usaola, and Mario Piattini Velthius. Software Product Line Testing - A Systematic Review. In Boris Shishkov, José Cordeiro, and Alpesh Ranchordas, editors, *ICSOFT (1)*, pages 23–30. INSTICC Press, 2009.

[McG07] John D. McGregor. Testing a Software Product Line. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *PSSE*, volume 6153 of *Lecture Notes in Computer Science*, pages 104–140. Springer, 2007.

[Mem07] Atif M. Memon. An event-flow model of gui-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3) :137–157, 2007.

[MFBC10] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling modeling modeling. *Journal of Software and Systems Modeling (SoSyM)*, 2010.

[MMYJ10] John D. McGregor, Dirk Muthig, Kentaro Yoshimura, and Paul Jensen. Guest editors' introduction : Successful software product line practices. *IEEE Software*, 27(3) :16–21, 2010.

[MPS03] Giulio Mori, Fabio Paternò, and Carmen Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, IUI '03, pages 141–148, New York, NY, USA, 2003. ACM.

[MS10] Mike Mannion and Juha Savolainen. Aligning Business and Technical Strategies for Software Product Lines. In *SPLC*, pages 406–419, 2010.

[MT00] Alessandro Maccari and Antti-Pekka Tuovinen. System Family Architectures : Current Challenges at Nokia. In Frank van der Linden, editor, *IW-SAPF*, volume 1951 of *Lecture Notes in Computer Science*, pages 107–115. Springer, 2000.

[NFTJ03] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. A Requirement-Based Approach to Test Product Families. In *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 198–210. Springer, 2003.

[Obj08] Object Management Group (OMG). Mof model to text transformation language 1.0. formal/2008-01-16, January 2008.

[OMG00] OMG. *Meta Object Facility (MOF) Specification*. Object Modeling Group, 2000.

[OMG06] OMG. Object Constraint Language - OMG Available Specification Version 2.0, 2006.

[OMG10] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3, 2010.

[Ove02] Stephanie Overby. The hidden costs of offshore outsourcing. Technical report, CIO Magazine, 2002.

[Pac05] Cyril Alexandre Pachon. *Une approche basée sur les modèles pour le test de robustesse*. PhD thesis, Université Joseph Fourier, 2005.

[PBL08] Vincent Pretre, Fabrice Bouquet, and Christophe Lang. Automating UML models merge for web services testing. In *iiWAS'08, 10th int. Conf. on Information Integration and Web-based Applications and Services*, pages 55–62, Linz, Austria, 2008. ACM Press.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005.

[PDH+10] Thibaut Possompès, Christophe Dony, Marianne Huchard, Hervé Rey, Chouki Tibermacine, and Xavier Vasques. Design of a UML profile for feature diagrams and its tooling implementation. Technical report, LIRMM - IBM, 2010.

[PdKB+09] Vincent Pretre, Adrien de Kermadec, Fabrice Bouquet, Christophe Lang, and Frédéric Dadeau. Automated UML models merging for web services testing. *Int. Journal on Web and Grid Services*, 5(2) :107–129, 2009.

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automatic and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing (ICST)*, Paris, France, April 2010. IEEE.

[PY10] Arno Puder and Ilmi Yoon. Smartphone Cross-Compilation Framework for Multiplayer Online Games. In *Proceedings of the 2010 Second International Conference on Mobile, Hybrid, and On-Line Learning*, ELML '10, pages 87–92, Washington, DC, USA, 2010. IEEE Computer Society.

[Raj08] Damith C. Rajapakse. Techniques for De-fragmenting Mobile Applications : A Taxonomy. In *SEKE*, pages 923–928. Knowledge Systems Institute Graduate School, 2008.

[RBBC10] Youssef Ridene, Nicolas Belloir, Franck Barbier, and Nadine Couture. A DSML for Mobile Phone Applications Testing. In *Proceedings of 10th Workshop on Domain-Specific Modeling*, pages 25–30. ACM Press, 2010.

[Sar09] Biswajit Sarkar. *LWUIT 1.1 for Java ME Developers*. Packt Publishing, 2009.

[SB00] Mikael Svahnberg and Jan Bosch. Issues Concerning Variability in Software Product Lines. In Frank van der Linden, editor, *IW-SAPF*, volume 1951 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2000.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[SE06] Markus Völter Sven Efftinge. oaw xtext : A framework for textual DSLs. = http ://www.voelter.de/data/workshops/EfftingeVoelterEclipseSummit.pdf, September 2006.

[SHS09] Osamuyimen Stewart, Juan M. Huerta, and Melissa Sader. Designing crowdsourcing community for the enterprise. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 50–53, New York, NY, USA, 2009. ACM.

[SKM07] Monalisa Sarma, Debasish Kundu, and Rajib Mall. Automatic Test Case Generation from UML Sequence Diagram. *Advanced Computing and Communications, International Conference on*, 0 :60–67, 2007.

[Sol00] Richard Soley. Model driven architecture. *Object Management Group*, 2000.

[Tol04] Juha-Pekka Tolvanen. MetaEdit+ : domain-specific modeling for full code generation demonstrated [GPCE]. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 39–40, New York, NY, USA, 2004. ACM.

[tS10] Jan ten Sythoff. Mobile app stores : A new mobile web ? Technical report, Pyramid Research, 2010.

[twu10] Twuik : Powerful ui technology for your javame applications. http://www.tricastmedia.com/, 2010.

[TZJ03] Loïc Hélouët Tewfik Ziadi and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *PFE*, Lecture Notes in Computer Science, pages 129–139, Seana, Italy, 2003. Springer.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.

[UML09] OMG Unified Modeling Language (OMG UML), Superstructure Version 2.2, 2009. Version 2.2 is a minor revision to the UML 2.1.2 specification. It supersedes formal/2007-11-02.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical report, University of WAkato, Hamilton, New Zealand, April 2006.

[VG07] M. Voelter and I. Groher. Handling Variability in Model Transformations and Generators. In *Proceedings of the 7th Workshop on Domain-Specific Modeling (DSM'07) at OOPSLA '07*, 2007.

[Vir05] Robert Virkus. Dancing around device limitations. In *Pro J2ME Polish*, pages 283–323. Apress, 2005.

[Vuk09] Maja Vukovic. Crowdsourcing for Enterprises. In *IEEE Congress on Services*, pages 686–692, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[Wei93] Mark Weiser. Hot topics : Ubiquitous computing. *Computer*, 26(10) :71–72, 1993.

[Whi01] James White. An introduction to Java 2 micro edition (J2ME) ; Java in small things. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 724–725, Washington, DC, USA, 2001. IEEE Computer Society.

[Whi05] Jim White. Tackle device fragmentation with netbeans and the netbeans mobility pack. http://www.devx.com/wireless/Article/29449, 2005.

[WS08] Jules White and Douglas C. Schmidt. Model-Driven Product-Line Architectures for Mobile Devices. In *Proceedings of the 17th Annual Conference of the International Federation of Automatic Control*, pages 6–11, 2008.

[XL06] Kai Xu and Donglin Liang. A Monitoring Profile for UML Sequence Diagrams. 2006.

[You05] Trevor J. Young. Using AspectJ to Build a Software Product Line for Mobile Devices. Master's thesis, University of British Columbia, 2005.

[Zdu07] Marcin Zduniak. Automated gui testing of mobile java applications. Master Thesis, 2007.

[Zia04] Tewfic Ziadi. *Manipulation de lignes de produits en UML*. PhD thesis, Université de Rennes 1, December 2004.

[ZJF03] Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Product line derivation with UML. In *Proceedings Software Variability Management Workshop, Univ. of Groningen Departement of Mathematics and Computing Science*, February 2003.

Résumé

L'engouement du grand public pour les applications mobiles, dont le nombre ne cesse de croître, a rendu les utilisateurs de plus en plus exigeants quant à la qualité de ces applications. Seule une procédure de test efficace permet de répondre à ces exigences. Dans le contexte des applications embarquées sur téléphones mobiles, le test est une tâche coûteuse et répétitive principalement à cause du nombre important de terminaux mobiles qui sont tous différents les uns des autres.

Nous proposons dans cette thèse le langage MATeL, un DSML (Domain-Specific Modeling Language) qui permet de décrire des scénarios de test spécifiques aux applications mobiles. Sa syntaxe abstraite, *i.e.* un métamodèle et des contraintes OCL, permet au concepteur de manipuler les concepts métier du test d'applications mobiles (testeur, mobile ou encore résultats attendus et résultats obtenus). Par ailleurs, il permet d'enrichir ces scénarios avec des points de variabilité qui autorisent de spécifier des variations dans le test en fonction des particularités d'un mobile ou d'un ensemble de mobiles. La syntaxe concrète de MATeL, qui est inspirée de celle des diagrammes de séquence UML, ainsi que son environnement basé sur Eclipse permettent à l'utilisateur de concevoir des scénarios relativement facilement.

Grâce à une plateforme de test en ligne construite pour les besoins de notre projet, il est possible d'exécuter les scénarios sur plusieurs téléphones différents. La démarche est illustrée dans cette thèse à travers des cas d'utilisation et des expérimentations qui ont permis de vérifier et valider notre proposition.

Mots-clés: Langage de modélisation spécifique, Ligne de produit logiciel, Model-Driven Testing, Applications mobiles.

Abstract

Mobile applications have increased substantially in volume with the emergence of smartphones. Ensuring high quality and successful user experience is crucial to the success of such applications. Only an efficient test procedure allows developers to meet these requirements. In the context of embedded mobile applications, the test is costly and repetitive. This is mainly due to the large number of different mobile devices.

In this thesis, we describe MATeL, a Domain-Specific Modeling Language (DSML) for designing test scenarios for mobile applications. Its abstract syntax, *i.e.* a metamodel and OCL constraints, enables the test designer to manipulate mobile applications testing concepts such as tester, mobile or outcomes and results. It also enables him/her to enrich these scenarios with variability points in the spirit of Software Product-Line engineering, that can specify variations in the test according to the characteristics of one mobile or a set of mobiles. The concrete syntax of MATeL that is inspired from UML sequence diagrams and its environment based on Eclipse allow the user to easily develop scenarios.

MATeL is built upon an industrial platform (a test bed) in order to be able to run scenarios on several different phones. The approach is illustrated in this thesis through use cases and experiments that led to verify and validate our contribution.

Keywords: Domain-Specific Modeling Language, Software Product Line, Model-Based Testing, Mobile applications.