TABLE DES MATIÈRES

PARTIE I – INTRODUCTION GÉNÉRALE

PARTIE II - CONTEXTE TECHNOLOGIQUE ET SCIENTIFIQUE

CHAPITRE 1 - DE L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES À L'APPROCHE MDA

PARTIE IV - CONCLUSION GÉNÉRALE

TABLE DES FIGURES

LISTE DES TABLEAUX

ACRONYMES

CIM : Computation Independant Model

DSL : Domain Specific Language.

EBNF : Extended Bachus Naur Form

HCIDL : Human-Computer Interface Description Languages.

IDM : Ingénierie Dirigée par les Modèles.

IHM : Interface Homme – Machine.

MDA : Model Driven Architecture.

MOFM2T : MOF Model To Text Transformation

MDE : Model Driven Engineering.

MVC : Model – View – Controller.

OCL : Object Constraint Language

OMG : Object Management Group.

PIM : Platform Independant Model

PSM : Platform Specific Model

QVT : Query/View/Transformation

UIDL : User Interface Description Language.

WIMP : Windows, Icons, Menus, Pointing.

PARTIE I
INTRODUCTION GÉNÉRALE

INTRODUCTION

1. CONTEXTE DE RECHERCHE

L'ingénierie dirigée par les modèles (IDM) se définie au sens large comme une approche qui étudie l'usage des modèles dans les processus d'ingénierie. Comme dans tout processus d'ingénierie, le processus est indissociable du produit. Le produit est le résultat à atteindre, tandis que le processus est le chemin qu'il faut parcourir pour atteindre le résultat. L'ingénierie dirigée par les modèles s'inscrit dans cette perspective qui considère les modèles comme les outils de base dans la production, le fonctionnement et l'évolution des systèmes. L'utilisation des systèmes informatiques prend de plus en plus de place dans notre vie quotidienne. Aujourd'hui, voitures, téléviseurs, micro-ondes, tablettes, smartphones, etc. fonctionnent tous avec des programmes informatiques communicants plus ou moins complexes.

L'ingénierie dirigée par les modèles est notre champ d'applications avec comme domaine cible le développement d'Interfaces Homme-Machine. Dans ce contexte, les modèles sont le pivot du processus de production là où les IHM deviennent le produit, i.e. le résultat à atteindre. Le domaine de l'interaction homme-machine, phare de l'ère internet et de la mobilité, a suscité un grand intérêt en particulier depuis l'émergence de nouveaux moyens d'interaction.

L'objectif principal de nos travaux est l'application de l'ingénierie dirigée par les modèles à l'étude de l'interaction homme-machine en se focalisant sur trois de ses principaux axes :

- les techniques de modélisation et métamodélisation ;
- les techniques de transformation ;
- les méthodologies de développement.

Nous allons dans un premier temps présenter brièvement les domaines de l'IDM et de l'interaction homme-machine afin de préciser le contexte général de nos travaux. Nous discuterons ensuite de notre problématique et de nos objectifs. L'organisation de ce document est présentée en terme de cette introduction générale.

2. MOTIVATIONS ET DOMAINES APPLICATIFS

2.1. L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

En novembre 2000, l'OMG (*Object Management Group*) initie l'approche MDA (*Model Driven Architecture*) [Soley, 2000] pour le développement et la maintenance des systèmes à prépondérance logicielle. Cette approche s'inscrit dans une tendance plus générale appelée Ingénierie Dirigée par les Modèles (IDM) qui met le modèle au centre du développement des logiciels et des systèmes d'information. L'IDM a pour but d'apporter une nouvelle vision unifiée de concevoir des applications en séparant la logique métier, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il paraît donc bénéfique de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique. Cette

séparation autorise alors la capitalisation du savoir logiciel. A ce niveau, les technologies basées sur l'approche objet et l'approche composant n'ont pas pu tenir leurs promesses. Face à la complexité croissante des systèmes d'information, il devenait de plus en plus difficile de développer des logiciels basés sur ces technologies. Le recours à des procédés supplémentaires, comme les patrons de conception [Gamma et al., 1993] ou la programmation orientée aspect [Kiczales et al., 1997], était alors nécessaire. De manière encore plus radicale que pouvaient l'être les approches des patrons et des aspects, l'IDM vise à fournir différents types de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux approches précédentes. Elle offre la possibilité d'arrêter l'empilement des technologies qui nécessite une conservation des compétences particulières pour faire cohabiter des systèmes divers et variés. Ceci est permis grâce au passage d'une approche interprétative à une approche transformationnelle, où les concepteurs ont un rôle simplifié et amoindri grâce à la construction automatisée et aux techniques de transformations entre modèles. L'initiative MDA proposée par l'OMG peut se définir comme la réalisation des principes de l'ingénierie des modèles autour d'un ensemble de standards comme le MOF, MOFM2T, XMI, OCL, UML, CWM, SPEM, QVT, ODM, etc. Nos travaux seront basés sur l'approche MDA et ses technologies standards.

2.2. L'INTERACTION HOMME-MACHINE

Les interfaces homme-machine doivent synthétiser l'état de l'application et aider l'être humain lors de ses activités. Les interfaces des ordinateurs de bureaux peuvent répondre de manière efficace aux besoins des utilisateurs pour des activités spécifiques. En revanche, lorsque l'information devient de plus en plus complexe à représenter ou à manipuler, que les situations d'interaction diffèrent, il convient d'adopter de nouvelles formes d'interaction. Les systèmes interactifs ont évolué pour offrir aux utilisateurs des systèmes efficaces, quel que soit le contexte d'utilisation. Pour que ces systèmes restent utilisables en dépit de leur complexité, il est nécessaire d'adopter de nouveaux moyens d'interaction qui vont au-delà de la triade clavier-écran-souris et se démarquent du paradigme d'interaction WIMP[1]. Ainsi, une représentation plus instinctive de l'information est nécessaire, que ce soit en entrée ou en sortie des systèmes.

Cette observation a conduit à repenser fondamentalement les modes d'interaction entre l'homme et la machine. L'objectif principal est de promouvoir, autant que possible, une communication naturelle avec l'utilisateur final. La combinaison de multiples modalités en entrée et/ou sortie permet d'améliorer à la fois la robustesse et la fiabilité de l'interaction. Les travaux de Bolt (1980) sont à l'origine de la conception et du développement d'applications multimodales, où l'ordinateur est enrichi de nouveaux modes d'interaction pour les soutenir. En outre, l'émergence de l'informatique mobile, les différents types de capteurs qui équipent les appareils mobiles (smartphones, tablette ...) a permis l'émergence de nouvelles modalités [Bellik, 1995], tels que l'inclinaison du téléphone ou de changer son orientation. La multimodalité réduit considérablement les limites des plateformes d'interaction, tels que les petits écrans ou les claviers inconfortables, et limite les erreurs de reconnaissance.

Par conséquent, les principes de l'interaction homme-machine sont véhiculés par deux objectifs principaux : le premier est de renforcer l'adéquation des systèmes aux besoins des utilisateurs. Le second objectif est d'augmenter la facilité d'utilisation des systèmes par la conception

[1] *Windows, Icons, Menus, Pointing*

d'interfaces en adéquation avec l'environnement dans lequel l'application va évoluer, et ayant des modalités d'interaction conviviales. Dans ce contexte, notre objectif est de proposer une approche pour la conception et la génération d'interfaces d'applications pouvant s'adapter à différentes plateformes d'interaction, tout en offrant un confort optimal d'utilisation.

3. PROBLÉMATIQUE

L'interaction avec un système informatique a profondément évoluée durant les dernières décennies. Les avancées technologiques ont profondément changé la définition d'un système informatique. L'ordinateur de bureau n'est plus le seul représentant des systèmes grands publics. Aussi, la miniaturisation des dispositifs, les performances des unités de calculs et l'apparition de nouveaux moyens d'interaction nous poussent à repenser totalement les modes d'interaction entre l'homme et la machine. Les défis à relever sont notamment liés à la prise en compte d'interactions multiples et de la diversité des dispositifs. Le large panel d'interactions (toucher, vibration, reconnaissance vocale, géolocalisation ...) et la diversification des dispositifs d'interaction peuvent être considérés comme un facteur de flexibilité, mais introduisant une complexité inhérente.

Dernièrement, avec l'amélioration des technologies utilisées et l'accélération du processus de conception, l'IDM a attiré l'attention de la communauté d'IHM (Interaction Homme-Machine) pour la conception et la génération des interfaces d'applications. La problématique à laquelle nous cherchons à apporter une solution est la suivante :

> ***Comment intégrer cette nouvelle définition de l'interaction homme-machine au développement d'interfaces d'applications en se basant sur une approche de type IDM ?***

4. OBJECTIFS ET CONTRIBUTIONS DE LA THÈSE

Ces travaux de thèse s'adressent aux concepteurs d'interfaces d'applications. Ils visent à faciliter le développement des interfaces d'application dans le contexte technologique actuel. En effet, le paysage des technologies de l'information et de la communication se présente sous la forme d'une impressionnante catégorie de dispositifs d'accès à l'information, dotés de nouvelles techniques d'interaction telles que le toucher, la vibration, la reconnaissance vocale, etc. La variété des dispositifs fixes/mobiles et des techniques d'interaction rend la tâche plus difficile pour les développeurs d'interfaces homme-machine en raison de l'absence d'un niveau d'abstraction d'interaction approprié, nécessitant ainsi la création de plusieurs versions de la même interface utilisateur en fonction des variations physiques et matérielles des périphériques.

Face à ce problème, le concept d'UIDL (*User Interface Description Language*) semble être la réponse pertinente pour la création de systèmes interactifs, comme le démontrent de nombreuses contributions. Le formalisme ConcurTaskTrees [Paterno et al., 1997] est l'une des notations les plus populaires pour les modèles de tâches. Il est largement utilisé par les approches de modélisation et de conception d'interfaces d'applications telles que UsiXML [Limbourg et al., 2004]. Cependant, dans UsiXML, la génération de code est uniquement orientée web et les capteurs mobiles ne sont pas pris en charge.

Nos travaux s'inscrivent dans le domaine des langages de description d'interfaces utilisateurs. Nos recherches se concrétisent par l'introduction de notre langage HCIDL (*Human-Computer Interface Description Languages*), un langage de modélisation mis en scène dans une approche MDA. Parmi les propriétés liées à l'interface homme-machine, notre proposition est destinée à la modélisation d'interfaces d'interaction plastiques, multi-cibles, multimodales, utilisant des langages de description d'interface utilisateur. En combinant la plasticité et la multimodalité, HCIDL améliore la convivialité des interfaces utilisateur grâce à un comportement adaptatif en fournissant aux utilisateurs finaux un ensemble d'interaction adapté aux entrées/sorties des terminaux et une disposition de l'interface graphique optimale.

5. ORGANISATION DE LA THÈSE

Cette thèse est divisée en quatre parties. La figure suivante permet de visualiser sa structure :

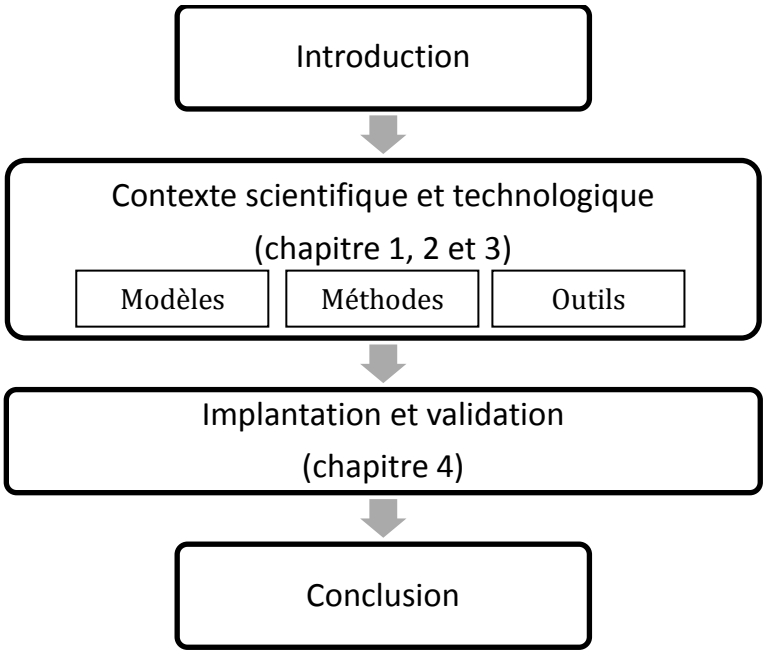

Figure 1. Structure de la thèse.

Les deux premiers chapitres exposent le contexte scientifique et technologique dans lesquels s'inscrivent nos travaux. Ainsi, le chapitre 1 couvre le domaine de l'ingénierie dirigée par les modèles. Nous en présentons les piliers fondateurs et détaillons chaque étape du processus de développement propre à ce domaine. Le chapitre 2 étudie l'ingénierie des interfaces homme-machine. Plus particulièrement, on s'intéressera à l'évolution de l'interaction homme-machine face à l'apparition de nouveaux dispositifs d'interaction. Ensuite, nous présenterons en chapitre 3 l'ingénierie dirigée par les modèles pour le développement d'interfaces homme-machine à travers un état de l'art afin de mettre en évidence l'approche que l'on souhaite cibler.

Nous présentons par la suite, dans la partie Implantation et validation, notre contribution à la création d'interfaces d'applications en tenant compte des points de variabilité dépendants des caractéristiques des périphériques cibles. Nous détaillons le langage développé et ses différentes

composantes. Nous présentons l'environnement de modélisation et de génération que nous avons utilisé et nous décrivons la procédure de conception et de génération d'une interface d'application.

PARTIE II
CONTEXTE TECHNOLOGIQUE ET SCIENTIFIQUE

CHAPITRE 1
DE L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES À L'APPROCHE MDA

" Le MDA n'est probablement pas la solution miracle pour le génie logiciel, mais: il n'y a pas d'alternative raisonnable au MDA "
Jean Bézivin

1.1. INTRODUCTION

En novembre 2000, *l'Object Management Group* (OMG[2]) propose l'approche MDA (*Model Driven Architecture*) pour le développement et la maintenance des systèmes logiciels [Soley, 2000]. MDA applique la séparation des préoccupations entre la logique métier des systèmes informatiques et la logique applicative des plateformes utilisées. Pour atteindre cet objectif, MDA vise à représenter sous forme de modèles toute l'information utile et nécessaire à la construction et à l'évolution des systèmes d'information. Les modèles sont au centre du cycle de vie des logiciels et des systèmes d'information.

Au-delà de la proposition spécifique MDA de l'OMG, l'Ingénierie Dirigée par les Modèles (IDM), ou *Model Driven Engineering* (MDE) en anglais, est une forme d'ingénierie générative dans laquelle tout ou une partie d'une application est engendrée à partir de modèles. L'objectif de l'IDM est de définir une approche pouvant intégrer différents espaces technologiques [Kurtev et al., 2002]. Ainsi, l'approche MDA devient une variante particulière de l'Ingénierie Dirigée par les Modèles.

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. L'un des principaux buts 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. Ainsi, les modèles sont les outils principaux du processus de transformation, apportant de ce fait une grande part d'automatisation au processus de développement.

Dans ce chapitre, nous proposons dans un premier temps une présentation des principes clés de l'IDM dans la section 2. Puis nous focaliserons notre discussion sur l'approche MDA dans la section 3 et les différents aspects qui nous intéressent dans le cadre de la définition d'un langage

[2] https://www.omg.org/

spécifique de domaine ou *Domain Specific Language* (DSL) en anglais. Nous détaillons les deux premiers axes principaux de l'IDM selon l'approche MDA et sur lesquels portent nos travaux de recherche : le processus de développement et la modélisation (métamodélisation). La section 4 présente la transformation de modèles qui est le troisième axe clé de MDA. Nous concluons enfin par la section 5 avec une synthèse des différents points discutés dans ce chapitre.

1.2. L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

La modélisation est la représentation de l'information à différents niveaux d'abstractions. Dans le cas d'une application informatique, un modèle permet de représenter, à chaque étape du développement, certaines caractéristiques de l'application et de son environnement. L'IDM est la mise en œuvre de la modélisation dans le domaine du développement logiciel.

Cette section présente les différentes définitions autour des quatre concepts de base de l'ingénierie dirigée par les modèles: système, modèle, métamodèle et langage, et les relations qui lient ces concepts (*cf.* figure 1.1).

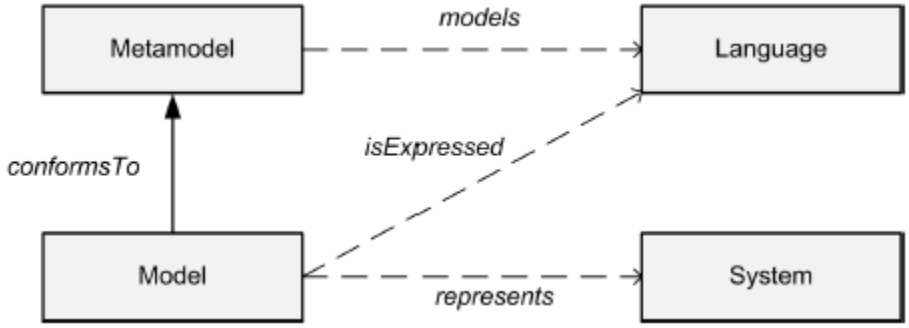

Figure 1.1. Relations entre système, modèle, métamodèle et langage.

1.4.1. MODÈLE ET SYSTÈME

En IDM, il n'y a pas de consensus concernant la définition de la notion de modèle. Minsky (1969) considère le modèle comme étant « *...une représentation (ou abstraction) d'un système, décrit dans une intention particulière* ». Bézivin et Gerbé (2001) définissent un modèle comme *« ... une simplification d'un système construit avec un objectif bien déterminé. Le modèle devrait être capable de répondre aux questions à la place du système actuel »*. Dans [Seidewitz, 2003], un modèle est considéré comme « *... un ensemble de définitions concernant un système qui est en train d'être étudié* ». Selon Mellor *et al.* (2003), c'est « *... un ensemble cohérent des éléments formels décrivant quelque chose (par exemple un système, une banque, un téléphone, ou un train), construit dans un but bien déterminé* ». Dans notre cas, nous avons retenu la définition proposée par l'OMG (2003) :

Définition. *Un modèle est une description ou une spécification d'un système et de son environnement dans un but bien déterminé. Un modèle est souvent présenté comme une combinaison de dessins et de textes. Le texte peut être dans un langage de modélisation ou dans une langue naturelle.*

Cette définition stipule qu'un modèle contient un ensemble d'informations choisi pour être pertinent vis à vis de l'utilisation qui sera faite du modèle. On dit alors que le modèle représente

le système. On déduit de cette définition la première relation fondamentale de l'IDM, entre le modèle et le système qu'il représente, nommée *represents* dans [Atkinson et Kühne, 2003] [Bézivin, 2004].

1.4.2. MÉTAMODÈLE : LANGAGE DE MODÉLISATION

Dans le domaine de l'IDM, la métamodélisation joue un rôle très important. En effet, elle est considérée comme une technique courante pour définir la syntaxe abstraite des modèles et des interrelations entre les éléments du modèle. Si le modèle est une abstraction des éléments du monde réel, le métamodèle représente encore une autre abstraction, définissant les propriétés du modèle lui-même. L'OMG a défini le métamodèle comme suit [OMG, 2006] :

Définition. *Un métamodèle est un modèle définissant le langage permettant d'exprimer un modèle.*

Ainsi, un métamodèle représente les concepts du langage de modélisation utilisé et la sémantique qui leur est associée. Cette notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, désignée par *conformsTo* dans [Bézivin, 2004] [Favre, 2004] et stipule en fait qu'un modèle est conforme à son métamodèle. Cette notion de conformité est essentielle à l'ingénierie dirigée par les modèles mais n'est pas nouvelle. Par exemple, un texte est conforme à une orthographe et une grammaire et un document XML est conforme à sa DTD.

1.4.3. MÉTA-MÉTAMODÈLE : LANGAGE DE MÉTAMODÉLISATION

Lorsqu'on rédige un programme en Java (ou dans tout autre langage), celui-ci respecte la grammaire du langage Java, c'est le formalisme de modélisation. Ce formalisme est lui-même définit par une grammaire EBNF (*Extended Bachus Naur Form*) [ISO, 1996] qui se définie elle-même. La grammaire BNF est appelée dans ce cas un méta-formalisme. Il en va de même pour l'élaboration des modèles dans l'IDM. Dans l'IDM, formalisme de modélisation et méta-formalisme sont appelés respectivement métamodèle et méta-métamodèle. Ainsi chaque modèle est exprimé dans le formalisme de son métamodèle, qui lui-même est définit par son métamodèle, autrement dit le méta-métamodèle. Afin de limiter le nombre de niveaux d'abstraction, un méta-métamodèle est conçu avec la propriété de méta-circularité, c'est-à-dire, la capacité de se décrire lui-même. L'OMG [OMG, 2006] a défini le méta-métamodèle comme suit :

Définition. *Un méta-métamodèle est un modèle qui permet de décrire un langage de métamodélisation. Un méta-métamodèle doit être réflexif pour limiter le nombre de niveaux d'abstraction.*

Les concepts de système, modèle et métamodèle ainsi que les relations qui les lient, représentent les principes de base sur lesquels s'appuie l'IDM. A cet égard, l'OMG a introduit l'architecture à quatre niveaux illustrée dans la Figure 1.2 appelée également *l'architecture 3+1* ou bien *la pile de modélisation*.

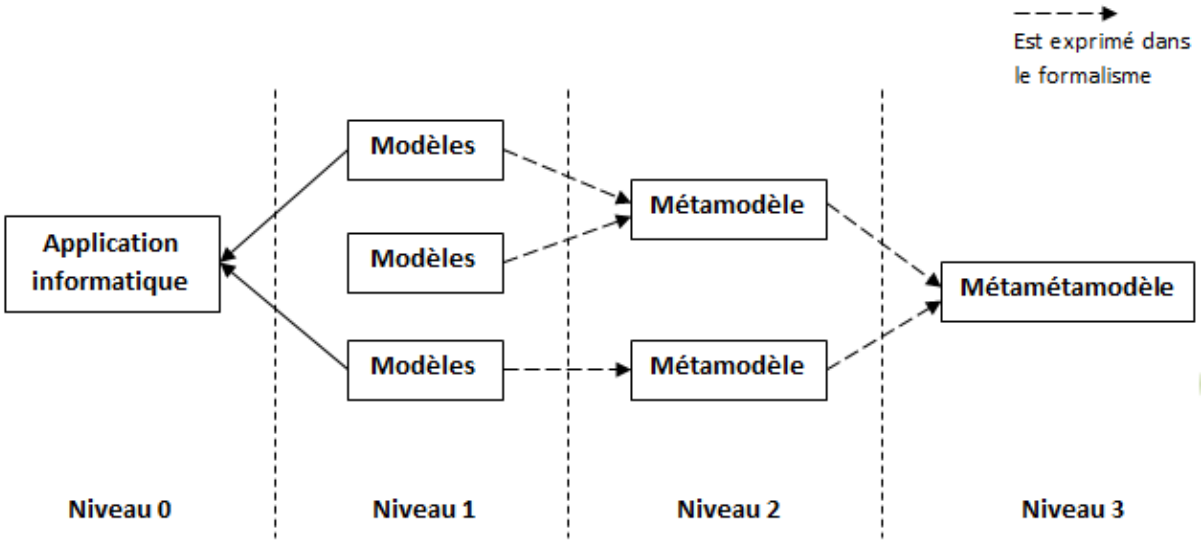

Figure 1.2. Architecture 3+1 adaptée de [Bézivin, 2005].

En IDM, une application informatique est représentée par un ou plusieurs modèles. Un métamodèle définit la structure (et non la sémantique) des modèles conforment à ce métamodèle. Un métamodèle est un diagramme de classes qui définit les entités du modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. Pour être valide, chaque modèle doit être conforme à son métamodèle. Cette relation de conformité est primordiale en IDM, il est possible ainsi de construire des outils capables de manipuler les modèles. Le méta-métamodèle est pour ainsi dire le métamodèle des métamodèles, et entretien avec eux la même relation qu'a un métamodèle avec ses modèles. Les métamodèles sont des modèles conforment à leur méta-métamodèle.

1.4.4. IDM ET TRANSFORMATION DE MODÈLES

Les deux principaux artefacts de l'ingénierie dirigée par les modèles sont les modèles et les transformations de modèles. Rendre les modèles productifs consiste à leur appliquer des transformations pour obtenir des résultats utiles au développement, généralement sous la forme de modèles plus détaillés et proches de la solution technique finale, ou du code. D'un point de vue général, on appelle transformation de modèles tout programme dont les entrées et les sorties sont des modèles. Plusieurs définitions ont été proposées pour préciser la transformation de modèles. Dans [Kleppe *et al.*, 2003], la transformation de modèles est définie de la manière suivante :

Définition. *Une transformation est une génération automatique d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources, en respectant une définition de transformation. Une définition de transformation est un ensemble de règles de transformation qui décrivent la manière avec laquelle un modèle dans le langage source peut être transformé en un modèle dans le langage cible. Une règle de transformation est une description de la façon avec laquelle une ou plusieurs constructions dans le langage source peuvent être transformées en une ou plusieurs constructions dans le langage cible.*

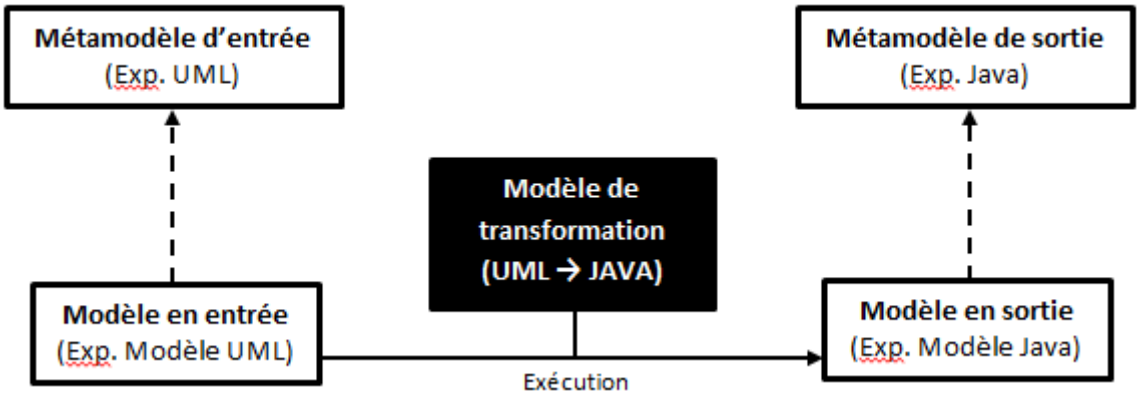

Figure 1.3. Moteur de transformation de modèles.

La figure 1.3 illustre les relations entre les transformations de modèles et les métamodèles. Une transformation exprime des correspondances structurelles entre les modèles source et cible. Ces correspondances structurelles s'appuient sur les métamodèles des modèles source et cible [Blanc, 2005]. Par exemple, une transformation de modèles visant à transformer des modèles UML vers des programmes JAVA spécifierait la règle suivante : à toute classe UML correspond une classe JAVA. Cette même règle, exprimée selon les métamodèles, deviendrait : à tout élément du modèle source instance de la métaclasse **Class** du métamodèle UML doit correspondre un élément du modèle cible instance de la métaclasse **Class** du métamodèle JAVA. Cette règle exprime bien une correspondance structurelle entre les métamodèles UML et JAVA et plus précisément une correspondance entre les métaclasses **Class (UML)** et **Class (JAVA)**. Si la transformation ici est binaire, avec une unique source et une unique cible, il est important de souligner que les transformations de modèles peuvent être N-aires, c'est à dire qu'elles peuvent avoir en entrée et en sortie plusieurs modèles.

Il existe plusieurs langages de transformation de modèles, parmi lesquels on peut citer ATL[3], JET, QVT ou encore Acceleo[4] qui est une implémentation du standard MTL [OMG, 2008] de l'OMG. Pour une lecture plus approfondie sur les différents langages de transformation, nous invitons le lecteur à se référer aux travaux de [Czarnecki et Helsen, 2003] [Jouault et Kurtev, 2006].

1.3. L'APPROCHE MDA

MDA (*Model-Driven Architecture*) est un standard, né de l'initiative de l'OMG [OMG, 2003], qui se base sur l'IDM, fournissant un ensemble de lignes directrices ainsi qu'une architecture pour la conception des systèmes logiciels. La spécification MDA adoptée en 2003 [Miller et Mukerji, 2003] donne une définition détaillée de l'architecture. La figure 1.4 illustre la philosophie, les technologies ainsi que l'architecture de modélisation de MDA. La figure de gauche illustre les trois piliers sur lesquels s'appuie l'approche MDA. La figure de droite illustre trois principaux standards de MDA dans l'architecture à 4 niveaux. Il est important de signaler que l'approche MDA n'est pas une méthodologie de conception mais un cadre générique à l'application de la modélisation dans un processus de développement applicatif.

[3] http://www.eclipse.org/m2m/atl/
[4] http://www.eclipse.org/acceleo/

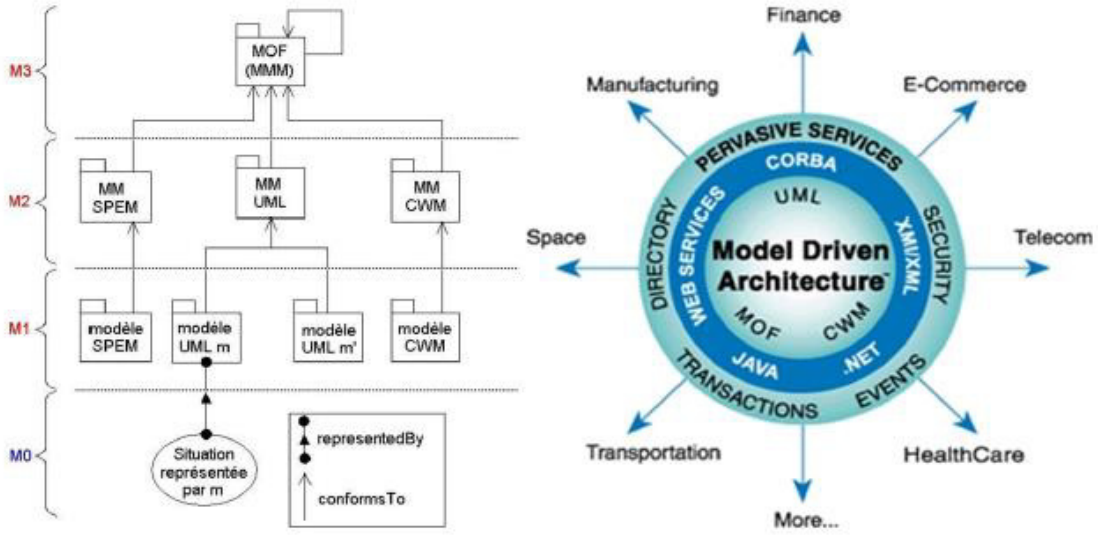

Figure 1.4. Principes, technologies et architecture MDA.

MDA définit plusieurs formalismes standards de modélisation, notamment UML, MOF et XMI, afin de promouvoir les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plateformes d'exécution. Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases de développement d'une application en s'appuyant sur le standard UML. Plus précisément, MDA préconise l'élaboration des modèles :

- d'exigences *(Computation Independant Model* - CIM), dans lesquels aucune considération informatique n'apparait ;
- d'analyse et de conception (*Platform Independant Model* - PIM) ;
- de code (*Platform Specific Model* - PSM).

Dans la section 3.1 nous discuterons plus en détails ces trois types de modèles et leur place dans le processus de développement MDA. L'objectif majeur de MDA [Blanc, 2005], est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plateformes d'exécution (J2EE, .Net, PHP, Oracle, etc.), afin de permettre la génération automatique de la totalité des modèles de code (PSM) et d'obtenir un gain significatif de productivité. Cette génération automatique est possible grâce à l'exécution des transformations de modèles. Le succès de MDA et de l'IDM en général, repose en grande partie sur la résolution de problèmes de transformation de modèles. Ainsi, afin de donner un cadre normatif pour l'implémentation des différents langages dédiés à la transformation de modèles, l'OMG a défini le standard QVT (*Query*/*View*/*Transformation*) [OMG, 2016]. La section 4 sera dédiée à la partie transformation de modèles dans MDA.

1.4.1. Processus de développement dans MDA

L'approche MDA est basée sur la séparation des préoccupations et la transformation de modèles. Elle préconise de modéliser, séparément, l'aspect métier qui représente les fonctions de l'application, et l'aspect technique qui représente la technologie de mise en œuvre. Le code source est obtenu par transformations successives des modèles de l'application, jusqu'à la génération automatique du code. Les modèles ne sont plus seulement un élément visuel ou de communication mais sont, dans MDA, un élément productif. La figure 1.5 illustre le processus de

développement selon l'approche MDA. Cette figure en trois couloirs est une extension d'une première ébauche présentée dans [Kleppe et al., 2003]. Le couloir de gauche présente les activités du processus classique de développement de logiciel. Le couloir central présente les différents modèles impliqués dans chaque activité et les liens entre les différents modèles. Enfin, le couloir de droite présente les principaux formalismes préconisés à chaque étape et pour chaque type de modèle. Comme il a été remarqué dans [Kleppe et al., 2003], les phases de développement dans MDA sont similaires au processus classique de développement logiciel. La principale différence réside dans la nature des artefacts qui sont crées durant le processus de développement. Les artefacts dans le cadre de MDA sont des modèles formels interprétables et manipulables par une machine.

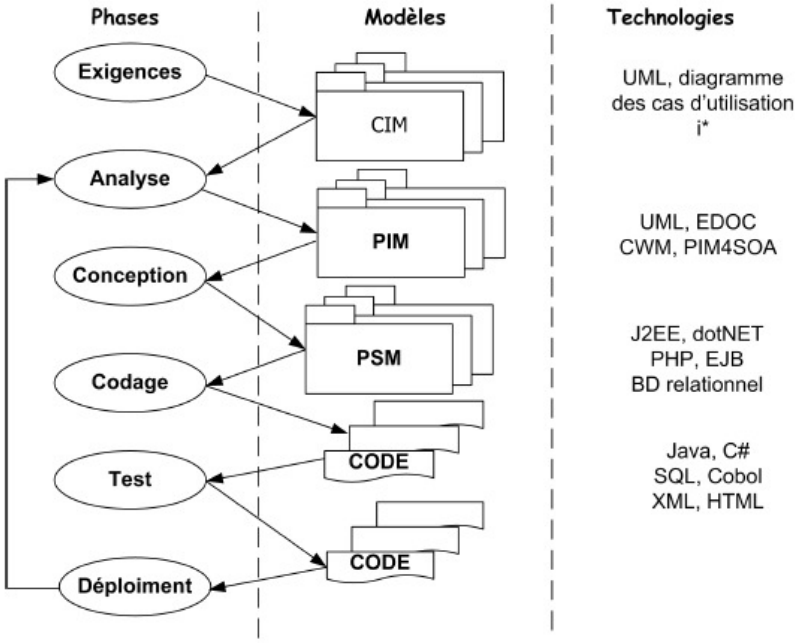

Figure 1.5. Processus de développement selon l'approche MDA.

1.3.1.1. LE MODÈLE D'EXIGENCES CIM

Un modèle d'exigence décrit les besoins fonctionnels de l'application, aussi bien les services qu'elle offre que les entités avec lesquelles elle interagit. Il traduit les besoins du client et représente ainsi le point de départ dans le cycle de développement. Il est important de noter qu'un modèle d'exigences ne contient pas d'informations sur la réalisation de l'application ni sur les traitements. D'où l'appellation qui leur ai attribuée dans le vocabulaire MDA de CIM (*Computation Independant Model*), modèle indépendant de la programmation. Avec UML, un modèle d'exigence peut se résumer à un diagramme de cas d'utilisation.

1.3.1.2. LE MODÈLE PIM

Les modèles appelés PIM (*Platform Independant Model*), désignés aussi par « modèle d'analyse et de conception abstraite » spécifient la logique métier de l'application et ne doivent pas contenir d'informations sur les technologies qui seront utilisées pour déployer l'application. Même si MDA préconise l'utilisation d'UML pour réaliser les modèles d'analyse et de conception, il n'exclut pas que d'autres langages puissent être utilisés. En effet, l'IDM, au contraire, favorise la définition de langages de modélisation dédiés à un domaine particulier [Jouault et Bézivin, 2006] (*Domain Specific Languages*– DSL) offrant ainsi aux utilisateurs des concepts propres à

leur métier et dont ils ont la maîtrise. Dans ce contexte, l'OMG a proposé des langages standards dédiés à des domaines d'applications, tel que EDOC (*Enterprise Distributed Object Computing*) pour les systèmes distribués à base de composants, CWM pour les bases données, etc. A ce titre, nous proposons à travers cette thèse un DSL pour la modélisation d'interfaces d'applications multimodales et adaptatives. L'objectif est d'aboutir à un langage de description permettant de décrire les différents aspects d'une interface d'application : aussi bien graphique qu'interactif, pouvant s'adapter aux différents dispositifs d'interaction existants.

1.3.1.3. Le modèle PSM

Cette phase de l'approche MDA concerne la génération de code. Elle consiste en l'élaboration du modèle PSM (*Platform Specific Model*), appelé aussi le modèle de code ou de conception concrète. MDA considère que le code final d'application peut être facilement obtenu à partir du PSM. En effet, le code d'une application se résume à une suite de lignes textuelles, alors qu'un modèle de code est plutôt une représentation structurée. Une caractéristique importante des modèles de code est qu'ils intègrent les concepts des plateformes d'exécution. Pour élaborer des modèles de code, MDA propose, entre autres, l'utilisation des profils UML (voir section 3.2.2). Un profil UML est une adaptation du langage UML à un domaine particulier.

1.3.1.4. Du modèle CIM au modèle PSM

Comme énoncé en section 3.1, l'un des rouages principal de l'approche MDA est la transformation de modèles. En effet, la mise en production de MDA passe par la mise en relation des trois principaux modèles CIM, PIM et PSM, par diverses techniques de transformations. La section 4 présentera les techniques de transformations dans l'espace MDA. A partir de l'observation du processus de développement dans MDA, nous pouvons lister les principales transformations de modèles suivantes conformément à MDA :

- **Transformations de modèles CIM vers PIM.** Ces transformations permettent d'élaborer des PIM partiels à partir des informations contenues dans les CIM. L'objectif est de s'assurer que les besoins exprimés dans les CIM sont bien représentés dans les PIM.
- **Transformations de modèles PIM vers PIM.** Ces transformations permettent de raffiner les PIM afin d'améliorer la précision des informations qu'ils contiennent. En UML, de telles transformations, peuvent être, par exemple, l'enrichissement de classes UML à partir des diagrammes de séquences et des diagrammes d'états transitions.
- **Transformations de modèles PIM vers PSM.** Ces transformations permettent d'élaborer une bonne partie des modèles PSM à partir des modèles PIM. Elles garantissent la pérennité des modèles aussi bien que leur productivité et leur lien avec les plateformes d'exécution.
- **Transformations de modèles PSM vers code.** Ces transformations permettent de générer la totalité du code. Elles consistent en une traduction entre un formalisme structuré tel qu'un diagramme UML et un formalisme textuel représentant le code final.

1.4.2. Modélisation et métamodélisation dans MDA

Pour garantir la pérennité des modèles, MDA définit une architecture à 4 niveaux (*cf.* section 2), appuyée par des standards tels que MOF, UML, OCL, XMI.

Le consensus sur le langage UML fut décisif dans cette transition vers l'IDM et les techniques de production basées sur les modèles. Après l'acceptation du concept clé de métamodèle comme

langage de description de modèle, de nombreux métamodèles ont émergés afin d'apporter, chacun, leurs spécificités dans un domaine particulier (développement logiciel, entrepôt de données, procédé de développement, etc.). Afin de donner un cadre général pour leur description, il fut nécessaire d'offrir un langage de définition de métamodèles qui prit lui-même la forme d'un modèle : ce fut le méta-métamodèle MOF (*Meta-Object Facility*) [OMG, 2006].

Une des lacunes du langage UML est son incapacité à exprimer une sémantique opérationnelle [Blanc, 2005]. Dans le métamodèle UML, une opération n'est définie que par son nom, ses paramètres et les exceptions qu'elle émet. Le corps de l'opération ne peut donc être défini. Le langage OCL (*Object Constraint Language*) a été précisément défini par l'OMG pour combler cette lacune et permettre la modélisation du corps des opérations UML.

Le standard XMI permet de représenter les modèles sous forme de documents XML et favorise ainsi leur échange et donc leur pérennité. Le principe de fonctionnement de XMI consiste à générer automatiquement une spécification de structuration de balises XML (DTD ou XML schema) à partir d'un métamodèle. Il est ainsi possible de bénéficier du mécanisme de validation des documents XML.

Pour permettre l'adaptation d'UML à d'autres domaines et pour préciser la signification de cette adaptation, l'OMG a standardisé le concept de profil UML. Sur ce même principe, l'OMG ne pouvait ignorer le franc succès des langages dédiés (DSL). Ainsi, l'OMG préconise aussi l'utilisation de DSLs afin d'adapter les modèles à un domaine particulier.

Les sous sections suivantes sont dédiées à une présentation de ces différents standards de l'OMG et de l'approche MDA. Nous évoquerons aussi, au moment opportun, la place de ces standards dans nos travaux de recherche.

1.3.2.1. LES LANGAGES DE MÉTAMODÉLISATION

Dans l'architecture à 4 niveaux de MDA, les langages de métamodélisation se situent au niveau M3, et permettent de spécifier des métamodèles au niveau M2. Dans l'approche MDA, le langage MOF incarne le socle architectural. Dans l'environnement Eclipse, le langage Ecore joue le même rôle que MOF, il permet la spécification de métamodèles.

a. Le langage MOF

Tous les métamodèles publics de l'OMG sont réalisés avec la version 1.4 de MOF, qui est assez simple d'utilisation, contrairement à la version 1.3, qui nécessite une connaissance de CORBA, ou de la version actuelle 2.0, très complexe. Les métamodèles MOF 1.4 sont définis sous forme de classes. Les modèles conformes aux métamodèles sont considérés comme des instances de ces classes. La figure 1.6 présente une partie de ce que pourrait être un diagramme de classe représentant MOF1.4 [Blanc, 2005]. Nous évoquerons l'état actuel de la version 2.0 en fin de cette section.

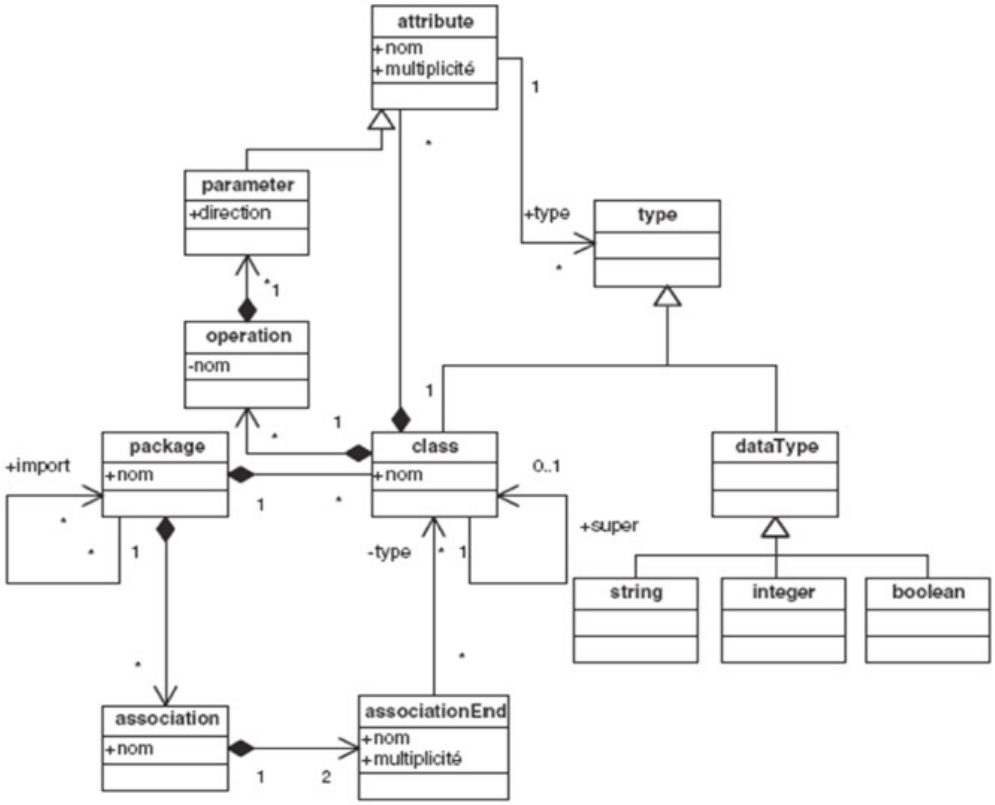

Figure 1.6. Représentation de MOF 1.4 sous forme de diagramme de classes.

Les principaux concepts de MOF 1.4 illustrés par la figure 1.6 sont les suivants :

- **Class.** Une classe (désignée par métaclasse dans MOF) permet de définir la structure de ses instances (désignées par méta-objets). Un ensemble de méta-objets constitue un modèle. Une métaclasse possède un nom, contient des attributs et des opérations aussi appelés méta-attributs et méta-opérations. Les désignations en "méta" permettent de discerner les entités du niveau M2 de celles du niveau M1.
- **DataType.** Un type de donnée permet de spécifier le type non-objet d'un méta-attribut ou d'un paramètre d'une méta-opération.
- **Association.** Pour exprimer des relations entre métaclasses, MOF 1.4 propose le concept de méta-association qui est une association binaire entre deux métaclasses. Une méta-association porte un nom, et chacune de ses extrémités peut porter un nom de rôle et une multiplicité.
- **Package.** Pour grouper entre eux les différents éléments d'un métamodèle, MOF 1.4 propose le concept de Package. Un Package, aussi appelé méta-package, est un espace de nommage servant à identifier par des noms les différents éléments qui le constituent.

La version MOF 2.0 est la version courante de MOF. Conceptuellement, son architecture ne diffère que très peu de celle de MOF 1.4 [Blanc, 2005]. Techniquement, l'un des objectifs de MOF 2.0 est de capitaliser les points communs existants entre UML et MOF au niveau des diagrammes de classes et d'en expliciter les différences. Pour y parvenir, il a été convenu qu'un métamodèle instance de MOF 2.0 pouvait ou non contenir des méta-associations. Pour faire la différence entre ces deux types de métamodèles, MOF 2.0 a été décomposé en deux parties : EMOF

(*Essential MOF*), pour l'élaboration de métamodèles sans association, et CMOF (*Complete MOF*), pour l'élaboration de métamodèles avec association [Blanc, 2005].

b. Le langage Ecore

Le langage de métamodélisation Ecore est proposé par la fondation Eclipse dans le cadre du projet EMF[5] (*Eclipse Modeling Framework*). Ce framework permet de générer automatiquement des interfaces Java à partir de métamodèles Ecore. La particularité d'EMF est qu'il se fonde sur la version MOF 2.0 et non MOF 1.4. Nos travaux de recherche se basent sur Ecore, c'est la raison pour laquelle nous le présentons ci-après. La figure 1.7 présente une partie du méta-métamodèle Ecore.

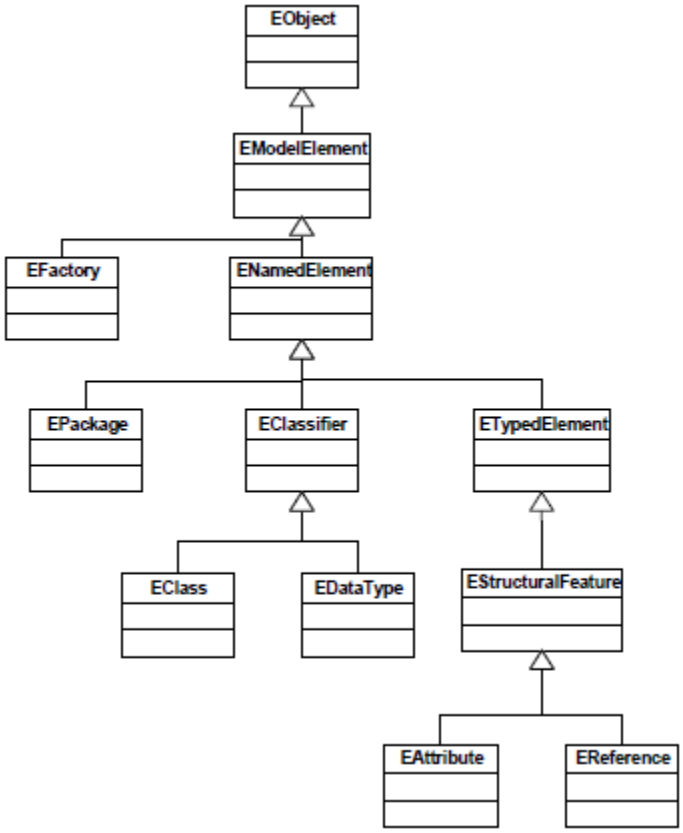

Figure 1.7. Sous-ensemble du méta-métamodèle Ecore.

Le méta-métamodèle Ecore ressemble fortement au méta-métamodèle EMOF, car il ne supporte que la notion de métaclasse sans méta-association. Pour exprimer une relation entre deux métaclasses, il faut utiliser des méta-attributs et les typer par des métaclasses. EMF impose cette contrainte pour faciliter la génération des interfaces Java. Le concept d'association n'existant pas en Java, il faudrait en effet une transcription particulière. Ainsi, les métamodèles conformes à Ecore sont composés d'EClass contenant des EAttribute et des EReference.

Les principaux concepts d'Ecore illustrés par la figure 1.7 sont les suivants :

[5] https://www.eclipse.org/modeling/emf/

- **EObject.** Elle représente n'importe quel élément, qu'il appartienne à un modèle ou à un métamodèle. Cette interface offre l'opération eClass(), qui permet d'obtenir l'EClass de l'élément et retourne un objet Java de type EClass. L'interface EObject offre aussi les opérations eGet() et eSet(), qui permettent respectivement de lire et d'écrire les valeurs des différentes propriétés de l'élément (attributs et références).
- **EClass.** L'interface EClass représente une métaclasse d'un métamodèle. Cette interface offre les opérations getEAttributes() et getEReferences(), qui permettent d'obtenir la liste, respectivement, de tous les attributs et références contenus dans la métaclasse. Elle offre aussi l'opération getEStructuralFeature(), qui permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom.
- **EPackage.** L'interface EPackage représente le moyen d'accès à toutes les EClass définies dans un package. Elle offre l'opération getEClassifier(String name), qui permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom.

1.3.2.2. LES LANGAGES DE MODÉLISATION

Les langages de modélisation ou métamodèles sont situés au niveau M2 de l'architecture MDA et permettent de spécifier des modèles. Nous distinguons deux grandes catégories : les spécifiques ou *Domain-Specific 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.

a. Le métamodèle UML et les profils UML

La première version de UML a été publiée en 1997 par l'OMG. Depuis, UML est devenu la référence pour la création de modèles et le métamodèle d'UML est le plus connu de l'approche MDA. Même si nous n'utilisons pas le standard UML dans nos travaux, nous discuterons dans cette section du métamodèle UML, des profils UML ainsi que de la position de chacun dans le processus de développement MDA.

Le métamodèle UML

Nous n'allons pas détailler ici le métamodèle d'UML, il existe aujourd'hui une multitude de documents pour cela. Une description plus détaillée est présentée dans [Blanc, 2005] [OMG, 2011]. Le métamodèle UML définit la structure que doit avoir tout modèle UML. Il précise, par exemple, qu'une classe UML peut avoir des attributs et des opérations. La figure 1.8 montre un fragment du métamodèle UML et précise au niveau métamodèle par exemple qu'une association UML est composée de deux extrémités, chacune représentant une classe.

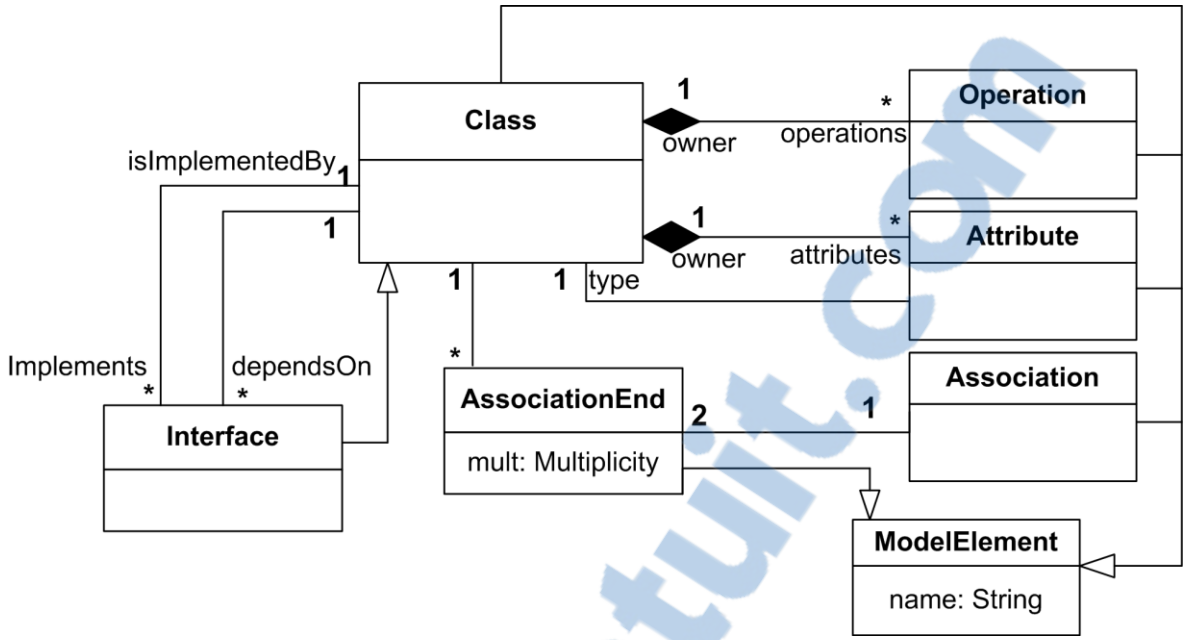

Figure 1.8. Un fragment du métamodèle du langage UML.

D'un point de vue plus conceptuel, le métamodèle UML permet de concevoir des modèles décrivant des applications objet selon différentes vues. Par exemple, les diagrammes de classes permettent de décrire la partie statique des applications alors que les diagrammes de séquences ou d'états transitions permettent d'en définir la partie dynamique. Comme évoqué précédemment, les modèles UML sont indépendants des plates-formes d'exécutions. Ils sont utilisés pour décrire aussi bien les applications Java, Bases de données ou C#. Plusieurs outils proposent des générateurs de code vers différents langages de programmation.

Les profils UML

UML est un langage général, et a donc été conçu pour prendre en charge une grande variété d'applications et de contextes de mise en œuvre. Cependant, même avec cette intention d'être général, UML ne peut pas couvrir tous les contextes et offre ainsi un mécanisme d'extensibilité basé sur les profils. Un profil permet la personnalisation d'UML pour prendre en charge des domaines spécifiques qui ne peuvent pas être représentés avec UML dans son état original. Le profil est constitué de trois concepts : les stéréotypes, les contraintes et les valeurs marquées. Ces éléments permettent d'adapter la sémantique sans changer le métamodèle d'UML.

D'un point de vue technique, un profil est un package stéréotypé spécifique à un domaine. Un stéréotype est défini comme une extension d'une métaclasse 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. La figure 1.9 montre un exemple de profil pour la modélisation de composants EJB[6] ; 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.

[6] Entreprise JavaBeans : http ://www.oracle.com/technetwork/java/javaee/ejb/index.html

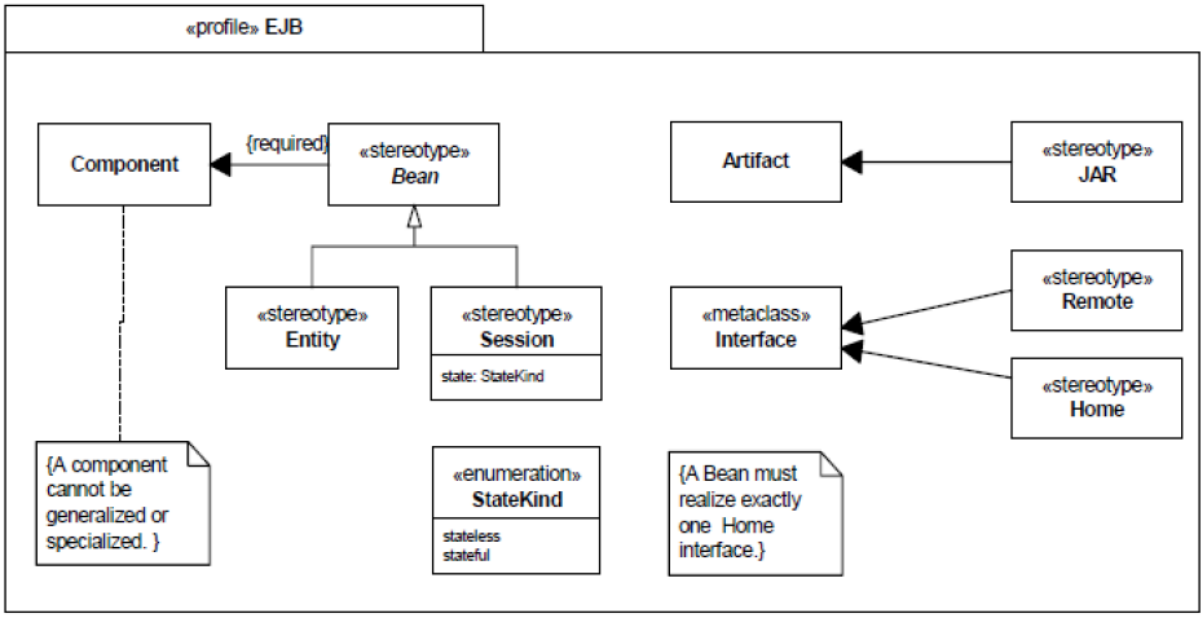

Figure 1.9. Exemple d'un profil UML pour les EJB [OMG, 2011].

Dans [Laforcade, 2004], l'auteur présente une synthèse des différents éléments structurant un 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 métaclasse spécifique du métamodèle de référence. Dans un profil UML, le stéréotype crée une métaclasse UML virtuelle basée sur la métaclasse UML existante et peut également aussi spécifier des contraintes additionnelles ou des valeurs marquées requises ;
- **les définitions de valeurs marquées.** 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. La valeur marquée agit comme un attribut d'une métaclasse 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 métaclasse 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) (*cf.* section 3.3). Mais les contraintes peuvent être spécifiées de manière informelle en langage naturel ;
- **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 dans un but spécifique. Par exemple, le profil CORBA exprime comment un modèle UML peut être transformé en une implémentation CORBA ;
    - **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 d'un autre profil. Nous verrons dans la section suivante qu'un DSL est un métamodèle construit à partir de zéro.

b. Les DSL

L'utilisation d'un DSL aide à se concentrer sur un sujet déterminé en fixant un cadre précis. 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 avec une approche basée sur un DSL.

La définition d'un DSL est une tâche dont la complexité est proportionnelle à celle du domaine cible. La fabrication d'un DSL est principalement axée sur deux phases : l'élaboration de la syntaxe abstraite et la définition de la syntaxe concrète.

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 et environnements de métamodélisation tel que MOF, Ecore, Kermeta[7], Xtext[8] ou encore MetaEdit+[9] offrent les concepts et relations élémentaires avec lesquels il est possible de décrire un métamodèle représentant cette syntaxe abstraite. La majorité de ces langages reposent sur les mêmes constructions élémentaires (*cf.* figure 1.10).

[7] http://www.kermeta.org
[8] http://www.eclipse.org/Xtext/
[9] http://www.metacase.com/fr/products.html

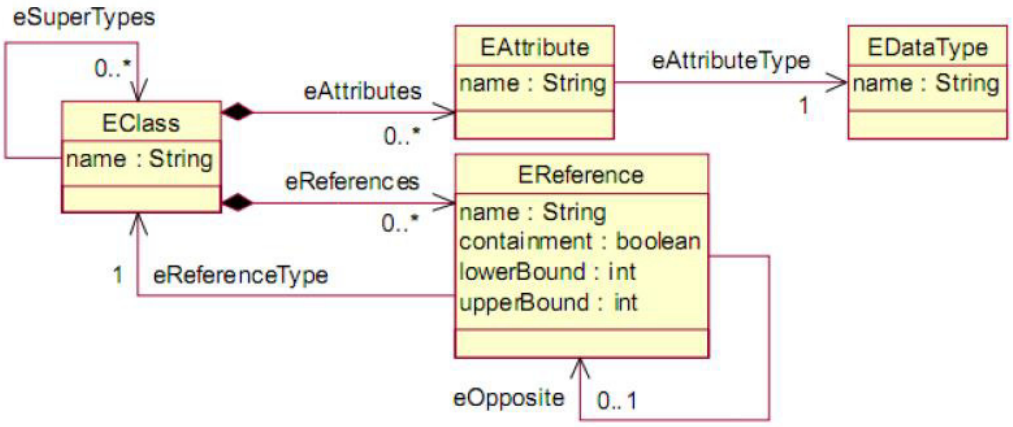

Figure 1.10. Concepts de base pour la métamodélisation (EMF/Ecore).

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. Pour exprimer ces règles, l'OMG définit le langage OCL (*Object Constraint Language*) (*cf.* section 3.3). 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 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 aspect visuel d'un DSL est sa syntaxe concrète.

Syntaxe concrète

Chaque élément d'un modèle (instance d'un métamodèle et donc du DSL) a une représentation/forme graphique ou textuelle particulière. La définition de la syntaxe concrète est donc la définition de cette représentation textuelle ou graphique. 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.

Le projet *Eclipse Modeling*[10] 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. Il existe aussi d'autres projets qui permettent de définir des représentations textuelles. Nous citons par exemple Xtext et TCS[11]. Nous utilisons dans nos travaux de recherches l'outil Xtext sous l'environnement Eclipse pour la définition de notre DSL textuel.

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

Exemple de fabrication d'un DSL

[10] http://www.eclipse.org/modeling/
[11] Textual Concrete Syntax: http://www.eclipse.org/gmt/tcs/

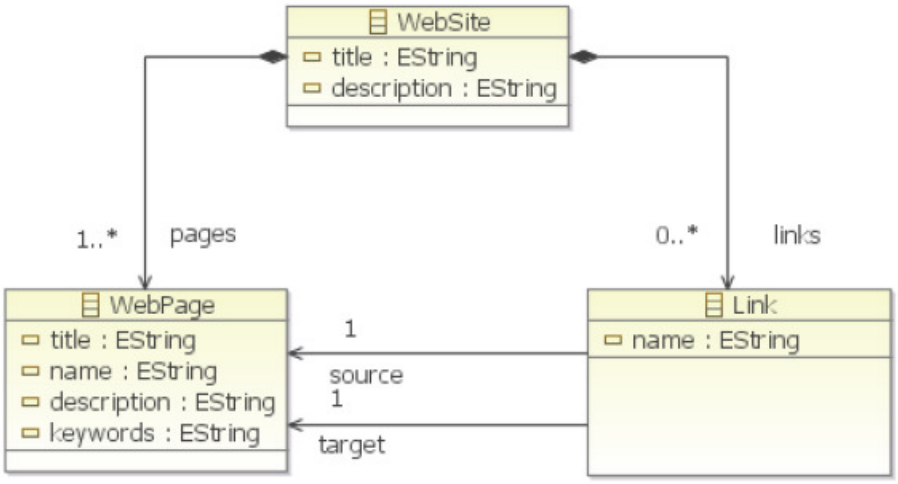

Figure 1.11. Exemple d'un DSL pour la modélisation de sites Web.

La figure 1.11 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 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.

1.4.3. OCL, ACTION SEMANTICS ET XMI

Le langage OCL [Warmer, 2002], *Object Constraint Language*, a été défini par l'OMG pour permettre la modélisation du corps des opérations dans un modèle UML. Il est utilisé pour exprimer des pré et post-conditions sur les opérations. OCL dispose d'un formalisme textuel, on parle d'expression OCL. Les expressions OCL ne génèrent aucun effet de bord. C'est-à-dire que l'évaluation d'une expression OCL n'entraîne aucun changement d'état dans le modèle auquel

elle est rattachée. La valeur retournée est vrai ou faux. L'évaluation de la contrainte OCL permet de la sorte de savoir si la contrainte est respectée ou non. Une expression OCL porte sur un élément du modèle. Pour être évaluée, une expression OCL doit être rattachée à un contexte, qui doit être directement relié à un modèle. L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur du solde doit être positive avant toute invocation de l'opération debit, c'est donc une pré-condition :

```
1 context CompteBancaire :: debit():Integer
2 pre: solde>0
```

Le métamodèle OCL permet de représenter n'importe quelle expression OCL sous forme de modèle. Pour des raisons de simplicité d'utilisation, OCL reste avant tout un langage textuel, ce qui rend son intégration aux modèles exprimés à l'aide de DSL textuels tout aussi simple.

Jusqu'à sa version 1.4, UML était très critiqué parce qu'il ne permettait pas de spécifier des créations, des suppressions ou des modifications d'éléments de modèles. Ces actions ne pouvant pas non plus être spécifiées à l'aide du langage OCL, puisque celui-ci est sans effet de bord. Il était nécessaire de standardiser un nouveau langage. C'est ce qui a donné naissance au langage AS (*Action Semantics*) [Blanc, 2005]. L'objectif de *Action Semantics* est de permettre de définir des actions. Une action au sens AS est une opération sur un modèle qui fait changer l'état du modèle. Grâce à ces actions, il est possible de modifier les valeurs des attributs, de créer ou de supprimer des objets, de créer de nouveaux liens entre les objets, etc. Ainsi le concept d'*Action Semantics* permet de spécifier pleinement le corps des opérations UML. Au départ AS était un langage développé à part entière hors de UML puis inclus dans la version 1.5. Dans UML 2.0, il est totalement intégré au métamodèle. AS n'est standardisé que sous forme de métamodèle, et aucune syntaxe concrète n'est définie, contrairement à OCL. Ainsi, le manque de format concret (textuel), rend l'utilisation de AS complexe. C'est la raison pour laquelle AS n'est pas encore pleinement exploité, contrairement à OCL.

Les modèles étant des entités abstraites, ils ne disposent pas intrinsèquement de représentation informatique. L'OMG à donc décidé de standardiser XMI (*XML Metadata Interchange*), qui permet l'échange de modèles sérialisés en XML. XMI se focalise sur les échanges de métadonnées conformes au standard MOF. L'objectif de XMI est de permettre de sérialiser et d'échanger des métamodèles MOF et des modèles basés sur ces métamodèles sous forme de fichiers en utilisant des dialectes XML. Il permet aussi de sérialiser des métamodèles qui sont créés avec Ecore. Ceci est possible car XMI ne définit pas un dialecte XML unique, mais un ensemble de règles qui permettent de créer une DTD ou un schéma XML pour différents métamodèles. Ainsi, les métamodèles conformes à MOF ou à Ecore et leurs modèles respectifs peuvent être portables en utilisant XMI. Toutes nos expérimentations autour de la plateforme Eclipse ont utilisées ce format d'échange pour le traitement des modèles et des métamodèles.

1.4. LA TRANSFORMATION DE MODÈLES DANS MDA

La transformation de modèles est la troisième problématique clé de MDA et de l'IDM en général. Elle permet de rendre les modèles productifs et d'obtenir une traçabilité entre des modèles très abstraits, proches des besoins exprimés par les utilisateurs, et des modèles très concrets, proches des plateformes d'exécutions. Nous avons vu en section 3.1 lors de la présentation du processus de développement dans MDA les différents types de transformations. Quel que soit

son type (PIM vers PSM, PIM vers PIM, etc.), une transformation de modèles s'apparente toujours à une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un ensemble de modèles. Nous avons vu en section 2.4 que si une transformation de modèles s'exécute au niveau des modèles, elle se spécifie au niveau des métamodèles. Les métamodèles permettent dans le processus de transformation MDA de définir les structurations possibles des modèles source et cible et de servir de base pour la définition des règles de transformation.

1.4.1. LES DIFFÉRENTES APPROCHES DE TRANSFORMATIONS DE MODÈLES

Trois principales approches ont été développées pour la mise en œuvre des transformations de modèles. La différence entre ces trois approches est liée à la manière dont les règles de transformations sont spécifiées [Blanc, 2005] :

- **Approche par programmation.** Un langage de programmation orienté objet est mis en œuvre pour programmer une transformation de modèle de la même manière que l'on développe n'importe quelle application. Les transformations selon cette approche sont donc des applications qui ont la spécificité de manipuler des modèles. Ces applications utilisent des interfaces de gestion de modèles offrant les opérations nécessaires pour la manipulation des modèles. Les standards et framework MOF, JMI et EMF disposent de ces interfaces. Cette approche est très utilisée car elle bénéficie de la richesse des langages de programmation et des environnements de développement fortement outillés.
- **Approche par template.** L'approche par template consiste à définir des canevas pour les modèles cibles souhaités. Ces canevas sont des modèles cibles paramétrés ou modèles templates. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d'un modèle source. Cette approche par template est implémentée par exemple dans *Softeam MDA Modeler* et Acceleo. L'approche par template, même si elle est complexe à mettre en œuvre, est très prometteuse, car elle offre une solution favorisant la réutilisabilité des modèles de transformations.
- **Approche par modélisation.** Cette approche consiste, quant-à-elle, à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plateformes d'exécution. Le standard MOF 2.0 QVT de l'OMG a été élaboré dans ce cadre et a pour but de définir un métamodèle permettant l'élaboration des modèles de transformation de modèles. Le langage de transformation de modèles ATL est aussi dans cette mouvance.

1.4.2. LANGAGES ET STANDARD POUR LA TRANSFORMATION DE MODÈLES

Afin de donner un cadre normatif pour l'implantation des différents langages dédiés à la transformation de modèle, l'OMG a défini les standards QVT (*Query/View/Transformation*) [OMG, 2016] et MOFM2T (*MOF Models to Text Transformation Language*) [OMG, 2008]. MOFM2T est utilisé pour exprimer des transformations de modèle vers texte (M2T), là où QVT est utilisé pour exprimer des transformations de modèle vers modèle (M2M). Durant ces dernières années, de nombreux outils ont été créés pour mettre en œuvre ces langages. Un choix entre ces différents outils peut être basé sur des études comparatives telles que [Diaw et al., 2010], [Combemale, 2008], [Czarnecki et Helsen, 2006] permettant d'évaluer les fonctionnalités fournies par chacun.

Dans ce qui suit, nous présentons les langages *MOF Model To Text Transformation Language* (MOFM2T) et QVT ainsi que les outils qui leurs sont associés.

1.4.2.1. LE STANDARD MOFM2T

Le langage *MOF Model to Text Transformation Language* (MOFM2T) est la norme proposée par l'OMG pour la transformation des modèles vers une représentation textuelle [OMG, 2008]. Ce langage utilise une approche basée sur les templates dans laquelle le texte généré à partir de modèles est spécifié comme un ensemble de templates de texte paramétrés avec des éléments du modèle. Un template spécifie pour chaque élément du modèle source, les éléments qui leurs correspondent au niveau du texte (code). Les règles de transformation sont spécifiées sur les entités des métamodèles sélectionnées par des requêtes qui permettent de sélectionner et d'extraire des valeurs à partir des modèles. Ces valeurs sont par la suite transformées en fragment de code en utilisant la bibliothèque de définition du langage cible. Les templates peuvent être composés pour répondre aux transformations complexes. Les grandes transformations peuvent être structurées en modules.

1.4.2.2. LE STANDARD QVT

QVT est l'acronyme de *Query, Views, Transformation* ; c'est un langage graphique de transformation et standardisé par l'OMG [OMG, 2008]. Le métamodèle de QVT est conforme à MOF et OCL et utilisé pour la navigation dans les modèles. Le métamodèle fait apparaître trois sous-langages pour la transformation de modèles (*cf.* figure 1.12), caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride). Les langages *Relations* et *Core* sont tous deux déclaratifs mais placés à différents niveaux d'abstraction. L'un des buts de *Core* est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation de *Relations* vers *Core*. Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour répondre à ce besoin, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou *black box*). *Operational Mappings* étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord. Les langages de style impératif sont mieux adaptés pour des transformations complexes qui comprennent une composante algorithmique importante. Par rapport au style déclaratif, ils ont l'avantage de gérer les cas optionnels dans une transformation.

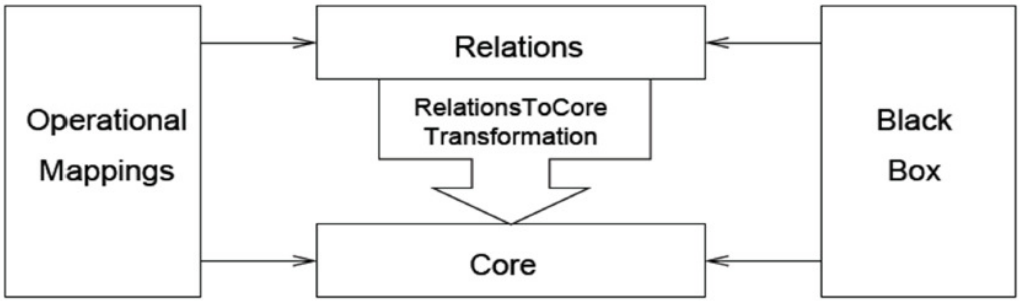

Figure 1.12. Architecture du standard QVT [OMG, 2008].

Dans la littérature, de nombreux outils implémentant le standard QVT ont vu le jour. Parmi eux, nous pouvons citer : SmartQVT[12], MediniQVT[13], etc. Parmi les plus aboutis, nous pouvons citer

[12] https://smartqvt.soft112.com/

Kermeta[14] de l'IRISA-INRIA Rennes. Une description en Kermeta est assimilable à un programme issu de la fusion d'un ensemble de métadonnées (EMOF) et du métamodèle d'action AS qui est maintenant intégré dans UML 2.0 Superstructure. Le langage Kermeta est donc une sorte de dénominateur commun des langages qui coexistent actuellement dans le paysage de l'IDM. Ces langages sont les langages de métadonnées (EMOF, ECORE,. . .), de transformation de modèles (QVT, ATL,. . .), de contraintes et de requêtes (OCL), et d'actions (*Action Semantics*).

1.5. DISCUSSION ET 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. Le développement des systèmes d'information dirigés par les modèles a pour principal objectif de concevoir des applications en séparant les préoccupations et en plaçant les notions de modèles, métamodèles et transformations de modèles au centre du processus de développement. Nous avons introduit dans ce chapitre ces notions qui sont à la base des principes généraux de l'IDM, c'est-à-dire la métamodélisation d'une part et la transformation de modèles d'autre part. Ces deux axes constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement. Aujourd'hui, l'IDM reste fortement marquée par l'approche MDA et les standards de l'OMG, c'est pourquoi nous avons focalisé notre présentation et nos travaux de recherche sur cette approche.

Nous avons présenté la possibilité de définir de nouveaux métamodèles afin de proposer des langages de modélisation spécifiques à un domaine (DSL). 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 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*[15] et inclut un nombre important de composants *open source* couvrant la métamodélisation (syntaxe abstraite), la génération d'éditeur graphique (syntaxe concrète) ou encore la génération de code.

Pour nos travaux, nous souhaitons bénéficier des avantages offerts par l'IDM et par les outils qui y sont associés pour proposer un langage de description (DSL) d'interface d'application permettant la modélisation d'interface graphique pouvant s'adapter aux différentes plateformes d'interactions existantes, tout en offrant un confort d'utilisation optimal aux utilisateurs. Ce modèle est mis en scène dans une approche MDA. Afin de compléter la compréhension du contexte technologique et scientifique de notre thèse, le chapitre suivant présente les différents éléments de gestion de l'interaction homme-machine à prendre en compte lors de l'élaboration d'un tel langage.

[13] http://projects.ikv.de/qvt
[14] http://www.kermeta.org
[15] http://www.eclipse.org/modeling/

CHAPITRE 2
DE LA PROGRAMMATION DES IHM À L'INGÉNIERIE DES IHM

" Ingénierie : conception, étude globale d'un projet sous tous ses aspects, coordonnant les études particulières des spécialistes. "
Petit Robert (1993)

2.1. INTRODUCTION

Les avancées technologiques telles que la miniaturisation des microprocesseurs et des capteurs, le succès incontestable des technologies connectées, portés par l'objectif "d'accès à l'information par tous, partout, à tout moment", ouvrent un vaste champ de possibilités. En effet, l'émergence des smartphones a apporté une grande variété de capteurs qui ont permis l'apparition de nouvelles modalités d'interaction telle que l'interaction en inclinant le téléphone ou en modifiant son orientation. Cependant, ces dispositifs mobiles ont apporté des restrictions aux interfaces graphiques et à l'interaction en générale. Cela est dû à leurs petits écrans, à l'interaction avec le doigt qui n'est pas assez précise, au clavier virtuel inconfortable, à l'obligation d'utiliser les yeux et les deux mains pour certaines interactions, etc. Dans ce contexte, ces systèmes doivent relever des défis majeurs dans leur développement et leur usage.

Du point de vue de l'IHM, les défis sont liés à l'adaptation au contexte d'usage et à la prise en compte des interactions nouvelles, multiples et complexes (diversité des dispositifs, multimodalité, utilisabilité). La recherche en IHM est un bien vaste domaine et, selon J.S. Sottet (2014), il convient de considérer cette thématique selon au moins deux perspectives : celle des IHM innovantes et celle de l'Ingénierie des IHM. Les IHM innovantes font références à l'apparition de nouveaux dispositifs et techniques d'interaction ouvrant la voie à de vastes champs de possibilités telles que la réalité mixte ou encore la multimodalité. En ingénierie des IHM, il s'agit de faciliter la conception des IHM indépendamment de leur degré d'innovation tout en préservant un confort d'utilisation optimal.

Nous positionnons nos travaux de thèse à la croisée de ces deux chemins et proposons une nouvelle approche pour l'adaptation des IHM aux différents dispositifs d'interaction, destinée aussi bien aux utilisateurs finaux qu'aux concepteurs d'IHM : les utilisateurs finaux pourront interagir grâce à une interface d'application optimisée en fonction du contexte d'usage, les concepteurs d'IHM pourront les adapter pour qu'elles s'exécutent dans différents contextes. La multitude des contextes d'usage et des technologies qui doivent être maîtrisés et mis en œuvre pour la réalisation d'IHM soulève une question de contexte d'ingénierie.

A ce titre, nous poursuivons ce chapitre en faisant la distinction entre contexte d'usage et contexte d'ingénierie dans la section 2. La section 3 présente la terminologie de l'interaction utilisée et que l'on retrouve tout au long de cette thèse. La section 4 traite des deux dimensions

de la recherche en IHM qui nous intéresse dans cette thèse à savoir la multimodalité et la plasticité. Puis, nous clôturons ce chapitre en section 5.

2.2. CONTEXTE D'UTILISATION ET CONTEXTE D'INGÉNIERIE

L'étude de l'interaction homme - machine étant un domaine très vaste, cette section permet de faire la distinction entre deux notions complémentaires mais sans cesse en dualité. Afin de permettre au lecteur de mieux situer les travaux effectués dans le cadre de cette thèse.

Dans ses travaux de thèse de doctorat, J.S. Sottet [Sottet, 2014] fait la distinction entre contexte d'usage et contexte d'ingénierie. Les appellations contexte d'usage et contexte d'ingénierie font référence respectivement aux utilisateurs et aux ingénieurs d'IHM. Les premiers utilisent les IHM, les seconds les conçoivent et les développent. Ce sont les acteurs influant et agissant sur le logiciel.

Par conséquent, on appelle "contexte d'usage" le triplet :

- utilisateur (amateur, expert, fatigué, attentif, etc.) ;
- environnement d'utilisation (rue, bureau, maison, transport en commun, etc.) ;
- plateforme d'interaction (ordinateur, smartphone, tablette, etc.).

Aussi, on appelle "contexte d'ingénierie" le triplet :

- ingénieur (spécialiste métier, spécialiste IHM, etc.) ;
- environnement d'ingénierie (équipe de développement, entreprise, travail individuel, etc.) ;
- plateforme d'ingénierie (UML/Rose, Visual Studio, Eclipse, etc.).

Cette distinction étant faite, la problématique à l'origine des motivations qui portent nos travaux consiste, d'un point de vue du contexte d'usage, dans la prise en compte de la plateforme d'interaction pour le développement d'interfaces d'applications, notamment les ressources de communication et d'interaction disponibles. La prise en compte de l'environnement d'utilisation étant hors de portée de nos travaux, celle-ci reste peu formalisée dans la littérature et fait référence au milieu dans lequel s'exerce l'interaction. D'un autre côté, la prise en compte du contexte d'usage pose un problème de contexte d'ingénierie de sorte que le choix de l'environnement et de la plateforme d'ingénierie doit être de telle sorte à permettre le développement d'une interface adaptative en fonction du contexte d'usage. Ainsi, les deux contextes s'influencent mutuellement.

2.3. ETUDE ET TERMINOLOGIE DE L'INTERACTION

Le domaine de l'interaction, homme-machine regroupe de nombreux termes dont le sens est parfois ambigu ou mal interprété. Cette section vise à présenter des définitions destinées à préciser le sens de ces termes.

La littérature associe plusieurs termes au domaine de l'IHM dont les trois principaux sont : modalité, mode et multimodalité. Suivant ces termes et leur définition, une interaction peut être schématisée de la façon suivante :

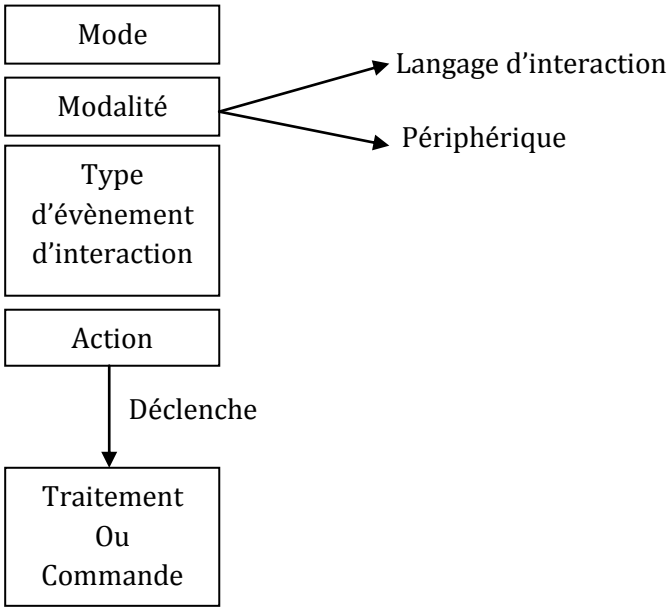

Figure 2.1. Définition de l'interaction et les différents termes utilisés dans cette thèse.

En se basant principalement sur les travaux de [Bellik, 1992], nous considérons les définitions suivantes dans la suite de nos travaux.

2.3.1. MODE

Le mode fait référence aux cinq sens de l'être humain : le toucher, l'ouïe, la vue, l'odorat, le goût (réception d'information), et aux différents moyens d'expression humains : le geste, la parole (émission d'information). Il définit la nature des informations servant pour la communication (mode visuel, mode sonore, mode gestuel…).

2.3.2. MODALITÉ

Une modalité est une forme concrète particulière d'un mode de communication. Par exemple, le bruit, la musique, la parole sont des modalités du mode sonore. Dans [Nigay et Coutaz, 1997], une modalité est définie par le couple <d, r> où "d" représente le périphérique d'entrée/sortie et "r" le langage d'interaction.

Modalité	Mode	Langage d'interaction	Périphérique
Parole	Vocal	Langage (pseudo) naturel	Microphone
Pointage tactile	Tactile	Manipulation directe	Ecran tactile

Tableau 2.1. Exemples de modalités d'interaction en entrée.

Modalité	Mode	Langage d'interaction	Périphérique
Synthèse vocale	Audio	Langage (pseudo) naturel	Haut-parleurs
Affichage de widgets	visuel	widgets	Ecran

Tableau 2.2. Exemples de modalités d'interaction en sortie.

Pour reprendre les termes de Bellik, nous pouvons définir la modalité comme une forme d'un mode particulier référant un des sens humains et utilisant un langage d'interaction et un périphérique (physique ou virtuel).

Aussi, en fonction du contexte d'utilisation, une modalité est dite "passive" lorsqu'elle est utilisée inconsciemment par l'utilisateur. Et à l'opposé, une modalité est dite "active" lorsque cette dernière est utilisée consciemment par l'utilisateur. Par exemple, la localisation de l'utilisateur avec un GPS est le plus souvent considérée comme passive, car elle n'a pas besoin de l'attention de l'utilisateur. Mais si l'utilisateur se déplace consciemment sur une carte, le positionnement par GPS est considéré come une modalité active.

2.3.3. MULTIMODALITÉ

Une application est dite multimodale si son interface intègre deux ou plusieurs modalités d'interaction en entrée et/ou en sortie en référant un ou plusieurs modes de communication.

L'intérêt de ce type d'interfaces réside dans le fait qu'elles offrent en théorie un moyen d'interaction plus adapté aux besoins des utilisateurs qui peuvent choisir une ou plusieurs modalités combinées selon leurs préférences et en fonction de la tâche effectuée. Ce qui réduit le nombre d'erreurs et rend plus robuste l'interprétation des informations (l'utilisation des modalités en parallèle peut augmenter le taux de reconnaissance correcte d'interaction).

2.4. MULTIMODALITÉ ET PLASTICITÉ

Cette section présente deux propriétés clés des interfaces homme-machine : la multimodalité et la plasticité des IHM. Nous en proposons des définitions permettant de comprendre la nature de ces notions. De même que nous tâchons de montrer comment la multimodalité participe à la plasticité des IHM.

Comme nous l'avons introduit en début de cette thèse, notre proposition est destinée à la modélisation d'interfaces d'interaction plastiques, multi-cibles, multimodales, utilisant des langages de description d'interface utilisateur. En combinant la plasticité et la multimodalité, HCIDL améliore la convivialité des interfaces utilisateur grâce à un comportement adaptatif en fournissant aux utilisateurs finaux un ensemble d'interactions adapté aux entrées/sorties des terminaux et une disposition de l'interface graphique optimale.

Les notions de plasticité et de multimodalité étant chacune un domaine de recherche distinct et bien vaste, nous nous limiterons donc à une définition correspondant au contexte des travaux effectués dans cette thèse.

2.4.1. IHM MULTIMODALES

2.4.1.1. MULTIMODALITÉ ET SES CONCEPTS

Le concept d'interfaces multimodales a été introduit par Richard Bolt (1980) dans son système «put-that-there», qui combine des commandes vocales avec des techniques de pointage (voir Figure 2.2). D'autres prototypes ont ensuite été construits pour tester l'utilisation combinée de la manipulation directe et du langage textuel naturel [Cohen et al., 1989], ou des gestes et de la parole [Weimer et Ganapathy, 1989]. Plus tard, des techniques de traitement vidéo en temps réel ont été utilisées pour améliorer la reconnaissance vocale en lisant les mouvements des

lèvres et pour contrôler une application de visualisation d'images panoramiques en combinant les mouvements oculaires et les commandes vocales [Yang et al., 1998]. Selon Laurence Nigay [Nigay et Coutaz, 1993], la multimodalité est la capacité d'un système à communiquer avec un utilisateur en utilisant différents types de canaux de communication. L'utilisation de canaux de communication multiples permet à chaque canal de compenser les faiblesses des autres, en particulier lors de la résolution d'ambiguïtés, mais permet également la propriété de redondance, dans laquelle une tâche peut être accomplie de diverses manières. Cette redondance des entrées est souvent souhaitable, notamment lorsque les contraintes liées à l'environnement sont variables. Par exemple, l'utilisation redondante d'un microphone et la saisie manuelle via un clavier permettent à l'utilisateur de définir son texte à l'aide de la reconnaissance vocale, ou si cette fonctionnalité est manquante, d'utiliser le clavier.

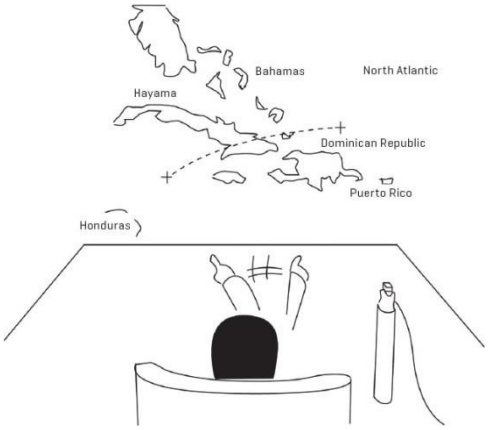

Figure 2.2. Le système "Put-that-there" de Richard Bolt (1980).

La recherche en multimodalité était initialement axée sur la multimodalité d'entrée (de l'utilisateur au système), impliquant plusieurs dispositifs d'entrée physique [Nigay et Coutaz, 1993] [Bourguet et Caelen, 1992] [Gourdol et al., 1992] [Mignot et al., 1993] [Martin, 1994] [Schüssel et al., 2012]. Peu d'études se sont intéressées à la conception de la multimodalité en sortie (du système à l'utilisateur). Néanmoins, nous notons deux thèses, Frédéric Vernier (2001) et Cyril Rousseau (2006), toutes deux consacrées à l'interaction multimodale en sortie. À ce stade, des outils pour la conception et la mise en œuvre d'interfaces multimodales sont disponibles. Le tableau 2.3 montre un ensemble d'exemples de modalités en entrée et en sortie :

Exemple de capteurs en entrée	**Exemple de capteurs en sortie**
GPS	Haut-parleur
Camera	Vibration
Accéléromètre	Synthèse vocale
NFC	
Gyroscope	
Boussole	
Capteur de proximité	
Capteur de lumière	

Tableau 2.3. Exemple de capteurs en entrée et en sortie [Elouali et al., 2012].

Des exemples de tels outils sont le *Framework for Adaptive Multimodal Environments* (FAME) [Duarte et Carrico, 2006], des approches basées sur des règles présentées dans [Bellik et Teil,

1992], ou ceux qui offrent des solutions plus complètes, comme PetShop [Bastide et al., 1998], mais qui nécessitent de solides compétences en programmation afin de coder les mécanismes spécifiques à chaque agent. Nous pouvons également noter des approches récentes où l'objectif est de fournir un moyen de concevoir des interactions multimodales comme les travaux de Hesenius et al. (2014). Les auteurs démontrent une approche pour incorporer l'interaction multimodale dans des tests d'acceptation d'utilisateur basés sur Gherkin pour spécifier le comportement d'un système, permettant d'inclure différentes modalités d'interaction en utilisant le langage de description de geste formel GeForMT. Cette approche est strictement orientée vers la communication humaine et ne peut pas être utilisée pour configurer des applications. Guedes et al. (2017) proposent d'étendre les langages multimédias pour soutenir le développement d'interfaces utilisateur multimodales sous la forme d'un framework de programmation de haut niveau. Le framework intègre des modalités d'utilisateur, générées par l'utilisateur (par exemple, discours, gestes) et consommées par l'utilisateur (par exemple, audiovisuel, haptique), en utilisant le langage déclaratif NCL (*Nested Context Language*) pour la spécification d'applications multimédias interactives.

2.4.1.2. TYPES D'INTERACTION ET DE COMBINAISON ENTRE LES MODALITÉS

Une interface multimodale intègre plusieurs modalités d'interaction différentes en entrée et/ou en sortie. Les types de combinaison entre modalités ont été présentés et formalisés sous le nom de propriétés CARE (*Complementarity, Assignment, Redundancy, Equivalence*) par Laurence Nigay et Joelle Coutaz en 1997 [Nigay et Coutaz, 1997], puis étendues sous le nom de TYCOON (*TYpes and goals of COOperatioN*) par J-C Martin en 1999 [Martin, 1999].

Pour étudier la multimodalité, il était nécessaire de définir les relations entre les modalités et la manière dont la combinaison peut être utilisée par le système. C'est à ce niveau qu'interviennent les types de combinaisons entre modalités CARE et TYCOON. Elles permettent de gérer différentes modalités d'interaction entre l'utilisateur et le système et la coopération établie entre elles. Le Tableau 2.4 et le Tableau 2.5 montrent quelques exemples de modalités d'interaction en entrée et en sortie :

Modalité	**Langage d'interaction**
Accélération	Manipulation directe
Localisation	Positionnement par GPS
Discours	Langage pseudo naturel
Pointage tactile	Manipulation directe
Orientation	Manipulation directe

Tableau 2.4. Exemples de modalités d'interaction en entrée [Duarte et Carrico, 2006].

Modalité	**Language d'interaction**
Synthèse vocale	Langage pseudo naturel
Affichage de widget	Widgets

Tableau 2.5. Exemples de modalités d'interaction en sortie [Duarte et Carrico, 2006].

Cette section présente les approches CARE et TYCOON auxquels nous nous sommes intéressés lors de nos recherches.

a. Les propriétés CARE

Les propriétés CARE décrivent la façon dont les modalités d'interaction peuvent coopérer dans un système multimodal. Une modalité d'interaction dans CARE est définie par le couple <d, r> où "d" représente le périphérique physique et "r", le langage d'interaction (*cf.* section 3.2).

Afin de donner une définition formelle des propriétés CARE, il est nécessaire de définir les termes suivants :

- Etat : en anglais *State*, est un ensemble de propriétés qui peuvent être mesurées à un moment donné pour caractériser une situation.
- But : en anglais *Goal*, est un état qu'un agent a l'intention d'atteindre.
- Agent : en anglais, *Agent*, est une entité capable d'initier l'exécution d'actions (l'utilisateur ou le système).
- Modalité : en anglais *Modality*, est une méthode d'interaction qu'un agent peut utiliser pour atteindre un but.
- Relation temporelle : en anglais *Temporal relationship*, caractérise l'utilisation dans le temps d'un ensemble de modalités. L'utilisation de ces modalités peut se produire simultanément ou séquentiellement dans une intervalle de temps.

Sur la base des termes ci-dessus, les propriétés CARE sont définies comme suit :

Complémentarité

Les modalités d'un ensemble M sont utilisées de façon complémentaire pour atteindre l'état s' à partir de l'état s dans une fenêtre temporelle, si toutes doivent être utilisées pour atteindre s' à partir de s, i.e., aucun d'entre eux pris individuellement ne peut atteindre l'état cible.

Si l'on prend l'exemple d'une galerie d'images, en entrée, l'utilisateur peut pointer du doigt une image et prononcer la phrase "supprimer cette image". L'interprétation de la phrase "supprimer cette image" indique la commande à exécuter, mais ne permet pas de connaître l'objet sur lequel elle doit s'appliquer. De même, le pointage par le doigt permet de connaître l'objet cible, mais pas la commande qui doit être appliquée dessus. La compréhension complète de l'énoncé nécessite donc la prise en compte des deux messages.

Assignation

On dit que la modalité m est assignée pour atteindre l'état s' à partir de l'état s, si aucune autre modalité ne peut être utilisée à cet effet. En d'autres termes, une modalité est assignée à une tâche donnée lorsque cette dernière ne peut être effectuée qu'à l'aide de cette modalité. Cette propriété exprime donc l'absence de choix.

Par exemple, en entrée, on peut restreindre la saisie d'un mot de passe uniquement au clavier.

Equivalence

Les modalités d'un ensemble M sont dites équivalentes s'il est possible d'atteindre l'état s' à partir de l'état s indifféremment au moyen de l'une ou l'autre des modalités. Formellement, deux modalités d'interaction sont dites équivalentes si chaque périphérique ou langage d'interaction permet d'atteindre le même but en produisant les mêmes données.

Par exemple, en entrée, l'utilisateur peut remplir un champ texte dans une interface en employant la commande vocale ou en écrivant le texte au clavier. En sortie, le système peut afficher une notification à l'écran ou la signaler avec un message sonore.

Redondance
Les modalités d'un ensemble M sont dites redondantes si elles ont la même puissance expressive (elles sont équivalentes) pour atteindre l'état s' à partir de l'état s, et si toutes sont utilisées dans la même fenêtre temporelle. La redondance est donc un cas particulier de l'équivalence où un même énoncé est transmis sur les modalités concernées.

Un exemple d'application est donné dans [Bouchet, 2006] où le montage redondant d'un microphone et d'une caméra qui observe le mouvement des lèvres d'un utilisateur permet d'augmenter la robustesse d'un système de reconnaissance vocale.

Parmi les outils mettant en scène les propriétés CARE, ICARE (*Interaction CARE*) [Bouchet, 2006] est une approche basée sur les composants pour la conception et le développement des interfaces utilisateurs multimodales, constituée de composants élémentaires. Un composant élémentaire supporte une modalité pure (par exemple, mode vocal seul, graphique seul). Un éditeur graphique permet aux concepteurs d'assembler graphiquement les composants selon les propriétés CARE. Cette composition est ensuite transformée automatiquement en code exécutable. Toutefois, à l'exécution, ce code est incapable de s'adapter dynamiquement au contexte d'utilisation. De plus, la multimodalité est uniquement traitée en entrée.

b. Le framework TYCOON

Le framework TYCOON a été conçu dans le but d'observer, d'évaluer et de préciser différents types de coopérations entre les modalités d'interaction.

Dans [Martin, 1999] une modalité d'interaction est définie comme un processus qui analyse et produit des "morceaux d'information". L'approche TYCOON repose sur les notions de types et d'objectifs de coopération entre les modalités. À la suite d'une étude menée dans des domaines tels que la psychologie, l'intelligence artificielle et l'interaction homme-machine, six types de coopération de base entre les modalités ont été distinguées :

(1) Transfert. Spécifie que l'information produite par une modalité est utilisée par une autre modalité. Le transfert peut se produire entre deux modalités de même nature (deux modalités en entrée ou deux modalités en sortie), ou bien entre une modalité en entrée et une modalité en sortie. Par exemple, lorsqu'une partie d'une phrase prononcée par l'utilisateur a été mal reconnue par la modalité parole, elle peut être modifiée à l'aide d'un clavier afin que l'utilisateur ne soit pas obligé de taper ou de répéter la phrase entière.

(2) Équivalence. On dit que deux modalités sont équivalentes si l'information produite par l'une des modalités peut être considérée comme une alternative par la seconde. Parmi les situations où l'équivalence est souhaitée on peut citer comme exemple un moteur de reconnaissance vocale ne fonctionnant pas correctement (par exemple dans un environnement bruyant), l'utilisateur peut sélectionner la commande avec un clic.

(3) Spécialisation. Indique qu'un type d'information spécifique est toujours traité par la même modalité. La spécialisation permet en autre :

a. d'aider l'utilisateur à interpréter les événements produits par le système ;
b. elle facilite et améliore la précision de la reconnaissance vocale car l'espace de recherche est plus petit ;

c. elle diminue la durée du processus d'intégration et de sélection des modalités.

(4) Redondance. Deux modalités sont dites redondantes lorsque le fait d'actionner l'une ou l'autre des modalités abouti au même résultat. Par exemple, la fermeture d'une boîte de dialogue peut se faire en tapant "quit" (modalité pointage tactile) ou en prononçant "quit" (modalité parole).

(5) Complémentarité. Ce type de coopération considère plusieurs modalités qui traitent chacune des "morceaux d'information" différents mais complémentaires et qui sont fusionnés par la suite.

(6) Concurrence. Contrairement au type de coopération précédent, la concurrence implique plusieurs modalités utilisées en parallèle sans être fusionnées : plusieurs tâches peuvent être exécutées en parallèle. Cela permet une interaction homme-machine plus rapide [Martin, 1999].

COMIT [Stanciulescu, 2008] est un outil basé sur le framework TYCOON qui permet aux utilisateurs d'interagir avec le système afin de construire des interfaces graphiques. COMIT est défini par un langage de commande qui est utilisé pour spécifier plusieurs types de coopération entre les modes d'interaction comme la reconnaissance vocale, le clavier et la souris.

2.4.2. IHM PLASTIQUES

2.4.2.1. DÉFINITION

Jusqu'à présent, l'innovation dans l'interaction homme-machine reposait essentiellement sur l'invention de nouvelles techniques d'interaction dont l'utilisation combinée sous la forme d'une interface d'interaction multimodale visait une communication plus efficace et naturelle entre l'utilisateur et la machine. Mais toutes ces techniques supposaient le même espace d'interaction : un ordinateur de bureau. Cependant, la diversité et le succès des ordinateurs de poche et des téléphones portables, la généralisation des capteurs et des réseaux, la multiplication des systèmes embarqués dans les objets du quotidien (automobile, télévision, etc.) ont modifié les règles du jeu. Dans cette nouvelle vision «l'espace physique comme lieu d'interaction», une interface utilisateur dite «traditionnelle» s'avère rapidement insuffisante. Pour mieux servir l'utilisateur et optimiser les capacités d'interactivité des systèmes, Thévenin et Coutaz introduisent la notion de plasticité des interfaces dans [Thevenin et Coutaz, 1999] avec la première thèse en France sur le sujet en 2001 [Thevenin, 2001]. La plasticité d'une interface indique sa capacité à s'adapter dynamiquement au contexte d'utilisation tout en respectant sa facilité d'utilisation [Bastien et Scapin, 1973] ou sa valeur [Cockton, 2005]. Le travail de Didon et al. illustre cette définition [Thevenin, 2001]. Les auteurs proposent une architecture basée sur des composants de conteneur logiciel pour concevoir des applications mobiles adaptatives composées de widgets omniprésents pouvant être dupliqués, supprimés et migrés dynamiquement sur des périphériques pendant l'exécution de l'application. Kalimucho est la plate-forme logicielle proposée pour gérer leur cycle de vie dynamique.

La notion d'IHM plastique que nous défendons découle de la problématique posée par la capacité d'un système interactif à être modifié, tout en garantissant les propriétés ergonomiques dans sa manipulation. Rendre les systèmes interactifs adaptables nécessite de préciser les objectifs de cette adaptation, les conditions et les limites dans lesquels le système devra évoluer. Ainsi, notre objectif est d'assurer l'adaptation des IHM à la plateforme d'interaction (*cf.* contexte

d'usage). Par conséquent, la plasticité garantit les propriétés ergonomiques des interfaces dans le respect de certaines contraintes liées à la plateforme d'interaction.

2.4.2.2. CADRE DE RÉFÉRENCE DE LA PLASTICITÉ DES IHM

Coutaz et Nigay proposent dans [Coutaz et Nigay, 2012] une décomposition technique simplifiée de l'espace problème de la plasticité des IHM, inspirée de la thèse HDR de G. Calvary [Calvary, 2007] et illustrée par la figure 2.3. Cette vision englobe, de façon non exhaustive, les questions en cours d'exploration dans la littérature ou à investiguer dans le domaine de la plasticité des IHM.

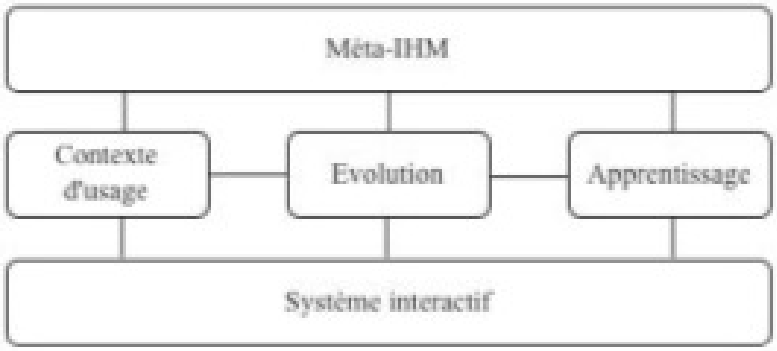

Figure 2.3. Décomposition fonctionnelle (simplifiée) d'un système interactif plastique.

Les auteurs tentent à montrer l'influence sur la plasticité du changement du contexte d'usage dans lequel le système interactif évolue ; que cette évolution peut être apprise ; et que l'ensemble du processus est contrôlé par l'utilisateur via une méta-IHM.

Contexte d'usage

Comme détaillé en section2, le contexte d'usage comprend un modèle de l'utilisateur, de l'environnement et de la plateforme d'interaction.

Evolution

La fonction d'évolution détermine la réaction de l'interface suite à un changement du contexte d'usage en faisant appel à l'un ou l'autre des mécanismes suivants :

- le *remodelage*, qui désigne le changement de forme de tout ou partie de l'IHM ;
- ou la *redistribution* qui consiste à migrer tout ou partie de l'IHM sur des ressources d'interaction gérées par des nœuds distincts de la plateforme.

Le remodelage a été le point de départ dans la recherche en plasticité. Ces premières recherches ont conduit à l'indentification des niveaux d'abstraction auxquels le remodelage peut avoir lieu. Ces niveaux d'appuient sur l'architecture générale des *Model-Based Interface Design Environment* (MB-IDE) introduite dans [Szekely, 1996]. De façon simplifiée, cette architecture distingue :

- Les tâches utilisateur et concepts du domaine qui décrivent l'interaction homme-machine d'un point de vue métier, indépendamment des modalités d'interaction ;
- L'interface abstraite qui structure l'IHM en espaces de travail. Un espace de travaille est un « lieu d'activité virtuel offrant les éléments nécessaires à la réalisation d'une ou plusieurs tâches » [Normand, 1992] ;
- L'interface concrète où se fait le choix des modalités d'interaction ;

- L'interface finale fait le choix d'un langage de programmation et d'environnement d'exécution.

Ces niveaux d'abstraction sont généraux. Aussi, ont-ils été repris en 1999 par Thévenin et Coutaz (1999) qui proposent un cadre conceptuel, établissant des lignes directrices pour promouvoir le développement d'interfaces utilisateurs plastiques. Basée sur une approche IDM, la spécification de l'interface utilisateur est définie par un ensemble de modèles combinant des descriptions de haut niveau et déclaratives des capacités d'interaction de l'interface utilisateur et de l'environnement physique dans lequel elle sera exécutée. Ces modèles sont ensuite soumis à une série de transformations à travers des outils automatiques ou semi-automatiques jusqu'à atteindre une implémentation complète. Le travail de Thévenin et Coutaz a conduit à la création du cadre de référence unifié [Calvary et al., 2003] [Thévenin, 2001] appelé Cameleon. Cameleon explique et formalise différents aspects qui contribuent à la production d'interfaces utilisateur plastiques en termes de modèles et de relations entre modèles, dans un esprit IDM. En plus du modèle de contexte d'utilisation et du modèle d'adaptation recommandés par Cameleon, les quatre niveaux de Cameleon illustrés à la figure 2.4 comprennent les modèles suivants: T&C (modèles de tâches et concepts), AUI (modèle d'interface utilisateur abstrait) (Modèle d'interface utilisateur concret) et FUI (modèle d'interface utilisateur final).

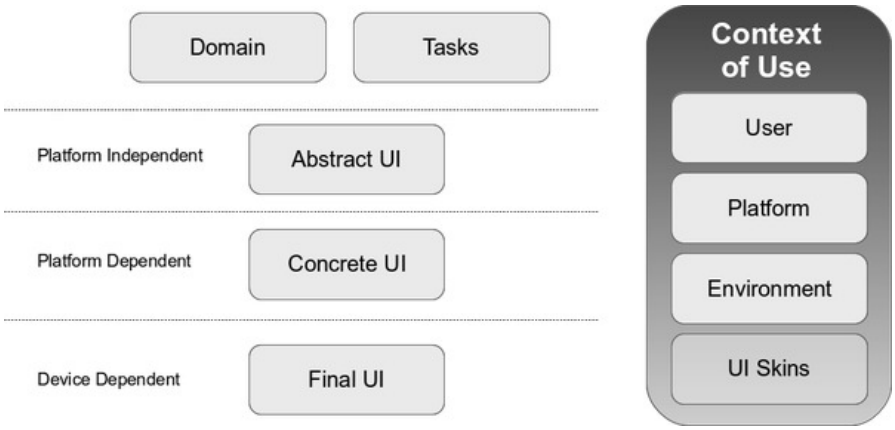

Figure 2.4. The Cameleon Reference Framework[16].

Bien que Cameleon soit un cadre théorique, il sert de base à de nombreuses approches dans le domaine des interfaces utilisateurs plastiques [Balme et al., 2004]. Parmi les préoccupations propres à la plasticité que couvre Cameleon [Calvary et al., 2003], nous distinguons :

- L'identification du contexte d'usage et du changement de contexte que le système interactif peut subir ;
- La préservation de l'utilisabilité du système interactif lors de l'adaptation à appliquer ;
- L'infrastructure d'exécution pour la mise en œuvre de l'adaptation. La mise en œuvre s'appuie sur une connaissance du système interactif dans son état, des mécanismes pour la détection du changement de contexte d'usage et la mise en œuvre de l'adaptation. On parle de remodelage intra-modal lorsque, dans le couple <d, r>, la modalité d est préservée mais le langage d'interaction r change. On parle de remodelage inter-modal lorsqu'il y a changement des deux unités du couple <d, r> faisant intervenir une modalité simple. Le remodelage est multimodal dés lors qu'il y a changement dans les

[16] https://www.w3.org/community/uad/wiki/Cameleon_Reference_Framework

combinaisons de modalités selon les propriétés CARE. C'est ainsi que la multimodalité intervient dans le remodelage des IHM plastiques.

Apprentissage

Encore peu exploré, l'apprentissage en plasticité désigne l'ajustement approprié des règles d'évolution, par exemple préférer les remodelages aux redistributions, selon le degré de contrôle de l'IHM le plus approprié à l'utilisateur. Les travaux d'A. Hariri entrent dans ce cadre [Hariri, 2006].

Méta-IHM

La notion de méta-IHM a été introduite dans [Coutaz, 2006] puis l'appellation a été revue dans [Sottet et al, 2007] pour éviter toute confusion avec le terme « méta » en IDM. « méta » faisant référence à la relation « est conforme à » en IDM, ce qui n'est pas approprié dans ce contexte d'IHM de l'adaptation. On lui préfèrera le terme « extra » faisant référence à la notion d'ajout ou d'IHM supplémentaire [Coutaz et Nigay, 2012].

Cette méta/extra-IHM aura pour mission de recouvrir l'ensemble des fonctions nécessaire et suffisant à l'utilisateur pour le contrôle de son espace interactif. A ce titre, elle doit permettre de répondre aux questions de Bellotti [Belloti et al., 2002] : « *Comment m'adresser au système ? Le système sait-il que je m'adresse à lui ? Est-il prêt à interpréter mes actions ? Exécute-t-il les actions attendues ? Comment éviter les erreurs ? Etc.* ».

2.5. SYNTHÈSE ET DISCUSSION

Nous avons consacré ce chapitre à l'étude de la conception des interfaces homme-machine. Dans le cadre de nos travaux, nous avons abordé la question du point de vue de deux propriétés liées au domaine des IHM : la multimodalité et la plasticité. Nous avons vu aussi comment la multimodalité participe à la plasticité des IHM.

Dès lors, le développement d'une interface homme-machine ne correspond plus à une simple activité de programmation. Ainsi, nous passons de la programmation des IHM à l'ingénierie des IHM impliquant la prise en compte de multiples points de vue. En vue de la complexité de la conception des IHM nouvelle génération, celle-ci ne pourra être maîtrisée que par la mise en commun des savoirs, non seulement au sein de la discipline IHM, mais aussi avec d'autres domaines comme par exemple l'Ingénierie Dirigée par les Modèles.

Comme étudié dans le chapitre précédent, l'Ingénierie Dirigée par les Modèles est une approche visant à modéliser les différents aspects d'un système. Dans cette thèse, nous utilisons l'ingénierie dirigée par les modèles afin de capitaliser les savoirs et savoir-faire en IHM. Les approches orientées modèles font partie du paysage de l'IHM depuis un certain temps déjà. Après une première génération d'approches basée sur des modèles qui ciblent des domaines d'application particuliers tels que les bases de données, un cadre de travail plus général est apparu, prenant en compte des problématiques liées à l'adaptabilité, la prise en compte du contexte d'utilisation, etc. Ce sont particulièrement cette dernière génération d'approches qui nous intéresse. Ainsi, le chapitre suivant est consacré a à un état de l'art des approches orientées modèles pour la prise en compte de la plasticité et de la multimodalité dans le domaine des IHM.

CHAPITRE 3
APPROCHES À BASE DE MODÈLES POUR LA CONCEPTION DES IHM

" Bien que l'indépendance des périphériques facilite la portabilité, l'interactivité et la convivialité sont améliorées par la dépendance des périphériques "
R. Baecker

3.1. INTRODUCTION

La conception d'interfaces basées sur des modèles est une pratique de longue date qui tire parti des pratiques de l'IDM (Pérez-Medina et al., 2007). D'après (Coutaz, 2010), dans les années 80, les grammaires (appelées aujourd'hui métamodèles) étaient la base formelle pour la génération d'interfaces textuelles ou graphiques. Pour un développement des IHM utilisant une approche IDM, les métamodèles à utiliser doivent présenter un haut niveau d'abstraction afin de promouvoir leur réutilisabilité et leur portabilité et prendre en compte l'hétérogénéité induite par les différents dispositifs d'interaction présents sur le marché.

Ce chapitre est entièrement dédié à l'étude de différents aspects de l'usage des modèles pour la conception des IHM, avec un focus sur les approches à base de modèles dédiées à la multimodalité et à la plasticité des IHM. Deux dimensions qui nous intéressent particulièrement dans le cadre de nos travaux.

Dans ce qui suit, vous trouverez en section 2 une classification des modèles spécifiques à l'IHM. La section 3 présente une vision synthétique des travaux de référence basés sur l'ingénierie des modèles dans le domaine des IHM où nos travaux sont positionnés, à cela nous ajoutons la définition d'une liste de critères d'évaluation basés sur les recommandations de la littérature. Nous discutons de l'intérêt d'une approche IDM pour le développement d'interfaces graphiques en section 4. La section 5 conclue ce chapitre.

3.2. USAGE DES MODÈLES EN IHM

Les modèles généraux impliqués dans le développement des systèmes interactifs tournent autour de plusieurs types de modèles, plus ou moins liés, et spécifiques à plusieurs niveaux d'abstraction et étapes du processus de développement. Dans (Basnyat et al., 2005), les auteurs classent les modèles de l'IHM dans onze catégories, communément acceptées, mais dont les limites sont parfois floues. Cela conduit à une carte des modèles spécifique à l'IHM (Tableau 3.1), assez vague car les objectifs de certains modèles peuvent s'entrecroiser.

Type de modèle	Description
Modèle d'analyse des besoins	Analyse de l'activité de l'utilisateur, exigences fonctionnelles et non fonctionnelles du système, caractéristiques du domaine.
Modèle de tâches	Description des différentes activités menant à la finalité du système. Utilisé par exemple pour la planification et pendant les différentes phases du développement de l'interface utilisateur.
Modèle utilisateur	Caractérisation des profils d'utilisateurs, définition de propriétés génériques (valables pour un large échantillon d'utilisateurs) et représentation d'informations sur la perception, la cognition ou l'interaction.
Modèle de contexte	Les aspects environnementaux, y compris l'emplacement, la température, la durée, les aspects sociaux et l'éthique culturelle, susceptibles d'affecter l'interaction entre les utilisateurs et les systèmes.
Modèle de plateformes	Du point de vue IDM, description des plates-formes logicielles en termes d'environnements d'exécution et de protocoles de communication. Du point de vue IHM, description des capacités matérielles des périphériques.
Modèle du système	Modélisation des aspects statiques et fonctionnels d'un système incluant l'état du système, les actions du système, la concurrence, la gestion des périphériques d'entrée, le rendu, les techniques d'interaction...
Modèle de présentation	Détaille l'organisation spatiale et hiérarchique des composants d'interface. Les interactions post-WIMP nécessitant d'autres types de modèles tels que les modèles d'interaction.
Modèle du domaine	Identification des concepts manipulés par un système et les dépendances existantes entre eux.
Modèle de dialogue	Définit les caractéristiques et l'organisation des échanges entre utilisateurs et systèmes.
Modèle de dispositifs	Description des caractéristiques fonctionnelles des dispositifs.
Modèle d'architecture	Modularisation du système par l'utilisation de patrons architecturaux (Seeheim, Arch, MVC, PAC, etc.)

Tableau 3.1. Modèles généraux pour l'IHM (Basnyat et al., 2005).

Les modèles d'analyse des besoins n'existent pas en tant que tels. D'autres types de modèles sont utilisés pour recueillir les résultats d'analyse tels que les modèles d'utilisateurs, les modèles de contexte, les modèles de tâches et les modèles d'interaction. Si les métamodèles de tâche et d'interaction sont disponibles, il est plus difficile d'en définir pour les modèles d'utilisateurs et de contexte. Les modèles d'architecture et de dialogue bénéficient des travaux du domaine de l'ingénierie logicielle et plusieurs métamodèles sont disponibles. Aussi, il est difficile de trouver une utilisation appropriée pour les modèles des dispositifs tant les modèles d'architecture et des plateformes peuvent fournir ce type de connaissances. Encore faut-il pouvoir définir clairement ce que seraient les modèles de plateforme, ainsi que les modèles de système, ces concepts étant insuffisamment précis et utilisés à des fins très différentes selon les besoins (Gauffre, 2003).

Enfin, pour chaque type de modèle, les caractéristiques des domaines d'application et des paradigmes d'interaction (web, systèmes critiques, multimodalité, interfaces tactiles, etc.)

introduisent une grande source de variabilité entre les modèles (Gauffre, 2003). L'utilisation productive de ces modèles est difficile car les passerelles entre chacun restent peu claires. Cependant, plusieurs types de modèles, pour lesquels des métamodèles émergent, sont liés et utilisés comme ressources premières du développement, y compris des modèles de tâches, d'interaction et d'architecture. C'est la tendance que suivent nombreux des travaux de la littérature dont nous détaillons quelques uns dans la section qui suit.

3.3. IDM-IHM : LES TENDANCES DE LA LITTÉRATURE

À ce jour, de nombreux travaux ont adopté une approche UIDL pour la conception d'interfaces utilisateur. Pour l'essentiel, la recherche se concentre sur la portabilité, la multimodalité, l'indépendance des périphériques dans le développement des interfaces utilisateur pour n'en nommer que quelques-uns. La diversité des dispositifs d'interaction existant aujourd'hui (PC, smartphone, tablettes ...) fait de la portabilité et de l'adaptabilité des interfaces d'interaction un thème de recherche récurrent dans la communauté IHM. Compte tenu des différences entre ces dispositifs d'interaction en termes de taille d'écran, de ressources d'interaction ..., il est difficile d'assurer une expérience utilisateur optimale pour chaque configuration. Nous abordons ici ce problème en combinant l'utilisation de la plasticité et de la multimodalité dans une approche MDA. L'objectif est de faire de la multimodalité un acteur du remodelage de l'interface utilisateur visant à offrir une expérience utilisateur satisfaisante, quelle que soit la configuration de l'appareil pour les paramètres: zone d'affichage et modalités d'interaction.

Dans cette section, nous proposons un bref historique des approches et des frameworks qui résume l'utilisation de la multimodalité et de la plasticité dans l'ingénierie des IHM, et dont les contributions représentent un tournant dans le développement d'interfaces utilisateur de prochaine génération. Ensuite, nous présentons les critères que nous avons utilisés pour évaluer notre approche. Ces critères sont construits en prenant en compte: la multimodalité, la plasticité, le niveau d'abstraction, l'hétérogénéité et l'approche de développement.

3.3.1. APPROCHES DE RÉFÉRENCE

La description des interfaces d'interaction est un point de recherche particulièrement dynamique et est souvent basée sur des langages dérivés du langage XML comme UsiXML (*USer Interface eXtensable Markup Language*) [Vanderdonckt et al., 2004]. UsiXML est un UIDL pour la modélisation d'applications web multimodales adaptées aux mobiles en entrée et en sortie. Il est structuré selon les quatre niveaux de Cameleon. La modélisation de l'interaction multimodale se produit au niveau CUI. Le métamodèle concret intègre les modalités de l'interaction tactile, de la voix et du graphisme. Les propriétés CARE de combinaison entre modalités sont également traitées par UsiXML. UsiXML dispose d'une large gamme d'outils, tels que IdealXML et GrafiXML [Michotte et Vanderdonckt, 2008] par exemple. Ils permettent la création de modèles de tâches, abstraits et concrets, le mapping entre les modèles, la simulation d'arbres de tâches et la génération de modèles de dialogue. Le code source final de l'interface est obtenu par génération après de nombreuses transformations basées sur ces modèles. Le principal problème d'UsiXML est que la génération de code est uniquement orientée web, lorsque notre langage HCIDL permet la génération d'interfaces d'interaction natives. En outre, lorsque UsiXML ne prend pas en charge les interactions basées sur des capteurs mobiles, HCIDL fournit un niveau d'abstraction adéquat permettant la gestion des capteurs mobiles.

DynaMo-AID (*Dynamic Model-Based User Interface Development*) [Clerckx et al., 2005] est un framework de développement pour les interfaces utilisateur contextuelles. Il permet la modélisation d'interfaces multimodales en entrée et en sortie et leur coopération en fonction des propriétés CARE. Différents modèles sont utilisés pour spécifier des interfaces avec une modélisation explicite des modalités d'interaction. Le modèle de tâche est créé par l'ingénieur ainsi que les modèles de contexte et de présentation, tandis que le modèle de dialogue est généré automatiquement par l'outil. Les modèles sont sérialisés via un langage XML appelé "DynaMOL". Ces modèles sont utilisés au moment de l'exécution pour générer le code de l'interface dans : J2ME pour les interfaces mobiles, Java Swing pour les applications de bureau et HTML/CSS pour les applications Web. Le problème avec cette approche réside dans la multitude et la diversité des modèles qui peuvent rendre la modélisation difficile pour un débutant. Comme mentionné dans le chapitre suivant, notre approche permet au concepteur de rassembler le code d'interface dans un seul fichier ou de le séparer en plusieurs, à travers une seule extension de fichier et un mécanisme d'inclusion de fichiers. De même que pour UsiXML, les interactions basées sur les capteurs mobiles ne sont pas prises en compte.

SMUIML (*Synchronized Multimodal User Interaction Modeling language*) [Dumas, 2010] est un dialecte XML pour la modélisation des interactions multimodales en entrée. Il a été créé pour configurer la plateforme HephaisTK. SMUIML modélise des interactions multimodales à l'aide d'un éditeur graphique dédié ainsi que leur coopération suivant les propriétés CARE. Après la modélisation graphique, SMUIML génère un script de configuration utilisé avec un autre fichier Java pour configurer HephaisTK associé à l'application cible. La modélisation du dialogue multimodal est représentée par une machine d'état dans l'éditeur graphique [Dumas et al., 2014]. L'éditeur propose en outre une édition double synchronisée sous forme graphique et textuelle ainsi qu'un certain nombre d'opérateurs pour la combinaison temporelle des modalités. Le principal défaut de cette approche est que la multimodalité de sortie n'est pas traitée par SMUIML. De plus, SMUIML/HephaisTK n'a pas été créé pour permettre la modélisation des interactions avec les appareils mobiles. Néanmoins, le langage fournit un niveau d'abstraction approprié pour les interactions de capteurs mobiles basées sur un modèle (les concepts *Recognizer* et *Trigger* dans le métamodèle). En comparaison, HCIDL permet de prendre en charge à la fois les interactions multimodales mobiles en entrée et en sortie.

Parmi les approches basées sur SMUIML, MIMIC [Elouali et al., 2014] est une approche basée sur l'IDM pour le développement et la génération d'applications mobiles multimodales en entrée et en sortie, basées sur SMUIML. Le langage proposé M4L s'appuie sur les concepts de SMUIML pour la description des événements en entrée et en sortie. Ainsi, chaque événement d'entrée/sortie est décrit par : la modalité d'interaction déclenchée ; la coopération (ou non) avec d'autres événements du même type sur la base des propriétés CARE et TYCOON ; et le traitement déclenché par l'événement. Dans cette approche, la coopération n'est pas considérée entre les modalités mais entre les événements qui utilisent ces modalités pour plus de précision lors de la modélisation des interactions. Pour la conception de modèles, MIMIC utilise le méta-éditeur ModX qui, à partir d'un métamodèle donné, génère l'éditeur graphique correspondant. Le générateur de code est implémenté comme un code JavaScript dans ce méta-éditeur. Cette approche permet la génération de code pour les plates-formes Android, iPhone et les navigateurs mobiles (HTML5/CSS3). Comme SMUIML, M4L ne modélise pas les positions exactes des widgets sur les écrans d'application et ne peut donc pas générer d'interfaces utilisateur esthétiques.

Hasselt [Cuena et al., 2014] [Cuena et al., 2016] est un langage textuel, déclaratif, axé sur les événements pour la description des modèles d'interaction multimodaux exécutables. Le concept de base de Hasselt est un événement composite, qui est essentiellement une séquence d'événements définie par l'utilisateur et qui est logiquement liée. Avec Hasselt, les développeurs définissent des événements composites en reliant plusieurs événements primitifs (par exemple des événements tactiles ou des entrées vocales) au moyen d'opérateurs spécialisés. Chaque opérateur représente une relation spécifique entre leurs opérandes. L'événement composite global peut alors être lié à un ou plusieurs gestionnaires d'événements, qui spécifient le comportement que le système doit exposer lorsque l'événement composite se produit. Lors de l'exécution, les gestionnaires d'événements sont exécutés chaque fois que leurs événements composites associés se produisent. Pour la détection d'événements, Hasselt s'appuie sur des dispositifs de reconnaissance existants pour traiter les entrées de bas niveau (comme la parole ou les mouvements de souris) et ne remplace pas les moteurs de fusion existants basés sur la reconnaissance. Hasselt fait partie d'une suite de système de gestion de l'interface utilisateur, appelée Hasselt UIMS [Cuena et al., 2015]. Il comprend un éditeur de code, un environnement d'exécution et des outils de débogage pour l'écriture, l'exécution et l'évaluation des programmes Hasselt.

L'objet de ces travaux se limite à la conception d'interactions multimodales. En plus de la multimodalité, notre travail couvre un autre angle. Nous nous concentrons sur la manière de fournir une disposition optimale pour les interfaces d'interaction multimodales. Nous proposons donc deux méthodes de positionnement que nous détaillons dans le chapitre suivant qui est dédié à notre approche.

RBUIS (*Role-Based User Interface Simplification*) [Akiki et al., 2016] est une technique d'adaptation de l'interface utilisateur qui améliore la convivialité grâce à un comportement adaptatif en fournissant aux utilisateurs finaux un ensemble de fonctionnalités minimales et une disposition optimale, en fonction du contexte d'utilisation. Les auteurs définissent un ensemble de fonctionnalités minimal comme l'ensemble minimal requis pour effectuer une tâche, et une disposition optimale comme celle qui optimise l'expérience de l'utilisateur en adaptant les propriétés des widgets de l'interface utilisateur. RBUIS utilise une approche basée sur un modèle d'exécution interprété basé sur l'architecture Cedar et est prise en charge par l'IDE Cedar Studio. Ainsi, les interfaces utilisateur simplifiées avec RBUIS montrent une amélioration significative de la facilité d'utilisation par rapport à leurs contreparties initiales. Plus proche de nos travaux, cette approche favorise le comportement adaptatif des interfaces d'interaction en fournissant l'ensemble de fonctionnalités minimal requis pour une tâche et une mise en page optimale. Dans sa définition, HCIDL favorise le comportement adaptatif des interfaces d'interaction à travers la gestion de la multimodalité afin de tirer parti de leur potentiel interactif, tandis que l'utilisateur évolue dans une mise en page optimale.

La recherche en ingénierie basée sur les modèles a entraîné l'émergence de nombreuses enquêtes et études comparatives. Dans sa thèse, Bouchet [Bouchet, 2006] fait un recensement et une classification des approches connues, selon différents critères. Ces critères incluent le cycle de développement, les utilisateurs ciblés (concepteurs, informaticiens, programmeurs, etc.), le pouvoir d'expression et la nature des combinaisons possibles. Dans leur article sur les interfaces multimodales, Dumas et al. [Dumas et al., 2009] ont présenté un aperçu des principes, des modèles et du cadre dans ce domaine. Leur étude couvre des sujets tels que la fusion de types de données hétérogènes, les architectures pour le traitement en temps réel, la gestion de dialogue,

les langages de modélisation, l'apprentissage automatique pour l'interaction multimodale et les frameworks. De ce point de vue, Dumas et al. [Dumas et al., 2010] ont publié une étude sur l'état de l'art de la modélisation des langages d'interaction multimodale et en ont énuméré les avantages. Cette étude met en évidence un ensemble de lignes directrices pour la conception de langages dédiés à la description de l'interaction multimodale, ainsi que les rôles que ce langage doit remplir : la communication, la configuration, l'enseignement et la modélisation. Enfin, les auteurs mettent en avant l'équilibre entre les caractéristiques particulières de lisibilité et d'expressivité que ces langages doivent établir.

Nous avons résumé les différentes étapes d'évolution de la plasticité et de la multimodalité afin de s'imposer dans le domaine de l'interaction homme-machine et d'arriver à la définition que nous en connaissons aujourd'hui. Depuis lors, de nombreuses méthodes et frameworks ont enrichi le domaine des interfaces utilisateurs nouvelle génération et contribuent à sa maturité.

3.3.2. CRITÈRES D'ÉVALUATION

Afin de procéder à une évaluation objective de nos travaux par rapport aux travaux existants, nous avons établi [Gaouar et al., 2018] la liste suivante de critères basés sur les recommandations de la littérature [Nigay et Coutaz, 1993]. Nous pouvons classer ces travaux selon cinq caractéristiques que nous considérons nécessaires de respecter, afin d'assurer une modélisation englobant toutes les spécificités du développement des IHM (Tableau 3.2) :

(1) **Approche de développement basée sur les modèles :** l'IDM a servi de base à de nombreuses recherches sur le développement de l'interface utilisateur. Cette approche présente des avantages tels que l'indépendance technologique et la possibilité de prendre en charge la vérification des propriétés des modèles ainsi que la traçabilité.
(2) **Prise en compte de l'hétérogénéité des plateformes/dispositifs :** nécessaire compte tenu de l'évolution rapide des différentes plateformes et de l'émergence de nouveaux dispositifs d'interaction (PC, smartphones, tablettes ...).
(3) **Support de la multimodalité :** prise en compte des interactions multimodales en entrée et/ou sortie. L'approche devrait également intégrer les combinaisons de modalité (propriétés CARE: Complémentarité (C) Affectation (A), redondance (R) Équivalence (E)).
(4) **Support des capteurs mobiles :** le niveau d'abstraction doit être suffisamment élevé pour masquer la complexité de l'implémentation tout en permettant de définir des interactions détaillées (pour la modalité de pointage tactile, elle doit permettre de différencier un toucher et un contact long, etc.). Cela donnera plus de clarté dans la modélisation.
(5) **Optimisation du layout :** optimiser le layout de l'interface utilisateur en adaptant les propriétés des widgets de l'interface utilisateur et leur positionnement aux contraintes imposées par les dispositifs d'interaction. Pour résoudre ces problèmes, il est nécessaire de modéliser les styles graphiques avec le niveau d'abstraction approprié.

RBUIS	Hasselt/ Hasselt UIMS	M4L/ MIMIC	SMUIML/ HephaisTK	DynaMOL/ Dynamo-AID	UsiXML	
Modélisation et génération de code	Modélisation et interprétation à l'exécution	Modélisation et génération de code	Modélisation et interprétation seulement	Modélisation et génération de code	Modélisation et génération de code (pour le Webuniquement)	**Approche de développement basée sur les modèles**
Multi-plateformes et multi-périphériques	Multi-plateformes et multi-périphériques	Multi-plateformes et multi-périphériques	Pas d'hétérogénéité	Hétérogénéité partielle	Pas d'hétérogénéité	**Prise en compte de l'hétérogénéité**
None	Modalités entrée/ortie par gestionnaires d'événements	Modalités entrée/sortie utilisant les propriétés CARE	Modalités en entrée utilisant les propriétés CARE	Modalités entrée/sortie utilisant les propriétés CARE	Modalités entrée/sortie utilisant les propriétés CARE	**Support de la multimodalité**
Intégration d'interactions basées sur des capteurs	Intégration d'interactions basées sur des capteurs	Intégration d'interactions basées sur des capteurs	aucune modalité d'interaction basée sur les capteurs fournie	aucune modalité d'interaction basée sur les capteurs fournie	aucune modalité d'interaction basée sur les capteurs fournie	**Support des capteurs mobiles**
Interfaces utilisateur adaptatives	None	None	None	Application sensible au contexte	None	**Optimisation du layout**

Tableau 3.2. Analyse des approches de référence selon les cinq caractéristiques.

Le tableau 3.2 résume l'analyse de ces approches selon les cinq critères que nous avons définis précédemment. Nous pouvons voir qu'aucune des approches n'implémente tous les critères. Ce tableau montre que la littérature actuelle apporte des réponses concrètes dans la gestion des interactions multimodales en utilisant des approches basées sur des modèles. Cependant, ces solutions ne sont pas toutes multi-plateformes/multi-dispositifs. En outre, nous notons une faiblesse de ces initiatives dans le support de modalités d'interaction à base de capteurs. Mais l'observation la plus pertinente que nous pouvons noter est que lorsque les approches se concentrent sur la conception d'interactions multimodales, d'autres se concentrent sur la conception d'applications plastiques. Mais aucun ne couvre les deux angles. Cependant, nous sommes convaincus que l'adaptabilité et la facilité d'utilisation des interfaces d'interaction impliquent de tirer parti de leur potentiel interactif et de fournir un layout optimal. C'est ce que nous proposons avec notre approche.

Dans cette thèse, nous complétons cette gamme de contributions en présentant un langage de description d'interface utilisateur nommé HCIDL. Dans notre approche basée sur le MDA, nous combinons multimodalité et plasticité dans un processus de remodelage. Nous favorisons le comportement adaptatif des interfaces d'interaction à travers la gestion de la multimodalité afin de tirer parti de leur potentiel interactif, tandis que l'utilisateur évolue dans un layout optimal :

- optimisation du layout : pour cela, nous utilisons deux méthodes de positionnement. Une approche "linéaire" pour les interfaces utilisateur simples où les composants s'alignent les uns après les autres. Une seconde approche "relative", pour des interfaces plus complexes, où la position d'un composant est exprimée par rapport à la position d'un autre composant ;
- la valorisation du potentiel interactif: nous utilisons la multimodalité à ce niveau afin de préserver le confort d'interaction entre l'utilisateur et la machine. La multimodalité associée aux propriétés CARE permet d'anticiper la différence en termes de ressources d'interaction entre les différents dispositifs.

3.4. L'INTÉRÊT D'UNE APPROCHE IDM POUR LE DÉVELOPPEMENT D'INTERFACES GRAPHIQUES (IHM)

En 1992, une étude sur la programmation des IHM (Myers, 1992) à montré que 48% du code des 74 logiciels étudiés, 45% de leur temps de conception, 50% de leur temps de développement et 37% de leur temps de maintenance sont consacrés aux IHM. La programmation des IHM n'est pas aisée [Myers et al., 2000]. Cela reste toujours valable aujourd'hui [Meixner et al., 2011]. En plus du fait que les besoins des systèmes interactifs ont considérablement augmenté pendant ces dernières années et des nombreuses modalités d'interaction, l'hétérogénéité dans l'environnement des systèmes interactifs (PC, smartphone, tablette…), et leurs différences en termes de dimension et de modalités d'interaction, ne facilite en rien le développement des IHM.

Les modèles sont depuis longtemps utilisés pour faciliter et réduire le temps de développement des interfaces. Au début, ils ont été utilisés pour faciliter l'implémentation avec les systèmes de gestion d'interfaces. Le but était de séparer la partie fonctionnelle de l'interface graphique dans les applications (ex. modèle Seeheim). Ensuite, avec l'apparition de l'IDM, on est passé de la

programmation à l'ingénierie des IHM. Les modèles sont devenus plus productifs, réduisant ainsi le temps de développement. Beaucoup des premières approches de génération de code (automatique et semi-automatique) à partir de modèles ont échoués [Myers et al., 2000]. Cependant, des progrès significatifs ont été réalisés au fil des années pour la génération totale ou partielle des interfaces.

L'utilisation d'une approche à base d'IDM pour la modélisation et la génération d'interfaces a de nombreux intérêts :

- Elle réduit significativement le temps de développement grâce à la génération automatique de code.
- Elle permet aussi d'élever le niveau d'abstraction. Même si cela permet de réduire les efforts de programmation en évitant ou en diminuant la programmation à bas niveau, on ajoute toujours l'effort de modélisation.
- Les modèles issus d'une approche IDM peuvent être utilisés par des concepteurs d'interfaces qui ne sont pas programmeurs.
- L'utilisation de modèles accélère le développement d'applications dans le contexte des plateformes (Android, iOS, Windows Phone, etc.) /dispositifs (ordinateur de bureau, smartphone, tablette, PDA, etc.) hétérogènes.

Il est légitime de se poser des questions telles que « pourquoi ne pas utiliser des APIs de haut niveau pour implémenter les interfaces au lieu de suivre une approche IDM ? ». Et aussi, « pourquoi ne pas juste copier/coller le code qui se répète au lieu de le générer automatiquement ? ». Il nous semble que la réponse s'impose d'elle-même si nous prenons un moment pour analyser la situation.

Premièrement, il y a une grande différence entre générer et copier/coller. Copier/coller est une activité manuelle, génère des erreurs et nécessite d'adapter le code à chaque fois, alors que la génération de code est automatique.

Deuxièmement, même si les APIs représentent une bonne alternative, elles sont coûteuses à développer et utiliser. Il faut développer des APIs pour les différentes plateformes et les développeurs doivent être familiers avec elles ce qui nécessite beaucoup d'efforts et de compétences en programmation. Alors que dans une approche à base de modèles, il suffit de créer un seul modèle pour générer du code sous toute les plateformes (les modèles étant indépendants des plateformes) sans aucun effort de développement (ou beaucoup moins). En outre, l'utilisation des modèles résiste à l'évolution technologique puisqu'ils sont abstraits et simplifie la compréhension de l'application, même après un certain temps.

3.5. DISCUSSION ET SYNTHÈSE

L'alliance IDM-IHM semble se renforcer au fil des nombreux travaux de recherche entamés dans cette voie. Plusieurs perspectives s'ouvrent dorénavant au couple IDM-IHM, notamment avec l'objectif de contribuer à la gestion du processus de développement. Ainsi, la revue des usages de l'IDM, en particulier en IHM, met en avant plusieurs avantages qui peuvent bénéficier au développement des systèmes interactifs. En effet, les bénéfices élémentaires d'une démarche IDM, exposées tout au long du chapitre précédent, favorisent la diffusion et l'accès aux modèles. L'IDM offre aussi le cadre à la définition d'un processus de développement avec ses différentes étapes, ponctuées de modèles, et ses transitions, ponctuées de transformations.

Pour un développement des IHM utilisant une approche IDM, les métamodèles à utiliser doivent présenter un haut niveau d'abstraction afin de promouvoir leur réutilisabilité et leur portabilité et prendre en compte l'hétérogénéité induite par les différents dispositifs d'interaction présents sur le marché.

Nos travaux s'inscrivent dans le domaine des langages de description d'interfaces utilisateurs. Nos recherches se concrétisent par l'introduction de notre langage HCIDL (*Human-Computer Interface Description Languages*), un langage de modélisation mis en scène dans une approche MDA. Parmi les propriétés liées à l'interface homme-machine, notre proposition est destinée à la modélisation d'interfaces d'interaction plastiques, multi-cibles, multimodales, utilisant des langages de description d'interface utilisateur. En combinant la plasticité et la multimodalité, HCIDL améliore la convivialité des interfaces utilisateur grâce à un comportement adaptatif en fournissant aux utilisateurs finaux un ensemble d'interaction adapté aux entrées/sorties des terminaux et une disposition de l'interface graphique optimale. Les chapitres suivants sont entièrement dédiés à notre travail et à mettre en avant notre approche et notre langage HCIDL.

PARTIE III
IMPLANTATION ET VALIDATION

CHAPITRE 4
HCIDL: UN LANGAGE DE DESCRIPTION POUR LES IHM MULTI-CIBLES, MULTIMODALES, PLASTIQUE

"La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et que personne ne sait pourquoi. … "
Albert Einstein

4.1. INTRODUCTION

L'interaction avec un système informatique a considérablement évolué au cours des dernières décennies. Les percées technologiques ont profondément modifié la forme des systèmes informatiques modernes. L'ordinateur de bureau n'est plus le seul représentant des systèmes informatiques. En effet, la miniaturisation des dispositifs, l'expansion des réseaux de communication, et la performance croissante des unités de calcul représentent les changements majeurs apportés en notre aire. Les utilisateurs ont aujourd'hui un large éventail de produits à leur disposition : les téléphones mobiles sont devenus de véritables ordinateurs; qui ont succédé aux tablettes, aux montres connectées, à la smart TV, etc. dont les capacités augmentent chaque jour un peu plus et qui s'enrichissent de nouvelles fonctionnalités. Pour que ces systèmes restent utilisables malgré leur complexité, il est nécessaire d'adopter de nouveaux moyens d'interaction qui vont au-delà de la triade souris-clavier-écran et qui se distinguent du paradigme de l'interaction WIMP (*Windows, Icons, Menus and Pointing device*). Ainsi, une représentation plus instinctive de l'information est requise, qu'il s'agisse en entrée ou en sortie des systèmes. Cette observation a conduit à repenser fondamentalement les modes d'interaction entre l'humain et la machine. L'objectif principal est de promouvoir, autant que possible, une communication naturelle avec l'utilisateur final.

La combinaison de plusieurs modalités en entrée et/ou sortie permet d'améliorer à la fois la robustesse et la fiabilité de l'interaction. Le travail présenté par Richard Bolt (1980) soutient la conception et le développement d'applications multimodales, où l'ordinateur est enrichi de nouveaux modes d'interaction pour les soutenir. De plus, l'émergence de l'informatique mobile et des différents types de capteurs équipant les appareils mobiles (accéléromètres, capteurs gravitationnels, gyroscopes ...) ont permis l'émergence de nouvelles modalités [Bellik, 1995] telles que l'inclinaison du téléphone ou le changement d'orientation. La multimodalité réduit considérablement les contraintes d'interaction des plates-formes, telles que les petits écrans ou les claviers inconfortables, et limite les erreurs de reconnaissance.

Cependant, l'innovation dans l'interaction homme-machine ne peut pas se limiter à l'invention de nouvelles techniques d'interaction dont l'utilisation est combinée sous la forme d'interfaces multimodales. En raison du contexte technologique actuel combinant la diversité des dispositifs

d'interaction et une redéfinition de l'espace d'interaction, la conception et le développement d'interfaces d'applications impliquent désormais de nouvelles exigences. Il ne suffit pas de considérer l'IHM conventionnelle, dédié à un dispositif cible dans un emplacement cible pour un type déterminé d'utilisateur final. Il est juste de passer de ces IHM invariables et unimodales à l'IHM multi-cible, multimodale et plastique.

Notre travail nous a amené à approfondir cette perspective. De nos investigations, nous présentons HCIDL, notre langage de description d'interface utilisateur. L'objectif est d'améliorer la convivialité des interfaces utilisateurs via un comportement adaptatif grâce à un ensemble d'interactions adaptées à l'entrée/sortie des terminaux et à une disposition optimale. Nous proposons d'utiliser à la fois la plasticité et la multimodalité comme support pour le remodelage de l'IHM, lorsque les travaux de la littérature ne se concentrent que sur l'une ou l'autre de ces caractéristiques à la fois.

HCIDL est structuré en trois paquets selon le modèle MVC [Krasner and Pope, 1988] et s'inspire des langages de modélisation existants tels que SMUIML [Dumas et al. 2010] et M4L [Elouali et al., 2014]. Il permet de soutenir à la fois les interactions multimodales mobiles en entrée et en sortie. Sur la base des propriétés CARE [Coutaz et al., 1994] et TYCOON [Martin, 1999], nous considérons quatre coopérations entre les modalités : équivalence, concurrence, redondance et complémentarité. Le niveau d'abstraction offert par HCIDL permet la gestion des capteurs existants sur les appareils mobiles. Pour définir une disposition optimale des interfaces d'application, nous développons deux méthodes de positionnement. Une approche "linéaire" pour les interfaces utilisateur simples où les composants s'alignent les uns après les autres. Une seconde approche "relative", pour des interfaces plus complexes, où la position d'un composant est exprimée par rapport à la position d'un autre composant.

Pour illustrer notre langage nous mettons en scène un exemple d'interface tout au long de ce chapitre. Le modèle de l'application est formulé avec HCIDL puis le code source de l'interface est généré automatiquement conformément à une approche de type MDA. Pour mettre en œuvre des règles de transformation pour la phase de génération, nous utilisons le framework Acceleo.

Le reste de ce chapitre est organisé comme suit : la section 2 traite brièvement du patron de programmation MVC. En section 3 nous discutons des éléments de gestion de l'interface homme-machine. Nous consacrons la section 4 à la présentation de notre langage HCIDL, détaillant sa syntaxe abstraite et concrète ainsi que l'étape de génération de code. Nous concluons ce chapitre par une synthèse et discussion en section 5.

4.2. MVC : UN PATRON DE PROGRAMMATION ADAPTÉ

Comme énoncé en introduction, l'une des particularités de notre langage de modélisation est sa structure en trois paquets selon le modèle MVC. Avant de continuer, il est important de comprendre notre motivation à l'utilisation de ce patron. Il est à noter que de nombreuses plateformes mobiles encouragent l'utilisation d'une architecture MVC pour structurer ses applications [Krasner and Pope, 1988]. Une architecture MVC permet de diviser le code source d'une application en trois couches distinctes :

- Modèle : dans une application, cette couche consiste souvent à récupérer les données et de les traiter en fonction de l'affichage souhaité par les utilisateurs.

- Vue : dans une application, les vues permettent d'afficher les données récupérées à partir du modèle à l'utilisateur. Elles permettent aussi de capter les interactions utilisateurs sur les éléments graphiques de l'interface (clic, swipe ...). Après une interaction utilisateur, les vues transmettent les évènements détectés aux contrôleurs.
- Contrôleur : les contrôleurs s'occupent de faire le lien entre les vues et le modèle. Ils récupèrent les évènements utilisateurs et en fonction de l'état dans lequel se trouve l'application, effectuent les actions souhaitées (changement de vue ...). Si ces actions nécessitent un changement de la couche modèle, le contrôleur s'occupe de demander ces changements. Ensuite, il notifie la vue concernée pour qu'elle mette à jour son état et affiche les nouvelles données.

C'est sur ce même principe que nous avons construit l'architecture de notre langage HCIDL. La particularité de notre travail est que le principe du patron est appliqué à une autre échelle, à savoir au niveau de la construction de l'interface graphique en elle-même. L'utilisateur interagit directement avec le package de présentation, qui est connecté au package de contrôle d'interaction qui détermine les modalités d'interaction et les services d'application devant être sollicités. Le package modèle permet d'accéder aux données de la partie fonctionnelle.

Notre choix s'est porté sur cette décomposition afin de renforcer le caractère modulable de notre approche. Nous justifierons ce choix plus en détail à travers les sections suivantes. Ainsi, notre vision de l'interaction se présente comme suit :

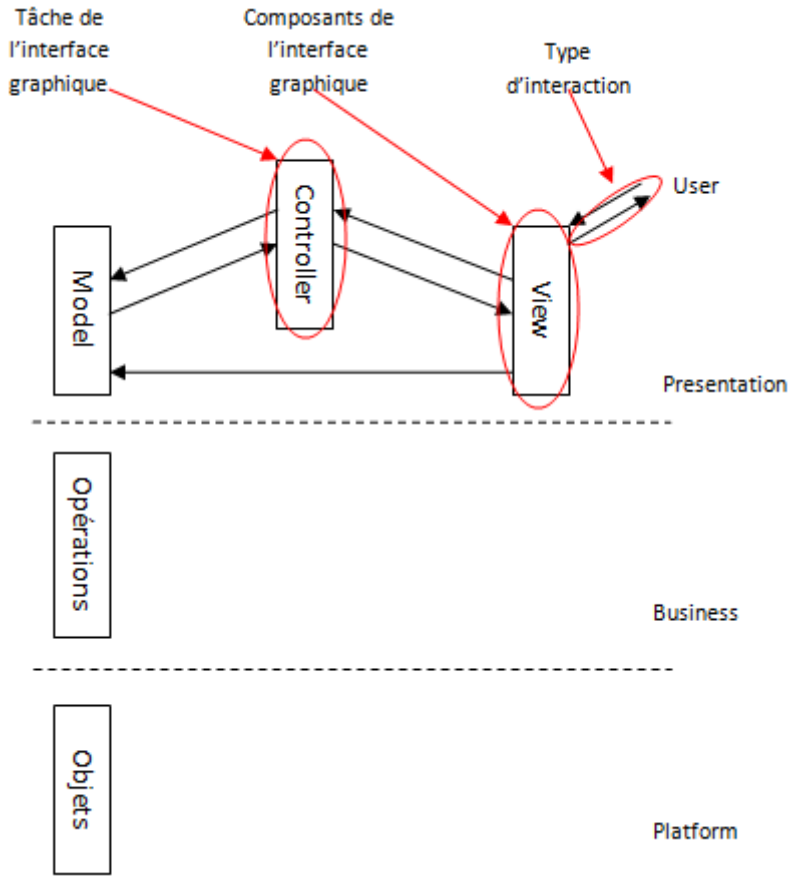

Figure 4.1. Représentation de l'interaction selon une décomposition en modèle MVC.

Une application se décompose en trois couches distinctes. Ainsi nous avons la couche « Platform » composée des objets liées à la plateforme d'interaction. La couche « Business » responsable des opérations sur les données et enfin la couche « Presentation » qui n'est autre que l'interface permettant à l'utilisateur d'interagir avec son application.

C'est cette couche de présentation qui nous intéresse dans nos travaux. Ainsi, nous proposons de décomposer cette couche en trois sous-couches selon le modèle MVC. La partie « View » représente les composants de l'interface graphique, ce que l'on voit. C'est avec cette sous-couche que l'utilisateur interagit directement. La partie « Controller » est responsable de la gestion des évènements d'interaction, c'est-à-dire toutes les tâches que doit exécuter l'application au déclenchement d'un évènement d'interaction. La partie « Model » tient pour rôle d'interface de communication avec la partie noyau fonctionnel.

4.3. ELÉMENTS DE GESTION DE L'INTERACTION HOMME-MACHINE

La communication entre l'homme et la machine se fait par le biais de périphériques d'entrée et de sortie qui composent l'interface graphique. Ces périphériques diffèrent selon le dispositif dont il est question (smartphone, tablette, ordinateur). Lors du développement d'une application, la construction de son interface devra être en adéquation avec le modèle d'interaction du dispositif cible.

En ce qui concerne les types d'interaction, celles-ci se divisent en plusieurs catégories. Elles représentent le panel de possibilités pour l'interaction entre l'homme et la machine. A ce jour, elles ont toutes été implémentées grâces à la prolifération des capteurs mobiles. Ainsi, nous avons : l'interaction vocale, l'interaction tactile, l'interaction sonore, l'interaction oculaire, l'interaction gestuelle, l'interaction physique (souris et clavier).

Une interaction ne pourrait se faire sans composants d'interface graphique. Ils représentent une porte de communication vers l'application. Nous pouvons citer : les icônes, les menus (contextuels, déroulants, hiérarchique, détachable), les boutons d'action, les boutons radio, les cases à cocher, les listes (Liste déroulante, Liste de choix hiérarchique), les classeurs à onglets, les zones de saisie, les labels, les molettes d'incrément, les curseurs sur une règle ou encore la barre de progression.

Enfin, parmi les tâches de l'interaction graphique, nous pouvons citer : tâche de saisie (texte, quantité, position, tracé), de sélection (élément dans un ensemble (liste, bouton radio), de plusieurs éléments (case à cocher)), tâche de déclenchement (bouton, menu, glisser-déposer), tâche défilement (barre de défilement, défilement automatique), de transformation (redimensionnement d'élément graphique, zoom).

4.4. HCIDL: HUMAN-COMPUTER INTERFACE DESCRIPTION LANGUAGE

Nous présentons dans cette section les concepts de modélisation qui constituent la syntaxe abstraite de HCIDL (le métamodèle) et sa syntaxe concrète textuelle associée, ainsi que les règles clés pour les transformations de modèles qui nous permettent d'obtenir le code Android correspondant à partir du modèle de l'interface.

Nous nous sommes tournés vers l'environnement de développement Eclipse pour la mise en œuvre de notre approche. Nous nous appuyons sur l'Eclipse Modeling Framework (EMF) pour construire le métamodèle. Pour mettre en œuvre notre langage de modélisation textuel, nous utilisons le puissant framework Xtext qui, à partir d'un métamodèle, génère un éditeur de code correspondant incluant des fonctionnalités utiles comme la coloration syntaxique, l'auto-complétion, etc.

4.4.1. HCIDL : SYNTAXE ABSTRAITE – LE MÉTAMODÈLE

Les modèles d'architecture définissent l'organisation logicielle du système interactif. Ils séparent la partie fonctionnelle (qui implémente les concepts propres au domaine d'application) de l'interface utilisateur afin d'avoir une meilleure modularité. Nous proposons dans nos travaux d'appliquer le principe de modularité au développement de la couche d'interface d'une application. Dans l'esprit de l'architecture MVC, notre langage sépare les préoccupations de développement d'une interface d'application en plusieurs packages comme le montre la Figure 4.2. Il se compose d'un package de présentation (IhmView), d'un package de contrôle d'interaction (Controller) et d'un package d'interface pour accéder aux données de la partie fonctionnelle (Model). L'utilisateur interagit directement avec le package de présentation. Ce package est connecté au contrôleur d'interaction qui détermine les modalités d'interaction et les services d'application à solliciter. Dans la suite, nous détaillerons chaque package de notre métamodèle.

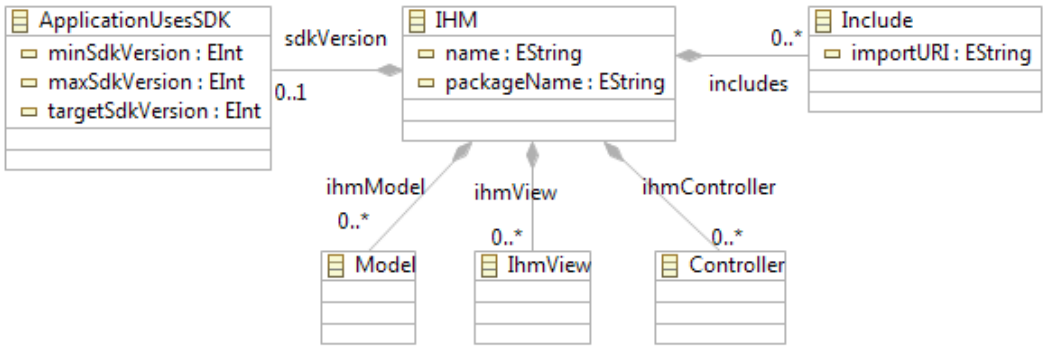

Figure 4.2. Vue globale du métamodèle HCIDL.

4.4.1.1. PACKAGE VIEW

Comme le montre la figure 4.3, une interface se compose d'un screen (métaclasse `Screen`), d'une barre de navigation (métaclasse `Menu`), de layouts (Metaclass `Layout`) et d'un conteneur pour les composants graphiques (métaclasse `GUIElement`). Le type de structure choisi est spécifié dans le modèle d'interface par le méta-attribut booléen `isRelative` de la classe `Layout`. Cette fonctionnalité de notre métamodèle permet la conception d'interfaces dynamiques pouvant s'adapter à n'importe quelle taille d'écran. Chaque écran peut éventuellement être composé soit d'un layout (qui peut lui-même contenir d'autres layouts ou composants d'interface), soit de composants d'interface (métaclasse `ViewCollection`). Ce la vient du fait que la métaclasse `ViewCollection` est définie comme une collection de vues (métaclasse `View`). Ces vues peuvent être soit des `Layout`, soit des `GUIElement`. Les éléments de l'interface graphique (par exemple les widgets) sont les composants qui peuvent être trouvés dans une interface (bouton, champ de texte ...). La métaclasse `StringVA` dans le schéma ci-dessous provient du package `Model`. C'est le lien entre les packages `Model` et `View`.

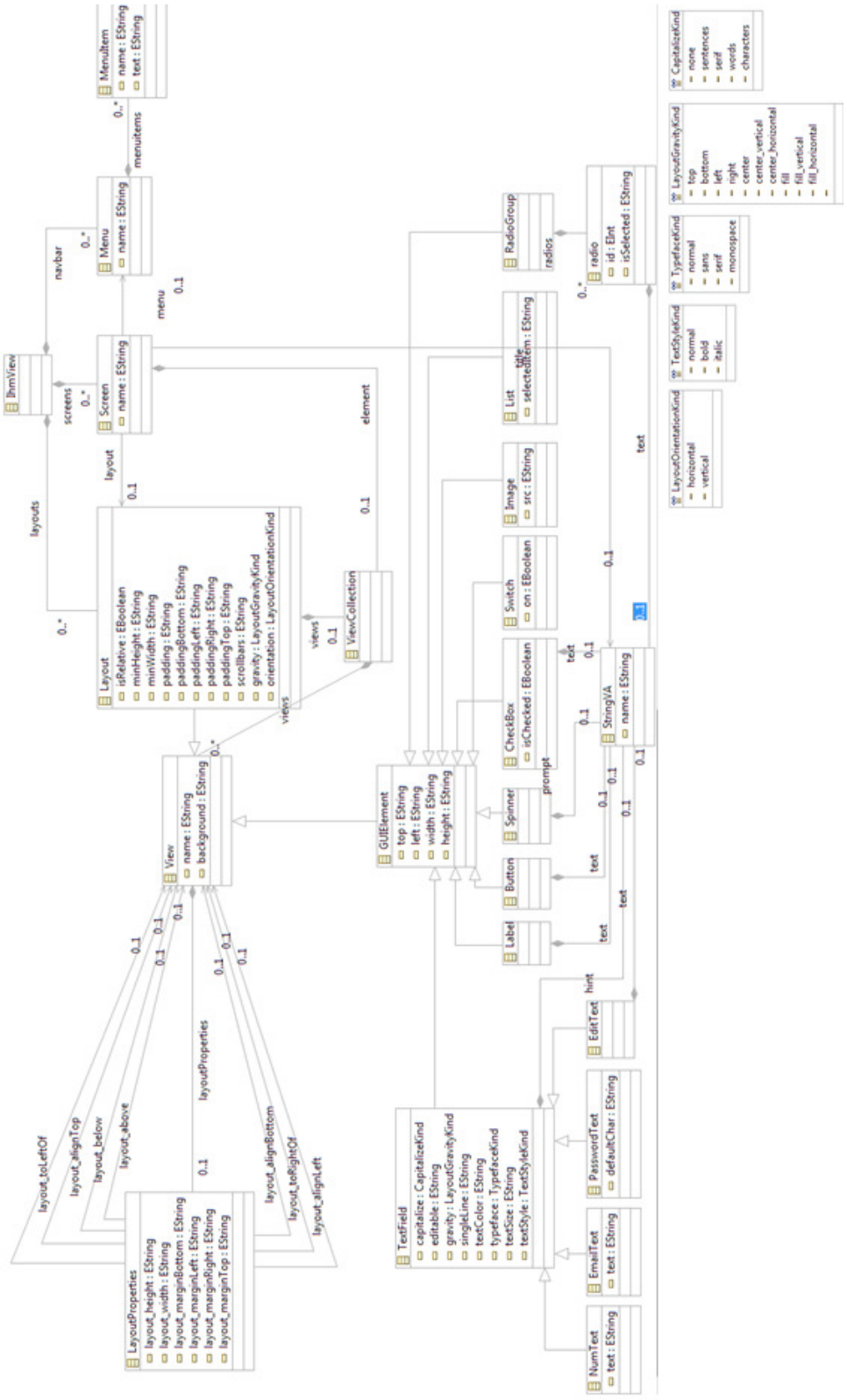

Figure 4.3. Package View du métamodèle HCIDL.

Une particularité de notre approche est la gestion de la structure visuelle et du positionnement des composants d'interface. C'est le rôle de cette classe. Le développeur peut choisir entre deux types d'agencements en fonction de la complexité des interfaces :

- Linéaire (`Linear`) : adapté aux interfaces graphiques simples. Les composants graphiques sont simplement disposés les uns après les autres de manière linéaire.
- Relatif (`Relative`) : la position dans l'interface de chaque composant graphique est spécifiée par rapport à la position d'un autre composant en utilisant les propriétés fournies par la classe `LayoutProperties`. Nous les illustrons par un exemple de fonctionnement simple présenté dans la figure 4.4 pour faciliter la compréhension. Dans cet exemple, le bouton Ok est placé sous le champ de texte (`below = editURL`) et son bord droit est aligné avec le bord droit du champ de texte (`alignRight = editURL`). De plus, le bouton Annuler est placé à gauche du bouton Ok (`toLeftOf = Ok`) et son bord supérieur est aligné sur celui de son voisin (`alignTop = Ok`).

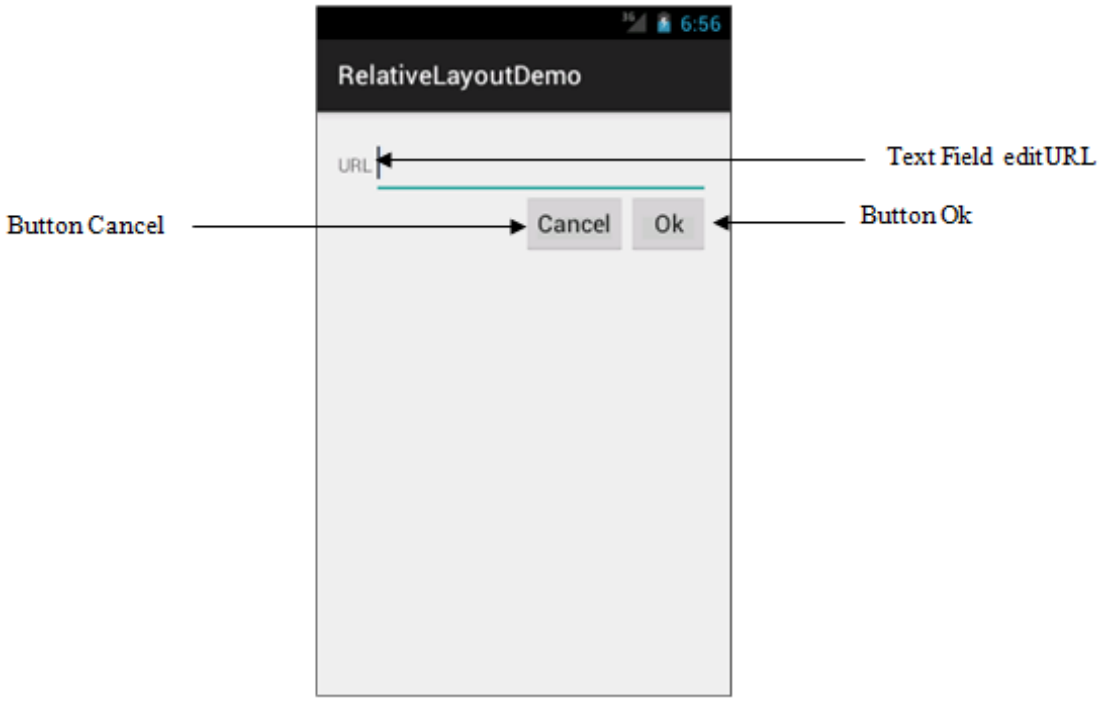

Figure 4.4. Rendu d'une interface d'application construite avec la propriété `Relative`.

4.4.1.2. PACKAGE CONTROLLER

Le package `Controller` (voir Figure 4.5) est la partie de notre métamodèle responsable de la gestion de l'interaction et de la multimodalité. À cette fin, nous avons été inspirés par les ouvrages de la littérature tels que SMUIML et M4L.

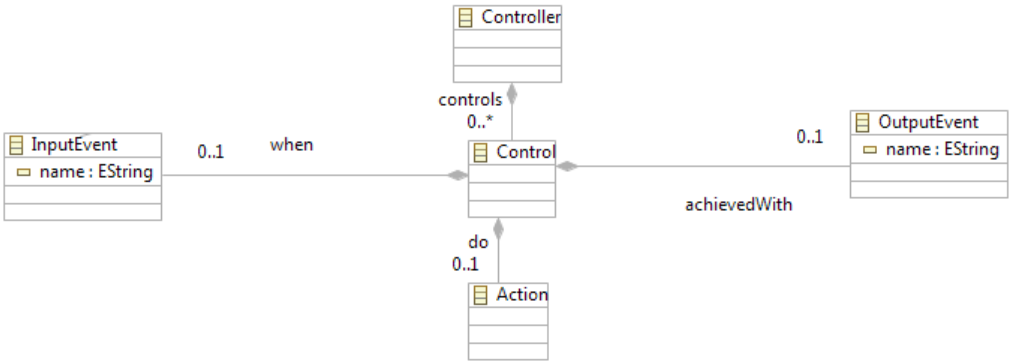

Figure 4.5. Vue réduite du package Controller du métamodèle HCIDL.

Le package `Controller` décrit les événements et les actions liées à l'interaction homme-machine. Une interaction est représentée par la métaclasse `Control` à travers trois points constituant la définition de l'interaction.

La figure 4.5.a illustre l'événement en entrée représenté par la métaclasse `InputEvent`. De plus, l'interaction en entrée est modélisée comme un événement émis par l'utilisateur (en cliquant sur un bouton, en faisant défiler une liste ou en sélectionnant un menu) ou par le système (événements internes, réception d'un SMS ou batterie faible). La classe `GUIElement` dans ce diagramme provient du package `View`. Il représente le lien existant entre les deux packages `Controller` et `View`.

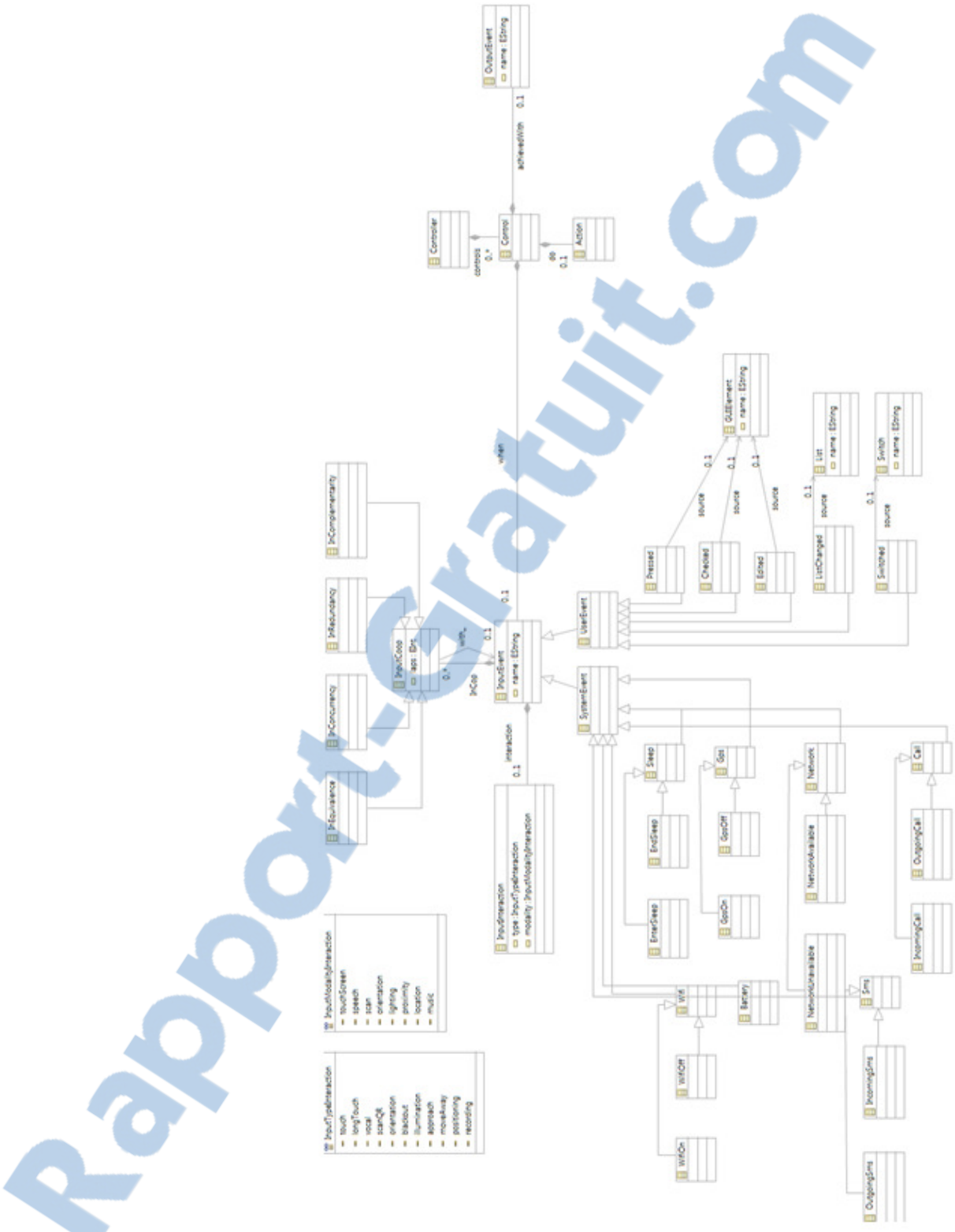

Figure 4.5.a. Package Controller : vue sur la métaclasse InputEvent.

La figure 4.5.b illustre le traitement déclenché par la métaclasse `Action`. Plus précisément, chaque événement en entrée peut avoir des effets sur le système tels que le passage à une autre application ou l'ouverture d'une fenêtre.

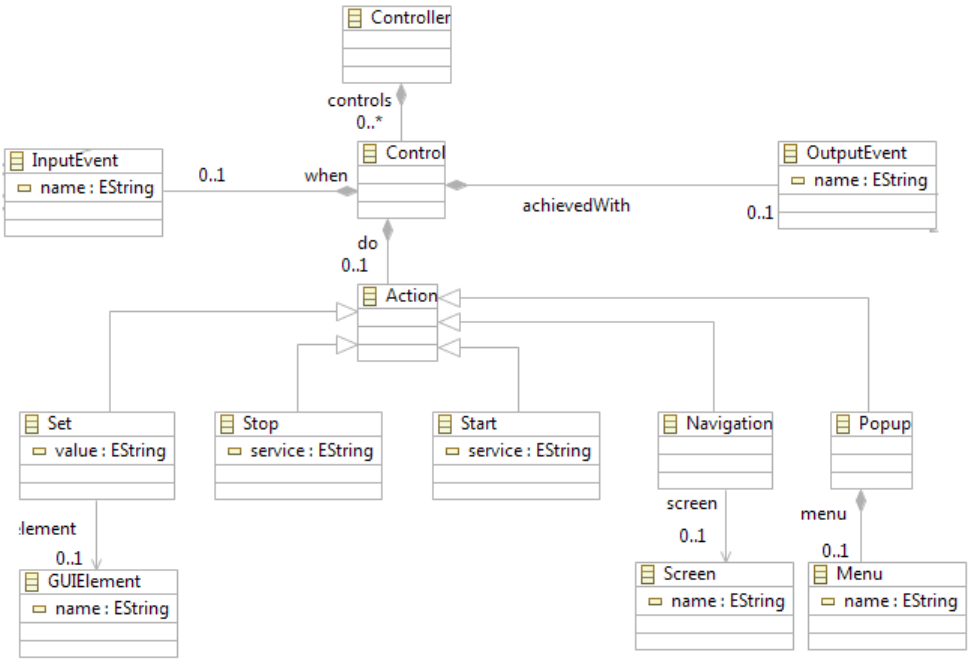

Figure 4.5.b. Package Controller : vue sur la métaclasse Action.

La figure 4.5.c illustre l'événement en sortie représenté par la métaclasse `OutputEvent`. En outre, l'interaction en sortie correspond à la réponse du système et se présente sous la forme de vibration, de synthèse vocale ou d'affichage de notification.

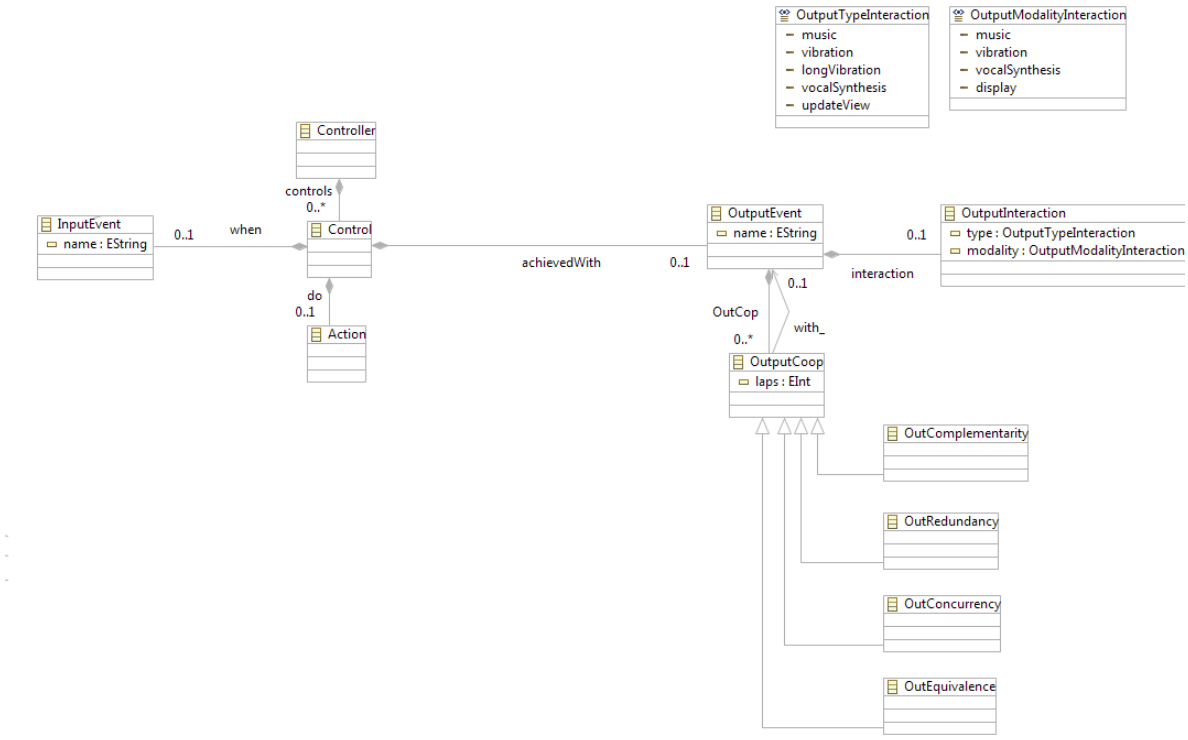

Figure 4.5.c. Package Controller : vue sur la métaclasse OutputEvent.

Pour chaque événement en entrée et en sortie :

- La métaclasse `InputInteraction`/`OutputInteraction` permet de spécifier le type (méta-attribut `type`) et la modalité d'interaction (méta-attribut `modality`). Cette distinction entre type et modalité d'interaction est faite car plusieurs types d'interaction peuvent utiliser la même modalité. Par exemple, les interactions à long toucher et toucher simple utilisent toutes les deux la modalité tactile. Cette caractéristique de notre métamodèle permet une description très précise de l'interaction qui minimise les erreurs de reconnaissance.
- Chaque événement coopère ou non avec d'autres événements du même type (métaclasse `InputCoop` pour les événements en entrée et `OutputCoop` pour les événements en sortie). Basé sur les propriétés CARE et TYCOON, nous considérons quatre coopérations : l'équivalence, la simultanéité, la redondance et la complémentarité. Pour chaque événement, le développeur spécifie la nature de la coopération, le nom de l'événement coopérant et l'intervalle de temps maximal dans lequel doivent se produire les deux événements pour la combinaison à considérer.

4.4.1.3. PACKAGE MODEL

Le package `Model` décrit la ressource (les données) qui peut être liée à la définition de l'interface. Comme le montre la figure 4.6, les ressources (métaclasse `Resource`) sont définies comme les principaux types de données. Par exemple, la métaclasse `StringResource` représente un type de ressource texte. La métaclasse `ValueAccess` détermine le type d'accès aux données : soit par des valeurs littérales (les métaclasses se terminant par "VA"); soit par des ressources locales ou externes déclarées (classe héritant de la métaclasse `ResourceAccess`), autorisant la déclaration de variables dans le modèle d'interface.

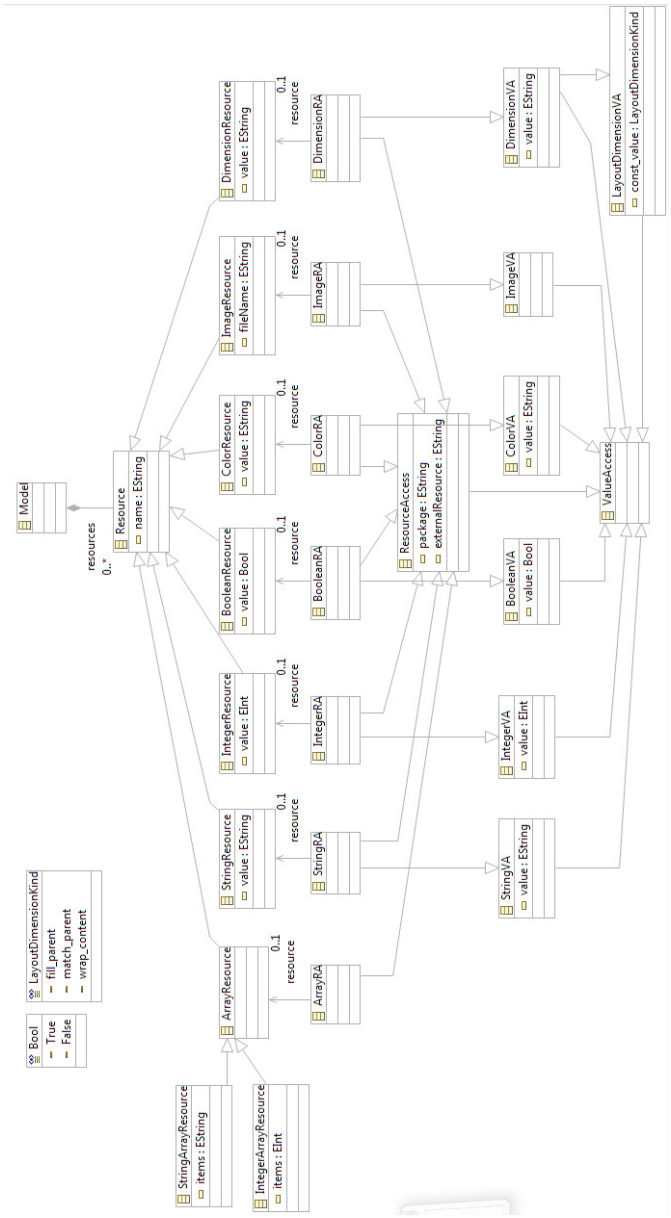

Figure 4.6. Package Model du métamodèle HCIDL.

Selon la configuration contenu-conteneur sur laquelle HCIDL est basé, les valeurs du type prédéfini `LayoutDimensionKind` (voir l'exemple présenté dans la Figure 4.7) signifient :

- `fill-parent` : le composant occupe toute la place dans le conteneur parent.
- `match_parent` : maximise la taille du composant dans son conteneur parent.
- `wrap_content`: indique au gestionnaire d'allouer la taille minimale afin que le composant soit rendu correctement.

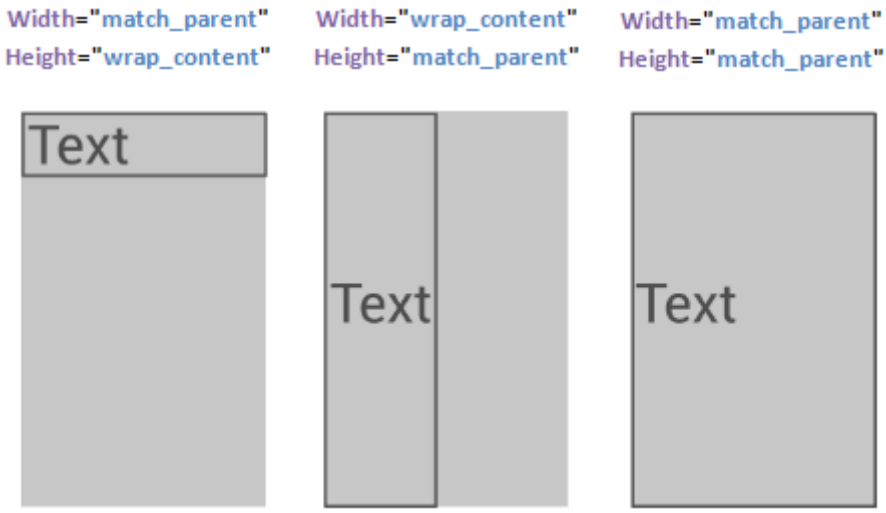

Figure 4.7. Exemple de configuration avec rendu.

4.4.2. HCIDL : SYNTAXE CONCRÈTE

HCIDL est un langage de modélisation textuel. La description d'une interface d'application nécessite la modélisation de :

- les éléments d'interface utilisateur graphique et leurs dispositions visuelles,
- les interactions proposées pour permettre la communication entre l'utilisateur et l'application, et
- les données utilisées dans la construction de l'interface. HCIDL rassemble tous ces points dans un langage de haut niveau, faisant abstraction des différences existantes entre les plateformes d'interaction.

Dans notre description d'une interface d'application, nous considérons que l'interface est un ensemble d'écrans, chacun décrivant un état de l'application à un instant donné.

Prenons l'exemple de l'application `RelativeLayoutDemo` représentée à la figure 4.8. Elle permet d'entrer une URL et d'y accéder. D'un point de vue graphique, cette application consiste en un label (lignes 7-13) suivi d'un champ de texte pour entrer une URL (lignes 14-22). Suivi par deux boutons, Ok (lignes 23-30) et Annuler (lignes 31-39). L'écran principal nommé LoadURL renvoie à une mise en page relative (ligne 4). Les attributs `width` et `height` déterminent la taille des éléments, comme la valeur `fill_parent` indique que l'élément occupera tout l'espace disponible dans le conteneur parent et la valeur `wrap_content` qu'il adaptera sa taille à son contenu. Ces propriétés ont également des valeurs fixes telles que : 5px (pixels), pour spécifier la taille des composants.

```
1   application "RelativeLayoutDemo"
2
3   screen LoadURL {
4       show layout_LoadURL
5   }
6   Relative layout layout_LoadURL {
7       # label URL {
8           layout:{
```

```
9                       width: wrap_content
10                      height: wrap_content
11          }
12          text: "URL:"
13      }
14          # textField editURL {
15              layout:{
16                      width: fill_parent
17                      height: wrap_content
18              toRightOf: URL
19              alignTop: URL
20              marginBottom: 5px
21          }
22      }
23          # button Ok {
24              text: "Ok"
25              layout:{
26                      width: wrap_content
27                      height: wrap_content
28                      alignRight: editURL
29              }
30          }
31          # button Cancel {
32              text: "Cancel"
33              layout:{
34                      width: wrap_content
35                      height: wrap_content
36                      toLeftOf: Ok
37                      alignTop: Ok
38              }
39          }
40  }
```


Figure 4.8. RelativeLayoutDemo.ihm

Dans le contexte de la gestion des interactions, la première interaction en entrée est représentée dans la figure 4.9 et implémente le clic sur le bouton Ok (ligne 4). Après avoir entré l'URL de la page Web, l'utilisateur peut s'y rendre de deux manières: i) interaction tactile: en cliquant sur le bouton OK et en invoquant la modalité `Touchscreen`, et ii) interaction vocale: ou en disant le mot "OK" "à travers la modalité `Speech`. Par la relation d'équivalence (ligne 6) existant entre les deux modalités d'interaction. De même, la seconde interaction en sortie valide cette entrée en utilisant la modalité d'affichage (lignes 9-12) pour visualiser la page Web (ligne 8).

```
1   %include <RelativeLayoutDemo.ihm>
2   when begin
3       USER clickOk
4       pressed[Ok]
5       has (touch, touchScreen)
6       equivalence=[(vocal, speech), 2]
7       End
```

```
8    do goto "http://www.univ-tlemcen.dz"
9    with begin
10         OUTPUT DisplayResult
11         has (updateView, display)
12        End
```

Figure 4.9. interaction.ihm

Les inclusions nous permettent d'inclure les fichiers `Model` contenant certaines instructions. Dans l'exemple précédent (voir figure 4.9), il indique l'inclusion (ligne 1) du fichier contenant la déclaration des éléments graphiques de l'interface. L'un des points forts de notre langage est de permettre au concepteur de rassembler le code d'interface dans un seul fichier ou de le séparer en plusieurs. Ce caractère de modularisation est rendu possible précisément en gardant la même extension pour chaque fichier, mais cela implique de gérer correctement les références et la portée des fichiers. Cette modularisation du code d'interface permet d'alléger le contenu de chaque fichier et favorise leur réutilisation.

4.4.3. HCIDL : LES RÈGLES DE GÉNÉRATION

Dans la continuité de notre exemple, nous présentons ici les règles clés pour les transformations de modèles qui nous permettent d'obtenir le code Android correspondant à partir du modèle de l'interface.

Comme énoncé en introduction, nous avons fait le choix du framework Acceleo pour définir les règles de transformation. Le déploiement de l'application générée est effectué à l'aide d'Eclipse. Le langage utilisé par Acceleo est une implémentation de la norme MOFM2T. Ce langage de génération de code utilise une approche par Template. Dans ce qui suit, nous présentons en premier lieu la transformation de modèles par Template avant d'entamer le processus de génération.

4.4.3.1. ELABORATION DE TEMPLATES POUR LA TRANSFORMATION DE MODÈLES

Comme expliqué en détail dans le chapitre 1, il existe plusieurs approches de transformations. Dans le cadre de notre exemple, nous avons fait le choix d'une approche par template. Les outils qui supportent cette approche proposent soit des langages graphiques spécifiques de métamodèles cibles, soit des langages textuels indépendants des métamodèles [Blanc, 2005] :

- Les langages graphiques spécifiques d'un métamodèle sont souvent faciles à utiliser mais très peu expressifs, étant donné leur adhérence à un unique métamodèle. Ils ne permettent la transformation que d'un seul type de modèle.
- Les langages textuels indépendants des métamodèles sont à l'inverse très expressifs mais plus difficiles à utiliser, étant donné leur caractère générique. Ils permettent la transformation de tous les types de modèles.

Nous avons déjà précisé que l'approche par template consistait à définir les canevas des modèles cibles souhaités en y déclarant des paramètres. Ces paramètres seront substitués par les informations contenues dans les modèles sources. La figure 4.10 présente un exemple [Blanc, 2005] de template pour les tables des schémas de base de données. Ce template contient plusieurs paramètres, dont les valeurs seront obtenues à partir des informations contenues dans le modèle source. Le template présenté ici contient deux paramètres, l'un pour le nom de la table et l'autre pour toutes les colonnes de la table. Ces paramètres seront respectivement remplacés

par le nom de la classe du modèle source et par toutes ses propriétés. Dans cet exemple, un pseudo langage dédié aux schémas de base de données est utilisé. Ce langage est fondé sur XMI et sur Java et utilise une approche semblable à JSP (*JavaServer Pages*), qui consiste à entourer les mots-clés Java par les caractères <% et %> afin de les distinguer des balises XMI.

```
[1] <tables name="<%clazz.getName()%>">
[2]   <%for (Iterator it=clazz.getProps().iterator() ; it.hasNext() ; ) {
[3]     Property next = (Property) it.next(); %>
[4]    <colonnes name="<%next.getName()%>">
[5]     <type name="<%next.getType().getName()%>">
[6]   </colonnes>
[7]   <%}%>
[8] </tables>
```

Figure 4.10. Exemple de template pour les tables de schéma de base de données.

Cet exemple présente un modèle template permettant la génération d'une table de base de données à partir d'une classe Java. Nous considérons que le modèle source qui permettra de fournir les valeurs aux paramètres du template contient une classe. Cette classe est identifiée par le terme *clazz* dans le template. La première ligne contient l'élément XMI représentant une table. Cette ligne contient en outre un paramètre qui sera remplacé par la valeur du nom de la classe contenue dans le modèle source (*clazz.getName()*). Les deuxième et troisième lignes du template font appel à Java pour itérer sur toutes les propriétés de la classe contenue dans le modèle source. La quatrième ligne contient l'élément XMI représentant une colonne. Cette ligne contient aussi un paramètre qui sera remplacé par la valeur du nom de la propriété de la classe contenue dans le modèle source. La cinquième ligne contient l'élément XMI représentant le type de la colonne. Cette ligne comporte un paramètre qui sera remplacé par la valeur du nom du type de la propriété du modèle source.

4.4.3.2. PROCESSUS DE GÉNÉRATION

Le développeur commence par créer un projet Android vide sous Eclipse, puis donne son adresse au générateur afin qu'il puisse générer les fichiers java et xml de l'application, en suivant les modèles définis dans la sous section précédente. Notre générateur se compose d'un ensemble de modules Templates. Chaque module est composé de plusieurs Template dont le rôle est de générer un fragment du code de l'application Android. Les templates sont invoqués au moment opportun pour assurer un assemblage cohérent du code final.

Le processus de génération commence par l'exécution du module main.mtl qui est présenté dans la Figure 4.11. L'élément racine de notre langage HCIDL est la classe IHM, et par conséquent le modèle principal doit générer tout le code d'application à partir de cet objet. C'est à partir de ces modules que sont invoqués les templates du générateur pour la création de l'application Android.

```
1  [template public main(aIHM : IHM)]
2  [comment @main/]
3          [aIHM.generateManifestFile()/]
4
5          [for (aScreen : Screen | aIHM.ihmview.screens)]
6                  [aScreen.generateScreen()/]
```

```
7                   [aScreen.generateLayout()/]
8           [/for]
9
10          [for (aLayout : Layout | aIHM.ihmview.layouts)]
11                  [aLayout.generateLayout()/]
12          [/for]
13          [aIHM.generateResourceFiles()/]
14  [/template]
```

Figure 4.11. main.mtl

a. Generation du fichier Android manifest

La première invocation dans la figure précédente (ligne 3) permet d'appeler le template responsable de la génération du fichier xml AndroidManifest. C'est le rôle du module manifest.mtl qui est présenté dans la Figure 4.12. Les templates sont des modèles paramétrés de la plateforme cible. L'exécution d'une transformation consiste à prendre un template et à remplacer ses paramètres par les valeurs du modèle source. Comme l'illustre la Figure 4.12, le fragment de code permet de déclarer dans le manifeste toutes les activités de l'application (lignes 19-25) et la classe screen est un exemple d'activité dans HCIDL.

```
1   [template public generateManifestFile(aIHM : IHM)]
2           [file ('AndroidManifest.xml', false, 'UTF-8')]
3   <?xml version="1.0" encoding="utf-8"?>
4   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
5     package="[aIHM.packageName/]" >
6     ...
7     ...
8     <application
9       ...
10      <activity
11        android:name=".[aIHM.ihmView.screens->first().name/]Activity"
12        android:label="@string/app_name" >
13        <intent-filter>
14          <action android:name="android.intent.action.MAIN" />
15
16          <category android:name="android.intent.category.LAUNCHER" />
17        </intent-filter>
18      </activity>
19  [if (aIHM.ihmView.screens->size()) > 1]
20          [for (screen : Screen | aIHM.ihmView.screens->subOrderedSet(2, aApplication.screens-
    >size()))]
21                  <activity
22                          android:name=".[screen.name.toUpperFirst()/]Activity">
23                  </activity>
24          [/for]
25  [/if]
26    </application>
27
28  </manifest>
```

```
29          [/file]
30  [/template]
```

Figure 4.12. manifest.mtl

b. Génération des activités de l'application

Comme le montre la Figure 4.11, la boucle dans main.mtl (lignes 5-8) permet de scanner le modèle et de transformer les objets screens en activités de l'application Android. C'est le rôle du module screen.mtl qui est présenté dans la Figure 4.13.

```
1   ...
2   [template public generateScreen(aScreen : Screen)]
3   ...
4   package [aScreen.eContainer(IHM).packageName/];
5   ...
6   public class [aScreen.name.toUpperFirst()/]Activity extends ActionBarActivity{
7
8           @Override
9           protected void onCreate(Bundle savedInstanceState) {
10                  super.onCreate(savedInstanceState);
11      [if (aScreen.layout <> null)]
12                  setContentView(R.layout.[aScreen.layout.name/]);
13      [/if]
14      [if (aScreen.element <> null)]
15        setContentView(R.layout.[aScreen.name.toLower()/]_screen);
16      [/if]
17          }
18  ...
19  [for (action : Action | aScreen.eAllContents(Action))]
20     [action.generateCallback()/]
21  [/for]
22  }
23          [/file]
24  [/template]
```

Figure 4.13. screen.mtl

Dans le module Action, une classe Java est générée pour chaque activité d'application paramétrée avec les valeurs du modèle source. Le code nécessaire pour recevoir les événements d'entrée et effectuer les effets associés est également généré dans l'activité à laquelle ils s'appliquent, à partir de l'objet Action du modèle. Par exemple, le fragment de code du module action.mtl (voir Figure 4.14) génère le code nécessaire pour appeler une page Web par son URL.

```
1   [template public generateCallback(action : Action) post (trim())]
2   //[action.eContainer(View).eClass().name/]([action.eContainer(View).name/]) onClick
3   public void [action.eContainer(View).name/]_[action.eContainingFeature().name/](View view) {
4      [action.actionCode()/]
5   }
6   [/template]
7
```

```
8   [template private actionCode(action : Action) ]
9   //This is an abstract template and should not be called
10  [/template]
11
12  [template private actionCode(action : GoToURL) ]
13  startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("[action.url/]")));
14  [/template]
15  ...
16  [/template]
```

Figure 4.14. action.mtl

c. Génération des layouts et des composants d'interface

Pour chaque activité, un layout est généré pour les événements en sortie avec une modalité d'affichage. C'est le rôle du module layout.mtl (voir Figure 4.15). Ce module est appelé à partir de la ligne 11 du module main.mtl (voir Figure 4.11). Comme le montre la figure 4.15, le module layout.mtl est composé de deux parties; le premier (lignes 2-11) est dédié à la génération des layout relatifs tandis que le second (lignes 13-30) est dédié à la génération des layouts linéaires.

Dans sa première version, le module layout.mtl était responsable de la génération des fichiers XML de mise en page et des nœuds XML des widgets. Dans cette version, nous optons pour une répartition des responsabilités. Ainsi, le module layout.mtl est responsable de la génération des fichiers XML de mise en page et la génération des widgets est déléguée au module widgets.mtl.

```
1   ...
2   [template public generateLayout(aLayout : Layout)]
3   [aLayout.generateLayout(aLayout.name)/]
4   [/template]
5
6   [template public generateLayout(aLayout : Layout, fileName : String)]
7   [file ('res/layout/'.concat(fileName.toLower()).concat('.xml'), false, 'UTF-8')]
8   <?xml version="1.0" encoding="utf-8"?>
9   [generateViewNode(aLayout)/]
10  [/file]
11  [/template]
12
13  [template public generateLayout(aViewCollection : ViewCollection, fileName : String) ]
14  [file ('res/layout/'.concat(fileName.toLower()).concat('.xml'), false, 'UTF-8')]
15  <?xml version="1.0" encoding="utf-8"?>
16  [generateLayoutNode(aViewCollection)/]
17  [/file]
18  [/template]
19
20  [template private generateLayoutNode(collection : ViewCollection)]
21  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
22     android: orientation="vertical"
23     android:layout_width="fill_parent"
```

```
24      android:layout_height="fill_parent"
25      >
26  [for (view : View | collection.views)]
27      [view.generateViewNode()/]
28  [/for]
29  </LinearLayout>
30  [/template]
```

Figure 4.15. layout.mtl

Le fragment de code du module widget.mtl (voir Figure 4.16) permet de créer un nœud xml pour la déclaration de chaque composant de vue.

```
1   ...
2   [template private generateViewNode(view : View, subViews: ViewCollection) post(trim()) ]
3   <[view.nodeTagName()/]
4   [if (view.isRootNode())]
5               xmlns:android="http://schemas.android.com/apk/res/android"
6   [/if]
7   [if (view.name <> null)]
8       android:id="@+id/[view.name/]"
9   [/if]
10      [view.generateLayoutProperties()/]
11              [view.generateViewProperties()/]
12  [if (subViews <> null)]
13  >
14  [for (view : View | subViews.views)]
15              [view.generateViewNode()/]
16  [/for]
17  </[view.nodeTagName()/]>
18  [else]
19  />
20  [/if]
21  [/template]
22  ...
```

Figure 4.16. widget.mtl

d. Génération des fichiers de ressources

La génération de ressources d'application est relativement simple. Il se compose d'un ensemble de déclarations. C'est le rôle du module resource.mtl qui est présenté à la Figure 4.17. Par exemple, le fragment de code présenté dans cette figure permet de générer dans le fichier de ressources strings.xml de l'application, les ressources de type chaîne déclarées dans le modèle :

```
1   [comment encoding = UTF-8 /]
2   [module resource('http://www.xtext.org/example/droid/Droid',
    'http://www.eclipse.org/emf/2002/Ecore')/]
3

4   [template public generateResourceFiles(app : Application)]
5               [app.stringResourceFile()/]
6               [generateResourceFiles(app.resources->asOrderedSet()->filter(Resource))/]
```

```
7   [/template]
8
9   [template private generateResourceFiles(resources : OrderedSet(Resource))]
10          [stringResourceFile(resources->filter(StringResource))/]
11          [integerResourceFile(resources->filter(IntegerResource))/]
12          [booleanResourceFile(resources->filter(BooleanResource))/]
13          [colorResourceFile(resources->filter(ColorResource))/]
14          [dimensionResourceFile(resources->filter(DimensionResource))/]
15          [integerArrayResourceFile(resources->filter(IntegerArrayResource))/]
16          [stringArrayResourceFile(resources->filter(StringArrayResource))/]
17  [/template]
18
19  [template private stringResourceFile(app : Application)]
20          [file ('res/values/strings.xml', false, 'UTF-8')]
21  <?xml version="1.0" encoding="utf-8"?>
22  <resources>
23
24      <string name="app_name">[app.name/]</string>
25      <string name="hello_world">Hello world!</string>
26      <string name="action_settings">Settings</string>
27
28  </resources>
29          [/file]
30  [/template]
31
32  [template private stringResourceFile(resources : OrderedSet(StringResource))]
33  [if (resources->size() >0)]
34          [file ('res/values/gen-strings.xml', false, 'UTF-8')]
35  <?xml version="1.0" encoding="utf-8"?>
36  <resources>
37          [for (resource : StringResource | resources)]
38      <string name="[resource.name/]">[resource.value/]</string>
39      [/for]
40  </resources>
41          [/file]
42  [/if]
43  [/template]
44  ...
```

Figure 4.17. resource.mtl

4.5. DISCUSSION ET SYNTHÈSE

Nous avons présenté dans ce chapitre notre contribution à savoir HCIDL, un langage de description pour la modélisation et la génération d'interfaces utilisateur multi-cibles multimodales et plastique. Notre travail est motivé par la difficulté du développement des

interfaces applicatives, combiné à la multiplicité et à la diversité des plateformes d'interactions existantes.

Notre approche respecte les critères principaux du MDA définissant un développement basé sur un modèle efficace. Nous proposons un langage de modélisation textuel clair et structuré. Son originalité réside dans sa structure en trois packages selon le modèle MVC, ce qui accentue sa modularité. A ce stade de son développement, HCIDL permet de supporter à la fois les interactions multimodales en entrée et en sortie. Le niveau d'abstraction offert par HCIDL permet la gestion des capteurs existants sur les appareils mobiles. Grâce à l'agencement et au mécanisme de positionnement des composants d'interface implémentés dans le package "View", HCIDL définit une disposition optimale pour les interfaces d'application. Nous avons détaillé le métamodèle qui constitue notre langage, aussi appelé syntaxe abstraite. Puis, à travers un exemple d'application nous avons illustré un exemple de modèle rédigé grâce à notre langage. Cependant, pour une vision complète et détaillée de notre langage le lecteur peut se référer à l'annexe de cette thèse. En complément, nous avons détaillés les étapes suivies pour la génération du code de l'interface de notre exemple à partir des modèles formulés en utilisant notre langage HCIDL. Les étapes de modélisation, de génération automatique de code et de déploiement de l'application ne nécessitent pas de migration et sont effectuées dans le même environnement, celui d'*Eclipse Modeling Framework*.

PARTIE IV
CONCLUSION GÉNÉRALE

CONCLUSION

1. RÉSUMÉ DES CONTRIBUTIONS

De nos investigations, nous avons présenté à travers cette thèse, HCIDL notre langage de description d'interface utilisateur. L'objectif est d'améliorer la convivialité des interfaces utilisateurs via un comportement adaptatif grâce à un ensemble d'interactions adapté aux entrées/sorties des terminaux et à une disposition optimale. Nous proposons d'utiliser à la fois la plasticité et la multimodalité comme support pour le remodelage de l'IHM, lorsque les travaux de la littérature ne se concentrent que sur l'un ou l'autre de ces caractéristiques à la fois. Le but de ce travail est l'adaptation de l'interface utilisateur en fonction des différents paramètres qui, une fois combinés, constituent un dispositif d'interaction. Plus précisément:

- Le terme "multi-cible" signifie que l'interface utilisateur est destinée à prendre en charge plusieurs dispositifs d'interaction. Le terme cible dans ce cas fait référence à des dispositifs d'interaction fixes ou mobiles.
- La dimension "plastique" de notre approche dénote l'aspect esthétique de l'adaptation de l'interface utilisateur. La disposition des composants de l'interface utilisateur dans la zone d'affichage implique que l'interface conserve sa facilité d'utilisation malgré les caractéristiques physiques du dispositif d'interaction cible.
- La "multimodalité" contribue à la plasticité de l'interface utilisateur. Dans le cadre de notre approche, la multimodalité offre des alternatives à l'interaction. Que ce soit en entrée ou en sortie, l'interface utilisateur peut s'adapter aussi bien à des dispositifs d'interaction luxueux qu'à des dispositifs d'interaction moins riches en termes de modalités d'interaction.

Notre approche respecte les critères principaux du MDA définissant un développement basé sur un modèle efficace. Nous proposons un langage de modélisation textuel clair et structuré. Son originalité réside dans sa structure en trois paquets selon le modèle MVC (*Model – View - Controller*), ce qui accentue sa modularité. Les étapes de modélisation, de génération de code et de déploiement de l'application ne nécessitent pas de migration et sont effectuées dans le même environnement. A ce stade de son développement, HCIDL permet de supporter à la fois les interactions multimodales en entrée et en sortie. Le niveau d'abstraction offert par HCIDL permet la gestion des capteurs existants sur les appareils mobiles. Grâce au mécanisme de positionnement des composants d'interface implémentés dans le package "View", HCIDL définit une disposition optimale pour les interfaces d'application.

Ainsi, les contributions de nos recherches sont les suivantes :

- Nous proposons un langage de description d'interface utilisateur inspiré des langages de modélisation existants tels que SMUIML et M4L. Notre langage de modélisation est structuré en trois paquets selon le modèle MVC. L'utilisateur interagit directement avec le package de présentation, qui est connecté au package de contrôle d'interaction qui

- détermine les modalités d'interaction et les services d'application devant être sollicités. Le package modèle permet d'accéder aux données de la partie fonctionnelle.
- HCIDL permet de soutenir à la fois les interactions multimodales en entrée et en sortie. Sur la base des propriétés CARE et TYCOON, nous considérons quatre coopérations entre les modalités : équivalence, concurrence, redondance et complémentarité.
- Le niveau d'abstraction offert par HCIDL permet la gestion des capteurs existants sur les appareils mobiles.
- Pour définir une disposition optimale des interfaces d'application, nous développons deux méthodes de positionnement. Une approche "linéaire" pour les interfaces utilisateur simples où les composants s'alignent les uns après les autres. Une seconde approche "relative", pour des interfaces plus complexes, où la position d'un composant est exprimée par rapport à la position d'un autre composant.
- Nous proposons une étude comparative basée sur des critères que nous avons définis et englobant la multimodalité, la plasticité, le niveau d'abstraction, l'hétérogénéité et la démarche de développement.

2. ORIGINALITIES ET POINTS FORTS

Dans cette thèse, nous avons présentés nos contributions dans le domaine de l'ingénierie homme-machine pour la conception d'interfaces d'interaction multi-cibles, multimodales et plastiques. Nous avons défini une approche à base de modèle en suivant un processus conforme à l'approche MDA. Elle comporte le langage HCIDL pour la modélisation des interfaces d'interactions, ainsi qu'un ensemble de règles de générations pour la génération du code en Android.

Ces travaux constituent une étape pour l'optimisation du développement des interfaces d'applications destinées à plusieurs dispositifs. Que ce soit des appareils à base de modalités WIMP ou encore des appareils riches de nouvelles modalités à base de capteurs et de combinaison de modalités en entrée et/ou en sortie.

Nous exposons dans cette section les originalités et points forts de notre approche. Ainsi, notre approche respecte les critères principaux du MDA définissant un développement basé sur un modèle efficace. Ensuite, nous fournissons un langage de modélisation textuel clair et structuré. Son originalité réside dans sa structure en trois packages selon le modèle MVC, ce qui accentue sa modularité. Les étapes de modélisation, de génération automatique de code et de déploiement d'application ne nécessitent pas de migration et sont effectuées dans le même environnement. A ce stade de son développement, HCIDL permet de supporter à la fois les interactions multimodales en entrée et en sortie. Le niveau d'abstraction offert par HCIDL permet la gestion des capteurs existants sur les appareils mobiles. Grâce à l'agencement et au mécanisme de positionnement des composants d'interface implémentés dans le package "View", HCIDL définit une disposition optimale pour les interfaces d'application.

3. LIMITES

Nous sommes conscients que notre travail est loin de la perfection et nécessite plus d'étude pour plus de maturité. Ainsi, nous avons notés quelques limitations technologiques et conceptuelles :

- Parmi les limitations technologiques liées aux outils que nous utilisons nous avons noté une instabilité de notre environnement de développement Eclipse+Xtext+Acceleo+Android de sorte qu'une fois combiné ceux-ci sont sujets à de nombreux bugs et en particulier Acceleo.
- Notre approche ne peut pas traiter des applications existantes. De ce fait, le concepteur d'interfaces est obligé de modéliser au moins une fois son interface avec notre langage avant de procéder à sa génération.
- Même si notre langage HCIDL permet de modéliser les positions exactes des composants graphiques sur l'interface, néanmoins il y une limite à la complexité que HCIDL peut gérer. Ainsi, nous n'avons effectué aucun teste pour des applications d'animation ou de jeux.
- A ce stade, notre approche ne permet la génération de code que pour la plateforme Android.

4. PERSPECTIVES

Plusieurs perspectives se présentent à nous pour la suite de notre travail de recherche. À court terme, nous visons à compléter notre générateur afin de pouvoir cibler de nouvelles plateformes. Par la suite, nous essaierons, en utilisant notre métamodèle, d'identifier les combinaisons de modalités les plus élémentaires, de les modéliser et de permettre leur réutilisation lors de la modélisation de nouvelles applications. Par la suite, un projet pour le développement de notre propre environnement de construction d'interface utilisateur basé sur notre langage de description d'interface utilisateur HCIDL sera considéré.

ANNEXE A
GRAMMAIRE DU LANGAGE HCIDL

```
// automatically generated by Xtext
grammar xtext.eclipse.Ihm with org.eclipse.xtext.common.Terminals

import 'http://www.lgaouar.org/ihm/View' as View
import 'http://www.eclipse.org/emf/2002/Ecore' as ecore
import 'http://www.lgaouar.org/ihm/Model' as Model
import 'http://www.lgaouar.org/ihm' as ihm
import 'http://www.lgaouar.org/Controller' as Controller



IHMRule returns ihm::IHM:
	('application' name=STRING)?
	('package' packageName=PackageName)?
	(includes+=IncludeRule)*

	((ihmModel=ModelRule)? & (ihmView=IhmViewRule)? &
(ihmController=ControllerRule)?)
;

IncludeRule returns ihm::Include:
    '%include' '<' importURI=INCLUDE '>'
;

terminal INCLUDE:
	ID('.ihm')
;

terminal PackageName:
	ID('.'ID)*
;

/* ************************************************************************ *
 *                                 _   _   _
 *                            |\/| | | | \ |_ |
 *                            |  | |_| |_/ |_ |_
 *
 * ************************************************************************ */

ModelRule returns Model::IhmModel:
	'#' 'IhmModel'
	(datasources+=DataSourceRule | resources+=ResourceRule)+
;

DataSourceRule returns Model::DataSource:
	'datasource' name=ID '=' '{' access=Access ',' return=Return '}'
;

enum Access returns Model::Access:
	remote | local
 ;
```

```
52
53  enum Return returns Model::Return:
54        json | xml | entity
55   ;
56
57  ResourceRule returns Model::Resource:
58        StringResourceRule | IntegerResourceRule | BooleanResourceRule |
59  ColorResourceRule | DimensionResourceRule | ArrayResourceRule | ImageResourceRule
60  ;
61
62  StringResourceRule returns Model::StringResource:
63        name=ID '=' value=STRING
64  ;
65
66  IntegerResourceRule returns Model::IntegerResource:
67        name=ID '=' value=INT
68  ;
69
70  BooleanResourceRule returns Model::BooleanResource:
71        name=ID '=' value=Bool
72  ;
73
74  enum Bool returns Model::Bool:
75        True | False
76  ;
77
78  ColorResourceRule returns Model::ColorResource:
79        name=ID '=' value=HEX_COLOR
80  ;
81
82  terminal HEX_COLOR:
83        '#'
84        ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
85        ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
86        ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
87        (('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f'))?
88  ;
89
90  DimensionResourceRule returns Model::DimensionResource:
91        name=ID '=' value=DimensionValue
92  ;
93
94  terminal DimensionValue:
95        FLOAT ('dp' | 'sp' | 'pt' | 'px' | 'mm' | 'in')
96  ;
97
98  terminal FLOAT:
99        INT ('.'INT)?
100 ;
101
102 ArrayResourceRule returns Model::ArrayResource:
103       IntegerArrayResourceRule | StringArrayResourceRule
104 ;
105
106 IntegerArrayResourceRule returns Model::IntegerArrayResource :
107       name=ID '=' '[' items+=INT (',' items+=INT)* ']'
108 ;
109
110 StringArrayResourceRule returns Model::StringArrayResource:
```

```
111         name=ID '=' '[' items+=STRING (',' items+=STRING)* ']'
112     ;
113
114     ImageResourceRule returns Model::ImageResource:
115         name=ID '=' fileName=STRING
116     ;
117
118     ValueAccessRule returns Model::ValueAccess:
119         (StringVARule | IntegerVARule | BooleanVARule | ColorVARule | ImageVARule |
120     DimensionVARule | LayoutDimensionVARule) | ResourceAccessRule
121     ;
122
123     LayoutDimensionVARule returns Model::LayoutDimensionVA:
124         DimensionVARule | (const_value=LayoutDimensionKind)
125     ;
126
127     enum LayoutDimensionKind returns Model::LayoutDimensionKind:
128         fill_parent | match_parent | wrap_content
129     ;
130
131     ColorVARule returns Model::ColorVA:
132         ColorRARule | value=HEX_COLOR
133     ;
134
135     ImageVARule returns Model::ImageVA:
136         ImageRARule
137     ;
138
139     StringVARule returns Model::StringVA:
140         StringRARule | value=STRING
141     ;
142
143     IntegerVARule returns Model::IntegerVA:
144         IntegerRARule | value=INT
145     ;
146
147     BooleanVARule returns Model::BooleanVA:
148         BooleanRARule | value=Bool
149     ;
150
151     DimensionVARule returns Model::DimensionVA:
152         DimensionRARule | value=DimensionValue
153     ;
154
155     ResourceAccessRule returns Model::ResourceAccess:
156         StringRARule | IntegerRARule | BooleanRARule | DimensionRARule | ColorRARule
157     | ImageRARule | ArrayRARule
158     ;
159
160     StringRARule returns Model::StringRA:
161         'string'
162         (
163               ('(' (package=PackageName '/')? resource=[Model::StringResource] ')')
164         |     ('[' (package=PackageName '/')? externalResource=ID ']')
165         )
166     ;
167
168     IntegerRARule returns Model::IntegerRA:
169         'integer'
```

```
	(
		('(' (package=PackageName '/')? resource=[Model::IntegerResource]
')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;

BooleanRARule returns Model::BooleanRA:
	'bool'
	(
		('(' (package=PackageName '/')? resource=[Model::BooleanResource]
')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;

ColorRARule returns Model::ColorRA:
	'color'
	(
		('(' (package=PackageName '/')? resource=[Model::ColorResource] ')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;

DimensionRARule returns Model::DimensionRA:
	'dimen'
	(
		('(' (package=PackageName '/')? resource=[Model::DimensionResource]
')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;

ArrayRARule returns Model::ArrayRA:
	'array'
	(
		('(' (package=PackageName '/')? resource=[Model::ArrayResource] ')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;

ImageRARule returns Model::ImageRA:
	'image'
	(
		('(' (package=PackageName '/')? resource=[Model::ImageResource] ')')
	|	('[' (package=PackageName '/')? externalResource=ID ']')
	)
;
/* ********************************************************************** *
 *                                 _
 *                            \  / | |_ \    /
 *                             \/  | |_  \/\/
 *
 * ********************************************************************** */

IhmViewRule returns View::IhmView:
	'#' 'IhmView'
	(navbar+=MenuRule | screens+=ScreenRule | layouts+=LayoutRule)+
;
```

```
229
230   MenuRule returns View::Menu:
231         'menu' name=ID '=' '{'
232   (menuitems+=MenuItemRule)(';'(menuitems+=MenuItemRule))* '}'
233   ;
234
235   MenuItemRule returns View::MenuItem:
236         '(' name=ID ',' text=STRING ')'
237   ;
238
239   ScreenRule returns View::Screen:
240         'screen' name=ID
241         '{'
242               ('title' title=[Model::StringVA])?
243               ('menu' menu=[View::Menu])?
244               ('show' layout=[View::Layout] | element=ViewCollectionRule)
245         '}'
246   ;
247
248   LayoutRule returns View::Layout:
249         (isRelative?='relative')? 'layout' name=ID '{'
250
251               //View Properties
252                ('background:' background=ColorVARule)?
253               &('minHeight:' minHeight=DimensionVARule)?
254               &('minWidth:' minWidth=DimensionVARule)?
255               &('padding:' padding=DimensionVARule)?
256               &('paddingBottom:' paddingBottom=DimensionVARule)?
257               &('paddingLeft:' paddingLeft=DimensionVARule)?
258               &('paddingRight:' paddingRight=DimensionVARule)?
259               &('paddingTop:' paddingTop=DimensionVARule)?
260               &('scrollbars:' scrollbars=BooleanVARule)?
261
262               &(layoutProperties=LayoutPropertiesRule)?
263
264               //Layout Properties
265               &('gravity:' gravity=LayoutGravityKind)?
266               &('orientation:' orientation=LayoutOrientationKind)?
267
268               views=ViewCollectionRule
269         '}'
270   ;
271
272   enum LayoutGravityKind returns View::LayoutGravityKind:
273         top | bottom | left | right | center | center_vertical | center_horizontal |
274   fill | fill_vertical | fill_horizontal
275   ;
276
277   enum LayoutOrientationKind returns View::LayoutOrientationKind:
278         horizontal | vertical
279   ;
280
281   LayoutPropertiesRule returns View::LayoutProperties:
282         //LinearLayoutParams
283          ('layout_height:' layout_height=LayoutDimensionVARule)?
284         &('layout_width:' layout_width=LayoutDimensionVARule)?
285         &('layout_marginBottom:' layout_marginBottom=DimensionVARule)?
286         &('layout_marginLeft:' layout_marginLeft=DimensionVARule)?
287         &('layout_marginRight:' layout_marginRight=DimensionVARule)?
```

```
288         &('layout_marginTop:' layout_marginTop=DimensionVARule)?
289
290         //RelativeLayout
291         &('layout_above:' layout_above=[View::View])?
292         &('layout_alignBottom:' layout_alignBottom=[View::View])?
293         &('layout_alignLeft:' layout_alignLeft=[View::View])?
294         &('layout_alignTop:' layout_alignTop=[View::View])?
295         &('layout_below:' layout_below=[View::View])?
296         &('layout_toLeftOf:' layout_toLeftOf=[View::View])?
297         &('layout_toRightOf:' layout_toRightOf=[View::View])?
298   ;
299
300   ViewCollectionRule returns View::ViewCollection:
301         ('#' views+=ViewRule)+
302   ;
303
304   ViewRule returns View::View:
305         GUIElementRule | LayoutRule
306   ;
307
308   GUIElementRule returns View::GUIElement:
309         ButtonRule | LabelRule | ListRule | TextFieldRule | CheckBoxRule | ImageRule
310   | SwitchRule | RadioGroupRule | SpinnerRule
311   ;
312
313   ButtonRule returns View::Button:
314         'button' (name=ID)?
315         '{'
316                ('text:' text=StringVARule)
317               &(layoutProperties=LayoutPropertiesRule)?
318
319               &('top:' top=DimensionVARule)?
320               &('left:' left=DimensionVARule)?
321               &('width:' width=DimensionVARule)?
322               &('height:' height=DimensionVARule)?
323               &('background:' background=ColorVARule)?
324         '}'
325   ;
326
327   LabelRule returns View::Label:
328         'label' (name=ID)?
329         '{'
330                ('text:' text=StringVARule)?
331               &(layoutProperties=LayoutPropertiesRule)?
332
333               &('top:' top=DimensionVARule)?
334               &('left:' left=DimensionVARule)?
335               &('width:' width=DimensionVARule)?
336               &('height:' height=DimensionVARule)?
337               &('background:' background=ColorVARule)?
338         '}'
339   ;
340
341   ListRule returns View::List:
342         'list' (name=ID)? '[' items+=ListItemRule (';'items+=ListItemRule)* ']'
343         '{'
344                ('selectedItem:' selectedItem=IntegerVARule)?
345               &(layoutProperties=LayoutPropertiesRule)?
346
```

```
347               &('top:' top=DimensionVARule)?
348               &('left:' left=DimensionVARule)?
349               &('width:' width=DimensionVARule)?
350               &('height:' height=DimensionVARule)?
351               &('background:' background=ColorVARule)?
352         '}'
353   ;
354
355   ListItemRule returns View::ListItem:
356         '(' id=INT ',' item=ValueAccessRule ')'
357   ;
358
359   TextFieldRule returns View::TextField:
360         EditTextRule | EmailTextRule | PasswordTextRule | NumTextRule
361   ;
362
363   enum TypefaceKind returns View::TypefaceKind:
364         normal | sans | serif | monospace
365   ;
366
367   enum TextStyleKind returns View::TextStyleKind:
368         normal | bold | italic
369   ;
370
371   enum CapitalizeKind returns View::CapitalizeKind:
372         none | sentences | serif | words | characters
373   ;
374
375   EditTextRule returns View::EditText:
376         'editText' (name=ID)?
377         '{'
378                ('text:' text=StringVARule)?
379               &(layoutProperties=LayoutPropertiesRule)?
380
381               &('top:' top=DimensionVARule)?
382               &('left:' left=DimensionVARule)?
383               &('width:' width=DimensionVARule)?
384               &('height:' height=DimensionVARule)?
385               &('background:' background=ColorVARule)?
386
387               //Text attributes
388               &('capitalize:' capitalize=CapitalizeKind)?
389               &('editable:' editable=BooleanVARule)?
390               &('gravity:' gravity=LayoutGravityKind)?
391               &('hint:' hint=StringVARule)?
392               &('singleLine:' singleLine=BooleanVARule)?
393               &('textColor:' textColor=ColorVARule)?
394               &('typeface:' typeface=TypefaceKind)?
395               &('textSize:' textSize=DimensionVARule)?
396               &('textStyle:' textStyle+=TextStyleKind ('|'
397   textStyle+=TextStyleKind)*)?
398         '}'
399   ;
400
401   EmailTextRule returns View::EmailText:
402         'emailText' (name=ID)?
403         '{'
404                ('text:' text=email)?
405               &(layoutProperties=LayoutPropertiesRule)?
```

```
406
407                 &('top:' top=DimensionVARule)?
408                 &('left:' left=DimensionVARule)?
409                 &('width:' width=DimensionVARule)?
410                 &('height:' height=DimensionVARule)?
411                 &('background:' background=ColorVARule)?
412
413                 //Text attributes
414                 &('capitalize:' capitalize=CapitalizeKind)?
415                 &('editable:' editable=BooleanVARule)?
416                 &('gravity:' gravity=LayoutGravityKind)?
417                 &('hint:' hint=StringVARule)?
418                 &('singleLine:' singleLine=BooleanVARule)?
419                 &('textColor:' textColor=ColorVARule)?
420                 &('typeface:' typeface=TypefaceKind)?
421                 &('textSize:' textSize=DimensionVARule)?
422                 &('textStyle:' textStyle+=TextStyleKind ('|'
423   textStyle+=TextStyleKind)*)?
424         '}'
425   ;
426
427   email:
428         STRING'@'STRING'.'('com' | 'fr' | 'org' | 'dz')
429   ;
430
431   PasswordTextRule returns View::PasswordText:
432         'passwordText' (name=ID)?
433         '{'
434                  ('defaultChar:' defaultChar=STRING)?
435                &(layoutProperties=LayoutPropertiesRule)?
436
437                 &('top:' top=DimensionVARule)?
438                 &('left:' left=DimensionVARule)?
439                 &('width:' width=DimensionVARule)?
440                 &('height:' height=DimensionVARule)?
441                 &('background:' background=ColorVARule)?
442
443                 //Text attributes
444                 &('capitalize:' capitalize=CapitalizeKind)?
445                 &('editable:' editable=BooleanVARule)?
446                 &('gravity:' gravity=LayoutGravityKind)?
447                 &('hint:' hint=StringVARule)?
448                 &('singleLine:' singleLine=BooleanVARule)?
449                 &('textColor:' textColor=ColorVARule)?
450                 &('typeface:' typeface=TypefaceKind)?
451                 &('textSize:' textSize=DimensionVARule)?
452                 &('textStyle:' textStyle+=TextStyleKind ('|'
453   textStyle+=TextStyleKind)*)?
454         '}'
455   ;
456
457   NumTextRule returns View::NumText:
458         'numText' (name=ID)?
459         '{'
460                  ('text:' text=IntegerVARule)?
461                &(layoutProperties=LayoutPropertiesRule)?
462
463                 &('top:' top=DimensionVARule)?
464                 &('left:' left=DimensionVARule)?
```

```
465             &('width:' width=DimensionVARule)?
466             &('height:' height=DimensionVARule)?
467             &('background:' background=ColorVARule)?
468 
469             //Text attributes
470             &('capitalize:' capitalize=CapitalizeKind)?
471             &('editable:' editable=BooleanVARule)?
472             &('gravity:' gravity=LayoutGravityKind)?
473             &('hint:' hint=StringVARule)?
474             &('singleLine:' singleLine=BooleanVARule)?
475             &('textColor:' textColor=ColorVARule)?
476             &('typeface:' typeface=TypefaceKind)?
477             &('textSize:' textSize=DimensionVARule)?
478             &('textStyle:' textStyle+=TextStyleKind ('|'
479 textStyle+=TextStyleKind)*)?
480       '}'
481 ;
482 
483 CheckBoxRule returns View::CheckBox:
484       'checkbox' (name=ID)?
485       '{'
486              ('isChecked:' isChecked?='on')?
487             &('text:' text=StringVARule)?
488             &(layoutProperties=LayoutPropertiesRule)?
489 
490             &('top:' top=DimensionVARule)?
491             &('left:' left=DimensionVARule)?
492             &('width:' width=DimensionVARule)?
493             &('height:' height=DimensionVARule)?
494             &('background:' background=ColorVARule)?
495       '}'
496 ;
497 
498 ImageRule returns View::Image:
499       'image' (name=ID)?
500       '{'
501              ('src:' src=ImageVARule)?
502             &(layoutProperties=LayoutPropertiesRule)?
503 
504             &('top:' top=DimensionVARule)?
505             &('left:' left=DimensionVARule)?
506             &('width:' width=DimensionVARule)?
507             &('height:' height=DimensionVARule)?
508             &('background:' background=ColorVARule)?
509       '}'
510 ;
511 
512 SwitchRule returns View::Switch:
513       'switch' (name=ID)?
514       '{'
515              ('on:' on?='on')?
516             &(layoutProperties=LayoutPropertiesRule)?
517 
518             &('top:' top=DimensionVARule)?
519             &('left:' left=DimensionVARule)?
520             &('width:' width=DimensionVARule)?
521             &('height:' height=DimensionVARule)?
522             &('background:' background=ColorVARule)?
523       '}'
```

```
;

RadioGroupRule returns View::RadioGroup:
	'radioGroup' (name=ID)? '=' '[' radio+=radioRule (';'(radio+=radioRule))*
']'
	'{'
		(layoutProperties=LayoutPropertiesRule)?

		&('top:' top=DimensionVARule)?
		&('left:' left=DimensionVARule)?
		&('width:' width=DimensionVARule)?
		&('height:' height=DimensionVARule)?
		&('background:' background=ColorVARule)?
	'}'
;

radioRule returns View::RadioButton:
	'(' id=INT ',' text=StringVARule ',' isSelected=BooleanVARule ')'
;

SpinnerRule returns View::Spinner:
	'Spinner' (name=ID)?
	'{'
		('prompt:' prompt=StringVARule )?
		&(layoutProperties=LayoutPropertiesRule)?

		&('top:' top=DimensionVARule)?
		&('left:' left=DimensionVARule)?
		&('width:' width=DimensionVARule)?
		&('height:' height=DimensionVARule)?
		&('background:' background=ColorVARule)?
	'}'
;
/* ********************************************************************** *
 *
 *                      _   _         ___   _   _           _   _
 *                     |   | | |\ |    |   |_| | | |   |   |_  |_|
 *                     |_  |_| | \|    |   |\  |_| |_  |_  |_  | \
 *
 * ********************************************************************** */

ControllerRule returns Controller::IhmController:
	{Controller::IhmController}
	'#' 'IhmController'
	(controls+=ControlRule)?
;

ControlRule returns Controller::Control:
	'when' 'begin' when=InputEventRule 'end'
	'do' 'begin' do=ActionRule 'end'
	'with' 'begin' achievedWith=OutputEventRule 'end'
;

InputEventRule returns Controller::InputEvent:
	'INPUT'
	UserEventRule | SystemEventRule
;

InputCoopRule returns Controller::InputCoop:
```

```
582         (InEquivalenceRule | InConcurrencyRule | InRedundancyRule |
583    InComplementarityRule)
584    ;
585
586    InEquivalenceRule returns Controller::InEquivalence:
587         'Equivalence' '=' '(' with_=[Controller::InputEvent] ',' laps=INT ')'
588    ;
589
590    InConcurrencyRule returns Controller::InConcurrency:
591         'Concurrency' '=' '(' with_=[Controller::InputEvent] ',' laps=INT ')'
592    ;
593
594    InRedundancyRule returns Controller::InRedundancy:
595         'Redundancy' '=' '(' with_=[Controller::InputEvent] ',' laps=INT ')'
596    ;
597
598    InComplementarityRule returns Controller::InComplementarity:
599         'Complementarity' '=' '(' with_=[Controller::InputEvent] ',' laps=INT ')'
600    ;
601
602    InputInteractionRule returns Controller::InputInteraction:
603         '('type=InputTypeInteraction ',' modality=InputModalityInteraction')'
604    ;
605
606    enum InputTypeInteraction returns Controller::InputTypeInteraction:
607         touch | longTouch | vocal | scanQR | orientation | blackout | illumination |
608    approach | moveAway | positioning | recording
609    ;
610
611    enum InputModalityInteraction returns Controller::InputModalityInteraction:
612         touchScreen | speech | scan | orientation | lighting | proximity | location
613    | music
614    ;
615
616
617    UserEventRule returns Controller::UserEvent:
618         'USER'
619         (PressedRule | EditedRule | CheckedRule | SwitchedRule | ListChangedRule)
620
621    ;
622
623    PressedRule returns Controller::Pressed:
624         name=ID
625         'pressed' '[' source=[View::GUIElement] ']'
626         'has' interaction=InputInteractionRule
627         ('incop' InCop+=InputCoopRule)*
628    ;
629
630    EditedRule returns Controller::Edited:
631         name=ID
632         'edited' '[' source=[View::GUIElement] ']'
633         'has' interaction=InputInteractionRule
634         ('incop' InCop+=InputCoopRule)*
635    ;
636
637    CheckedRule returns Controller::Checked:
638         name=ID
639         'checked' '[' source=[View::GUIElement] ']'
640         'has' interaction=InputInteractionRule
```

```
641         ('incop' InCop+=InputCoopRule)*
642     ;
643
644     SwitchedRule returns Controller::Switched:
645         name=ID
646         'switched' '[' source=[View::Switch] ']'
647         'has' interaction=InputInteractionRule
648         ('incop' InCop+=InputCoopRule)*
649     ;
650
651     ListChangedRule returns Controller::ListChanged:
652         name=ID
653         'listselectionchanged' '[' source=[View::List] ']'
654         'has' interaction=InputInteractionRule
655         ('incop' InCop+=InputCoopRule)*
656     ;
657
658     SystemEventRule returns Controller::SystemEvent:
659         'SYSTEM'
660         (CallRule | SmsRule | NetworkRule | BatteryRule | WifiRule | GpsRule |
661     SleepRule)
662     ;
663
664     CallRule returns Controller::Call:
665         IncomingCallRule | OutgoingCallRule
666     ;
667
668     IncomingCallRule returns Controller::IncomingCall:
669         name=ID
670         'incomingcall'
671         'has' interaction=InputInteractionRule
672         ('incop' InCop+=InputCoopRule)*
673     ;
674
675     OutgoingCallRule returns Controller::OutgoingCall:
676         name=ID
677         'outgoingcall'
678         'has' interaction=InputInteractionRule
679         ('incop' InCop+=InputCoopRule)*
680     ;
681
682     SmsRule returns Controller::SMS:
683         IncomingSmsRule | OutgoingSmsRule
684     ;
685
686     IncomingSmsRule returns Controller::IncomingSMS:
687         name=ID
688         'incomingsms'
689         'has' interaction=InputInteractionRule
690         ('incop' InCop+=InputCoopRule)*
691     ;
692
693     OutgoingSmsRule returns Controller::OutgoingSMS:
694         name=ID
695         'outgoingsms'
696         'has' interaction=InputInteractionRule
697         ('incop' InCop+=InputCoopRule)*
698     ;
699
```

```
700   NetworkRule returns Controller::Network:
701         NetworkAvailableRule | NetworkUnavailableRule
702   ;
703
704   NetworkAvailableRule returns Controller::NetworkAvailable:
705         name=ID
706         'networkavailable'
707         'has' interaction=InputInteractionRule
708         ('incop' InCop+=InputCoopRule)*
709   ;
710
711   NetworkUnavailableRule returns Controller::NetworkUnavailable:
712         name=ID
713         'networkunavailable'
714         'has' interaction=InputInteractionRule
715         ('incop' InCop+=InputCoopRule)*
716   ;
717
718   BatteryRule returns Controller::Battery:
719         name=ID
720         'lowbattery'
721         'has' interaction=InputInteractionRule
722         ('incop' InCop+=InputCoopRule)*
723   ;
724
725   GpsRule returns Controller::GPS:
726         GpsOnRule | GpsOffRule
727   ;
728
729   GpsOnRule returns Controller::GPSOn:
730         name=ID
731         'gpson'
732         'has' interaction=InputInteractionRule
733         ('incop' InCop+=InputCoopRule)*
734   ;
735
736   GpsOffRule returns Controller::GPSOff:
737         name=ID
738         'gpsoff'
739         'has' interaction=InputInteractionRule
740         ('incop' InCop+=InputCoopRule)*
741   ;
742
743   WifiRule returns Controller::Wifi:
744         WifiOnRule | WifiOffRule
745   ;
746
747   WifiOnRule returns Controller::WifiOn:
748         name=ID
749         'wifion'
750         'has' interaction=InputInteractionRule
751         ('incop' InCop+=InputCoopRule)*
752   ;
753
754   WifiOffRule returns Controller::WifiOff:
755         name=ID
756         'wifioff'
757         'has' interaction=InputInteractionRule
758         ('incop' InCop+=InputCoopRule)*
```

```
;

SleepRule returns Controller::Sleep:
	EnterSleepRule | EndSleepRule
;

EnterSleepRule returns Controller::EnterSleep:
	name=ID
	'sleep'
	'has' interaction=InputInteractionRule
	('incop' InCop+=InputCoopRule)*
;

EndSleepRule returns Controller::EndSleep:
	name=ID
	'endsleep'
	'has' interaction=InputInteractionRule
	('incop' InCop+=InputCoopRule)*
;

ActionRule returns Controller::Action:
	PopupRule | NavigationRule | CheckForRule | StartRule | StopRule | FetchRule
| SetRule
;

PopupRule returns Controller::Popup:
	'popup' menu=MenuRule
;

NavigationRule returns Controller::navigation:
	(OpenRule | CloseRule) '[' screen=[View::Screen] ']'
;

CheckForRule returns Controller::CheckFor:
	'checkfor'
;

OpenRule returns Controller::open:
	'open'
;

CloseRule returns Controller::close:
	'close'
;

FetchRule returns Controller::fetch:
	'fetch' '[' datasource=[Model::DataSource] ']'
	'into' '[' into=[View::List] ']'
;

StartRule returns Controller::Start:
	'start' '[' service=STRING ']'
;

SetRule returns Controller::Set:
	'set' '[' element=[View::GUIElement] ':' value=STRING ']'
;

StopRule returns Controller::Stop:
```

```
818         'stop' '[' service=STRING ']'
819     ;
820
821     OutputInteractionRule returns Controller::OutputInteraction:
822         '('type=OutputTypeInteraction ',' modality=OutputModalityInteraction')'
823     ;
824
825     enum OutputTypeInteraction returns Controller::OutputTypeInteraction:
826         music | vibration | longVibration | vocalSynthesis | updateView
827     ;
828
829     enum OutputModalityInteraction returns Controller::OutputModalityInteraction:
830         music | vibration | vocalSynthesis | display
831     ;
832
833     OutputEventRule returns Controller::OutputEvent:
834         'OUTPUT'
835         name=ID
836         'has' interaction=OutputInteractionRule
837         ('outcop' OutCop+=OutputCoopRule)*
838     ;
839
840     OutputCoopRule returns Controller::OutputCoop:
841         (OutEquivalenceRule | OutConcurrencyRule | OutRedundancyRule |
842     OutComplementarityRule)
843     ;
844
845     OutEquivalenceRule returns Controller::OutEquivalence:
846         'Equivalence' '=' '(' with_=[Controller::OutputEvent] ',' laps=INT ')'
847     ;
848
849     OutConcurrencyRule returns Controller::OutConcurrency:
850         'Concurrency' '=' '(' with_=[Controller::OutputEvent] ',' laps=INT ')'
851     ;
852
853     OutRedundancyRule returns Controller::OutRedundancy:
854         'Redundancy' '=' '(' with_=[Controller::OutputEvent] ',' laps=INT ')'
855     ;
856
857     OutComplementarityRule returns Controller::OutComplementarity:
858         'Complementarity' '=' '(' with_=[Controller::OutputEvent] ',' laps=INT ')'
859     ;
```

BIBLIOGRAPHIE

Aho, A. V., Sethi, R. et Ullman, J. D. (1986). Compilers : Principles, Techniques, and Tools. Addison-Wesley. ISBN 0-201-10088-6.

Akiki Pierre, A, Bandara Arosha, K and Yu, Y (2016). Engineering Adaptive Model-Driven User Interfaces. *IEEE Transactions on Software Engineering*, **42**(12):1118-1147.

Atkinson, C. et Kühne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41.

Balme, L, Demeure, A, Barralon, N, Coutaz, J and Calvary, G (2004). CAMELEON-RT: A software architecture reference model for distributed, migratable, and plastic user interfaces. *2nd European Symposium on Ambient Intelligence (EUSAI 2004)*, Eindhoven, The Netherlands, Lecture Notes in Computer Science, 3295, Springer, Berlin, Heidelberg.

Bastide, R, Palanque, P, Le, D and Munoz, J (1998). Integrating rendering specifications into a formalism for the design of interactive systems. *5th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (Dsv-is'98)*, Abingdon, UK, Springer Verlag.

Bastien, J and Scapin, D (1973). Ergonomic Criteria for the Evaluation of Human-Computer Interfaces*. Technical report INRIA*, N°156.

Bellik, Y and Teil, D (1992). Multimodal Dialog Interface. *International Conference on Work with Display Units (WWDU'92).* Berlin, Germany, Luczak, H, Cakir, A & G (Eds).

Bellik, Y. (1995). Interface multimodales: concepts, modèles et architectures. *PhD thesis,* Paris 11 University, Paris, France.

Bellotti, V, Back, M, Edwards, K, Grinter, R, Henderson, A and Lopes, C. (2002). Making Sense of Sensing Systems: Five Questions for Designers and Researchers. *In Proceedings of CHI 2002 Conference*, 4(1): 415–422, ACM New York.

Bézivin, J. (2004). Model engineering for software modernization. *In Proceedings of the WCRE Conference*, page 4.

Bézivin, J. (2009). The three ages of mde. *In 2èmes Journées sur l'Interopérabilité des Applications d'Entreprise*.

Bézivin, J. et Gerbé, O. (2001). Towards a precise definition of the omg/mda framework. *In ASE, IEEE Computer Society*, pages 273–280.

Bézivin, J., Blay, M., Bouzhegoub, M., Estublier, J., Favre, J.-M., Gérard, S. et Jézéquel., J. M. (2005). Rapport de synthèse du cnrs sur le mda (model driven architecture). *Rapport technique*, CNRS.

Blanc, X. (2005). MDA en action. *Eyrolles*.

Bolt, RA (1980). Put-that-there: Voice and gesture at the graphics interface. *7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'80)*, Seattle, Washington, USA, ACM Press.

Bouchet, J. (2006). Ingénierie de l'interaction multimodale en entrée : Approche à composants ICARE. *Thèse de doctorat*, Université de Grenoble, France.

Bourguet , ML and Caelen, J (1992). Interfaces Homme-Machine Multimodales : Gestion des Evénements et Représentation des Informations. *Conférence Ergonomie et Informatique Avancée (ERGO-IA'92)*, Biarritz, France.

Calvary, G. (2007). Plasticité des Interfaces Homme-Machine. *Thèse d'Habilitation à diriger des Recherches*. Université Joseph Fourier – Grenoble I.

Calvary, G, Coutaz, J, Thevenin, D, Limbourg, Q, Bouillon, L and Vanderdonckt, J (2003). A unifying reference framework for multi-target user interfaces. *Journal of Interacting with Computers (JIC)*, 15(3):289-308.

Clerckx, T, Winters, F and Coninx, K (2005). Tool support for designing context-sensitive user interfaces using a model-based approach. *4th international workshop on Task models and diagrams (TAMODIA'05)*, Gdansk, Poland, ACM, New York, NY, USA.

Cockton, G (2005). A Development Framework for Value-Centred Design. *Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*, Portland, OR, USA, ACM Press.

Cohen, PR, Dalrymple, M, Moran, DB, Pereira, F C and Sullivan, J W (1989). Synergistic use of direct manipulation and natural language. *Conference on Human factors in computing systems (SIGCHI'89)*. Austin, Texas, USA, ACM Press.

Coutaz, J. (2006). Meta-User Interfaces for Ambient Spaces, *Interational Workshop on Task Model and Diagram (TAMODIA'06)*, Bucarest, Romania, 2006

Coutaz, J. (2010). User Interface Plasticity: Model Driven Engineering to the Limit!. *ACM, Engineering Interactive Computing Systems (EICS 2010) International Conference*. Keynote paper, pages 1-8.

Coutaz, J. and Nigay, L. (1994). Les propriétés CARE dans les interfaces multimodales. *In Actes de la conférence IHM'94*, Lille. pages 7-14.

Coutaz, J. and Nigay, L. (2012). Multimodalité et plasticité des interfaces homme-machine en informatique ambiante : concepts et espaces de conception. Information Interaction Intelligence - Le point sur le i (3), *Cépaduès Editions*, pp page 179-214.

Cuenca Lucero, F, Van den Bergh, J, Luyten, K and Coninx, K (2015*) Hasselt UIMS: a tool for describing multimodal interactions with composite events. 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, Germany, ACM Press.

Cuenca Lucero, F, Van den Bergh, J, Luyten, K and Coninx, K (2016) Hasselt: Rapid Prototyping of Multimodal Interactions with Composite Event-Driven Programming. *International Journal of People-Oriented Programming* (IJPOP), 5(1): 19-38.

Cuenca, F, et al (2014). A domain-specific textual language for rapid prototyping of multimodal interactive systems. *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems*. ACM.

Czarnecki, K. et Helsen, S. (2003). Classification of model transformation approaches. *In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture.*

Czarnecki, K. et Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. Journal*, 45(3):621–645.

Davis, J. (2003). Gme: the generic modeling environment. *In OOPSLA Companion*, pages 82-83.

Diaw, S., Lbath, R. et Coulette, B. (2010). Etat de l'art sur le développement logiciel basé sur les transformations de modèles. *Technique et Science Informatiques*, 29(4-5):505–536.

Dibon, P, Dalmau, M and Roose P (2013). Ubiquitous widgets: Designing interactions architecture for adaptive mobile applications. *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*. IEEE.

Duarte, C and Carrico, L (2006). A conceptual framework for developing adaptive multimodal applications. *11th International Conference on Intelligent User Interfaces (IUI'06)*. Sydney, Australia, ACM Press.

Dumas, B (2010). Frameworks, description languages and fusion engines for multimodal interactive systems. *PhD thesis,* University of Fribourg, Switzerland.

Dumas B, Lalanne D, Ingold R. (2010). Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. Journal of Multimodal User Interfaces, 3(3):237-247.

Dumas, B, Lalanne, D and Oviatt, S (2009). Multimodal interfaces: A survey of principles, models and frameworks. Human Machine Interaction, *Lecture Notes in Computer Science*, 5440:3-26, Springer-Verlag, Berlin, Heidelberg.

Dumas, B, Signer, B and Lalanne D. (2014) A graphical editor for the SMUIML multimodal user interaction description language. *Science of Computer Programming*, 86:30-42.

Elouali N, Le Pallec X, Rouillard J, Tarby JC. A model-based approach for engineering multimodal mobile interactions. *In: 12th International Conference on advances in mobile computing and multimedia (MoMM 2014*). Taiwan: ACM Press; 2014.

Elouali, N, Tarby, JC, Le Pallec, X and Rouillard, J (2012). Approche IDM pour le développement d'applications mobiles multimodales. *9th edition of the conference MAnifestation of Young Researchers in Information and Communication Sciences and Technologies (MajecSTIC'12)*, Lille, France.

Favre, J.-M. (2004). Foundations of meta-pyramids : Languages vs. metamodels - episode ii : Story of thotus the baboon. *In Language Engineering for Model-Driven Software Development.*

Gamma, E., Helm, R., Johnson, R. E. et Vlissides, J. M. (1993). Design patterns : Abstraction and reuse of object-oriented design. *In ECOOP*, pages 406-431.

Gauffre, G. (2007). Domain Specific Methods and Tools for the Design of Advanced *Interactive Techniques MDDUI 2007 @Models Conference*.

Gourdol, A, Nigay, L, Salber, S and Coutaz, J (1992). *Two case Studies of Software Architecture for Multimodal Interactive Systems: VoicePaint and a Voice-enabled Graphical Notebook*. Conference on Engineering for Human-Computer Interaction (EHCI'92). Ellivouri, Finland, IFIP Transactions Series.

Greenfield, J. et Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. *In OOPSLA Companion*, pages 16-27.

Guedes, Á, Azevedo, RGdA and Barbosa, SDJ (2017). Extending multimedia languages to support multimodal user interactions. *Multimedia Tools and Applications*, **76**(4), 5691-5720.

Hariri, A, Tabary, D and Kolski, C. (2006). Démarche en vue de la Génération d'Interfaces Mobiles et Plastiques. Actes *de la troisième conférence UBIMOB2006 Mobilité et Ubiquité*, CNAM, Paris, 5-8 septembre.

Hesenius, M, Griebe, T and Gruhn V (2014). Towards a behavior-oriented specification and testing language for multimodal applications. *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'14,* ACM Press.

ISO. (1996). Information technology - syntactic metalanguage - extended bnf. Technical report*, International Organization for Standardization*, 1996.

Jacquet, C. (2006). Présentation oppotuniste et multimodale d'informations dans le cadre de l'intelligence ambiante. *Thèse de doctorat*, Université Paris XI, France.

Jouault, F. et Bézivin, J. (2006). Km3 : A dsl for metamodel specification*. In FMOODS*, pages 171–185.

Jouault, F. et Kurtev, I. (2006). On the architectural alignment of atl and qvt. *In Proceedings of the 2006 ACM symposium on Applied computing, SAC '06*, pages 1188-1195, New York, NY, USA. ACM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. et Irwin, J. (1997). Aspect-oriented programming. *In ECOOP*, pages 220-242.

Kleppe, A. G., Warmer, J. et Bast, W. (2003). MDA Explained : The Model Driven Architecture : Practice and Promise. *ADDISON-WESLEY*.

Krasner, GE and Pope, ST (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Program (JOOP)*, **1**(3), 26-49.

Kurtev, I., Bézivin, J. et Aksit, M. (2002). Technological spaces : An initial appraisal. *In : International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences,* Industrial Track.

Laforcade, P. (2004). Méta-modélisation UML pour la conception et la mise en œuvre de situations-problèmes coopératives. *Thèse de Doctorat*, Université de Pau et des Pays de l'Adour.

Lamia Gaouar, Abdelkrim Benamar, Olivier Le Goaer and Frédérique Biennier. (2018). HCIDL: Human-Computer Interface Description Language for multi-target, multimodal, plastic user interfaces. Future Computing and Informatics Journal. DOI: 3. 10.1016/j.fcij.2018.02.001.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. et Lopez-Jaquero, V. (2004). Usixml: a language supporting multi-path development of user interfaces. *Springer-Verlag*. Pages 11-13.

Martin, JC. (1994). *Cadre d'étude de la multimodalité fondé sur les types et buts de coopération entre modalités*. *Journées internationales L'interface des mondes réels & virtuels*, Montpellier, France

Martin, JC (1999). Tycoon: Six primitive types of cooperation for observing, evaluating and specifying cooperations. *Symposium on Psychological Models of Communication in Collaborative Systems*, North Falmouth, Massachusetts, USA, AAAI Press.

Meixner, G., Paterno, F., Vanderdonckt, J. (2011). Past, present, and future of model-based user interface development. I-com 10(3):2-11.

Mellor, S. J., Clark, A. N. et Futagami, T. (2003). Guest editors' introduction : Model-driven development. IEEE Software, 20:14–18.

Mens, T. et Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes In Theorical Computer Science*, 152:125–142.

Michotte, B. and Vanderdonckt, J. (2008). GrafiXML, a Multi-target User Interface Builder Based on UsiXML. *Fourth International Conference on Autonomic and Autonomous Systems* (ICAS'08).

Mignot, C, Valot, C and Carbonell, N (1993). An Experimental Study of Future 'Natural' Multimodal Human-Computer Interaction. *Conference on Human Factors in Computing Systems* (SIGCHI'93), Amsterdam, The Netherlands, ACM Press.

Miller, J. et Mukerji, J. (2003). Model driven architecture (mda) 1.0.1 guide. *Rapport technique, OMG*.

Minsky, M. L. (1969). Semantic Information Processing. *The MIT Press.*

Muller, A. (2006). De la modélisation objet des logiciels à la métamodélisation des langages informatiques. *Thèse de doctorat*, Université de Rennes 1.

Myers, B.A., Hudson, S.E., Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*. 7(1): 3-28.

Nigay, L and Coutaz, J (1993). A design space for multimodal systems: Concurrent processing and data fusion. *Conference on Human Factors in Computing Systems (SIGCHI'93*), Amsterdam, The Netherlands, ACM Press.

Nigay, L and Coutaz, J (1997). Multifeature systems: The CARE properties and their impact on software design. *Intelligence and Multimodality in Multimedia Interfaces: Research and Applications*, AAAI Press.

Normand, V. (1996). Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateur à leur réalisation. *Thèse de l'Université Joseph Fourier-Grenoble* I.

Object Management Group (2006). Meta Object Facility (MOF) Core Specification Version 2.0.

Object Management Group, OMG. (2008). Model Driven Architecture, MDA Guide Version 1.0.1.

Object Management Group. (2008). Mof model to text transformation language 1.0.

Object Management Group. (2011). OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.4.1.

Object Management Group. (2016). Mof query/view/transformation specification version 1.3.

Paternò, F., Mancini, C. et Meniconi, S. (1997). ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. *IFIP TC13 Interantional Conference on Human-Computer Interaction (INTERACT '97)*, London, UK, UK, pages 362-369.

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

Rousseau, C (2006). Présentation multimodale et contextuelle de l'information. *PhD thesis, Paris 11 University, Paris, France.*

Schüssel, F, Honold, F and Weber, M (2012). *Using the transferable belief model for multimodal input fusion in companion systems*. IAPR Workshop on Multimodal Pattern Recognition of Social Signals in Human-Computer Interaction. Springer, Berlin, Heidelberg.

Seidewitz, E. (2003). What models mean. *IEEE Software*, 20(5):26–32.

Soley, R. (2000). Model driven architecture (mda), draft 3.2. *Rapport technique, Object Management Group, Inc.*

Sottet, J.S. (2008). Méga-IHM : malléabilité des Interfaces Homme-Machine dirigées par les modèles. *Thèse de doctorat Informatique.* Laboratoire d'Informatique de Grenoble (LIG), Université Joseph Fourier.

Sottet, J.S, Calvary,G, Favre, J.M and Coutaz, J. (2007). Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: Mega-UI, CHISE book II, Seffah, A. & Vanderdonckt, J. (Eds), 2007.

Stanciulescu, A. (2008). A Methodology for Developing Multimodal User Interfaces of Information System. *Thèse de doctorat*. Université catholique de Louvain, Louvain-la-Neuve, Belgium.

Szekely, P. (1996). Retrospective and Challenges for Model-Based Interface Development. *In Proceedings of Computer-Aided Design of User Interfaces, CADUI'96, J. Vanderdonckt (eds)*, Presses Universitaires de Namur.

Sztipanovits, J., Glossner, C. J., Mudge, T. N., Rowen, C., Sangiovanni-Vincentelli, A. L., Wolf, W. et Zhao, F. (2005). *Grand challenges in embedded systems.* In CODES+ISSS.

Thevenin, D (2001). L'adaptation en Interaction Homme-Machine: le cas de la plasticité. *PhD thesis, Joseph Fourier University, Grenoble, France*.

Thevenin, D and Coutaz, J (1999). *Plasticity of User Interfaces: Framework and Research Agenda*. Human-computer Interaction conference, Edinburgh, Scotland, A. Sasse & C. Johnson (Ed), IFIP IOS Press.

Thevenin, D, Coutaz, J and Calvary, G (2003). A reference framework for the development of plastic user interfaces. *Multi-Device and Multi-Context User Interfaces: Engineering and Applications Frameworks.* Wiley Publ, Javahery Eds.

Touzi, J. (2007). Aide à la conception de Système d'Information Collaboratif support de l'interopérabilité des entreprises. *Thèse de doctorat*, Centre de Génie Industriel – Ecole des Mines d'Albi Carmaux.

Vanderdonckt, J, Limbourg, Q, Michotte, B, Bouillon, L, Trevisan, D and Florins, M (2004). *USIXML: A user interface description language for specifying multimodal user interfaces*. W3C Workshop on Multimodal Interaction (WMI 2004), Sophia Antipolis, France.

Vernier, F(2001). La multimodalité en sortie et son application à la visualisation de grandes quantités d'information. *PhD thesis, Joseph Fourier University, Grenoble, France*.

Warmer, J. (2002). The role of ocl in the model driven architecture. In FIDJI.

Weimer, D and Ganapathy, SK (1989). A synthetic visual environment with hand gesturing and voice input. *Conference on Human factors in computing systems (SIGCHI'89),* Austin, Texas, USA, ACM Press.

Yang, J, Stiefelhagen, R, Meier, U and Waibel, A (1998). Visual tracking for multimodal human computer interaction. *Conference on Human Factors in Computing Systems: Making the Impossible Possible (SIGCHI'89),* Los Angeles, California, ACM Press and Addison-Wesley Publishing Co.

CONTRIBUTIONS SCIENTIFIQUES

Djamila Benhaddouche, Lamia Gaouar. (2009). Mise en Place d'un Processus d'Extraction de Connaissances en milieu Hospitalier. JDTIC'09 Les 1ères Journées Doctorales en Technologie de l'Information et de la Communication. 16-18 Juillet 2009, Rabat Maroc.

Lamia Gaouar, Abdelkrim Benamar, Fethi Tarik Bendimerad. (2014). *Requirements of cross platform mobile development tools*. Conference: International conference on Information Technology for Organization Development (IT4OD 2014), pp 9-15,, At Tebessa, Algeria, Volume: pp 9-15.

Lamia Gaouar, Abdelkrim Benamar, Fethi Tarik Bendimerad. (2015). Model Driven Approaches to Cross Platform Mobile Development. In Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication (IPAC '15), Djallel Eddine Boubiche, Faouzi Hidoussi, and Homero Toral Cruz (Eds.). ACM, New York, NY, USA, , Article 19 , 5 pages. DOI: http://dx.doi.org/10.1145/2816839.2816882

Lamia Gaouar, Abdelkrim Benamar, Olivier Le Goaer and Frédérique Biennier. (2018). HCIDL: Human-Computer Interface Description Language for multi-target, multimodal, plastic user interfaces. Future Computing and Informatics Journal. DOI: 3. 10.1016/j.fcij.2018.02.001.

RÉSUMÉ

Dans le domaine de l'interaction homme-machine, les défis à relever sont liés à la prise en compte de multiples nouvelles interactions et à la diversité des dispositifs. Notre travail s'inscrit dans le domaine des langages de description de l'interface utilisateur. Cette thèse présente HCIDL, un langage de modélisation mis en scène dans une approche MDA. Parmi les propriétés liées à l'interface homme-machine, notre proposition est destinée à la modélisation d'interfaces d'interaction plastiques, multi-cibles et multimodales. En combinant la plasticité et la multimodalité, HCIDL améliore la convivialité des interfaces utilisateur grâce à un comportement adaptatif en fournissant aux utilisateurs finaux un ensemble d'interaction adapté aux entrées/sorties des terminaux et une disposition optimale.

MOTS CLÉS

Ingénierie dirigée par les modèles; Interface Homme-Machine, Langage de description d'interfaces utilisateur; Applications multimodales; Interfaces utilisateur plastiques.

ABSTRACT

From the human-computer interface perspectives, the challenges to be faced are related to the consideration of new, multiple interactions, and the diversity of devices. Our work is part of the field of user interface description languages. This thesis presents HCIDL, a modelling language staged in a model-driven engineering approach. Among the properties related to human-computer interface, our proposition is intended for modelling multi-target, multimodal, plastic interaction interfaces. By combining plasticity and multimodality, HCIDL improves usability of user interfaces through adaptive behaviour by providing end-users with an interaction-set adapted to input/output of terminals and, an optimum layout.

KEYWORDS

Model Driven Engineering; Human-Computer Interface; User Interface Description Languages; Multimodal Applications; Plastic User Interfaces.

ملخص

في مجال التفاعل بين الإنسان والآلة ، ترتبط التحديات بالنظر في العديد من التفاعلات الجديدة وتنوع الأجهزة. يعد عملنا جزءًا من لغة وصف واجهة المستخدم. تقدم هذه الأطروحة (HCIDL) ، وهي لغة نمذجة نظمت في مقاربة MDA. من بين الخصائص المتعلقة بالواجهة بين الإنسان والآلة ، يقصد اقتراحنا لنمذجة واجهات التفاعل البلاستيكية ، متعددة الأهداف ومتعددة الوسائط. من خلال الجمع بين اللدونة ومتعدد الوسائط ، يحسِّن HCIDL قابلية استخدام واجهات المستخدم من خلال السلوك التكيفي من خلال تزويد المستخدمين النهائيين بمجموعة مناسبة من التفاعل من أجل الإدخال / الإخراج الطرفي والتخطيط الأمثل.

الكلمات الرئيسية

الهندسة المدعمة بالنموذج ;لغة وصف واجهة المستخدم ;واجهة آلة الإنسان ;تطبيقات متعددة الوسائط ;واجهات المستخدم البلاستيكية.