UNIVERSITE JOSEPH FOURIER – GRENOBLE 1 UFR D'INFORMATIQUE ET DE MATHEMATIQUES APPLIQUEES

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

Présentée et soutenue publiquement

par

Jean-Sébastien Sottet

Le 10 Octobre 2008

Titre :

Mega-IHM : Malléabilité des Interfaces Homme-Machine Dirigée par les Modèles

Directeurs de thèse :

Gaëlle Calvary & Jean-Marie Favre

JURY:

M. Jean Bézivin, Rapporteur

M. Franck Tarpin-Bernard, Rapporteur

M. Hervé Martin, Président

Mme. Mireille Blay-Fornarino, Examinatrice

M. Marco Winckler, Examinateur

M. Jean Vanderdonckt, Examinateur

Table des matières

Chapitre I : Introduction

Les systèmes informatiques actuels offrent progressivement plus de possibilités aux utilisateurs. Internet sur téléphone mobile en est un exemple : l'information est accessible partout, tout le temps, pour une majorité d'usagers dont les capacités et les degrés d'expertise sont extrêmement variés. La course aux dispositifs bat son plein : les téléphones mobiles deviennent de vrais micro-ordinateurs ; les téléviseurs, chaînes-hifi, tableaux de bord voient leurs capacités augmenter et s'enrichissent de nouvelles fonctionnalités.

Dans ce cadre, et avec l'essor du web 2.0, l'information devient personnalisable : on ne consulte pas uniquement des pages, on peut interagir avec elles, choisir d'y intégrer de nouveaux composants, les configurer, les composer, etc. Cet ensemble de choix et cette combinatoire autour de la personnalisation posent de nombreuses problématiques. Combiné à l'informatique ambiante, le problème s'amplifie de manière vertigineuse. L'informatique d'aujourd'hui n'a plus rien à voir avec un programme figé destiné à un utilisateur donné et fonctionnant sur un type d'ordinateur prédéfini. C'est tout le contraire. Pour que ces systèmes restent utilisables, malgré leur complexité croissante, les *Interfaces Homme-Machine* (IHM) doivent subir des mutations profondes et répondre à de nouvelles exigences.

Cette thèse propose de nouvelles techniques destinées, d'une part aux *utilisateurs* et d'autre part aux *ingénieurs :* les utilisateurs pourront personnaliser les IHM qu'ils utilisent, les ingénieurs pourront les adapter pour qu'elles puissent s'exécuter dans différents contextes. Il s'agit sensiblement de la même opération, même si les compétences des utilisateurs et des ingénieurs ne sont pas les mêmes…

Dans cette introduction nous définissons tout d'abord les différents contextes auxquels nous nous intéressons : les contextes de conception d'une part et les contextes d'utilisation d'autre part. Ensuite nous décrivons les besoins induits par l'informatique ambiante par rapport aux contextes. Nous définissons ainsi le besoin de « malléabilité » des systèmes informatiques pour la « plasticité ». Finalement, nous présentons brièvement l'approche dirigée par les modèles ainsi que les objectifs que nous nous sommes fixés dans le cadre de cette thèse.

1 IHM Innovantes (IHMI) vs. Ingénierie des IHM (IIHM)

Le domaine de recherche traitant de l'étude des IHM est vaste. En fait, on peut considérer cette thématique selon au moins deux perspectives qu'il convient de bien distinguer pour éviter toute confusion. Dans le domaine de l'IHM deux types de contributions sont possibles :

- **IHM Innovantes (IHMI)**. L'apparition de nouveaux dispositifs permet d'imaginer des interactions innovantes basées, par exemple, sur la réalité mixte, la multi-modalité, etc. La combinaison de multiples modalités, dispositifs et techniques d'interaction, donne lieu à de nouvelles possibilités qu'il s'agit d'explorer. C'est un thème de recherche particulièrement prometteur. Dans ce type d'approche, c'est la partie « visible » des interfaces que l'on cherche à améliorer et l'utilisateur est au centre d'absolument toutes les discussions. Evaluer ce genre de travail consiste à mesurer la satisfaction de l'utilisateur face à une IHM et un problème à résoudre. De nombreux travaux de recherche suivent actuellement cette approche définitivement et résolument centrée utilisateur. Cette branche de l'IHM est d'une certaine manière le « noyau dur ». Historiquement, il a fallu faire reconnaître l'importance de l'utilisateur et des IHM dans le monde informatique, très largement centré systèmes informatiques. Cela a prit du temps.

- **Ingénierie des IHM (IIHM)**. L'importance de la recherche d'IHM innovantes ne doit pas faire oublier que les IHM plus « classiques » ont fait leurs preuves dans l'industrie et qu'il s'agit là d'un fantastique acquis. Après tout, c'est sur les IHM de type « WIMP » (Windows, Icons, Menus, Pointers) que sont basées la plupart des applications existantes. Celles-ci ont joué un immense rôle dans la diffusion de l'informatique auprès du public. Les techniques d'interactions, qu'elles soient innovantes ou non, ne constituent que la partie visible de l'iceberg. Le logiciel réalisant l'IHM forme la partie cachée. La complexité de cet élément logiciel va sans cesse croissante et la gestion de cette complexité est peu à peu apparue comme étant une problématique à part entière en IHM. Dans cette deuxième approche il s'agit de *faciliter l'Ingénierie des IHM*, et ce indépendamment de leur degré d'innovation. Autrement dit, il s'agit de *diminuer le coût* de production d'interfaces (innovantes ou « classiques ») tout en préservant leurs qualités et leurs fonctionnalités. Ici on ne considère pas seulement l'utilisateur, mais également le concepteur d'IHM. Autrement dit la facilité de conception doit être prise en compte ; les coûts de production constituent un problème important tout comme la qualité du résultat.

Bien évidemment ces deux approches sont tout à fait complémentaires. Même si en principe, elles devraient être intégrées harmonieusement dans le domaine plus général de l'IHM, en pratique, IHMI

et IIHM sont souvent étudiées de manière relativement séparées car elles correspondent à première vue à des objectifs distincts et des valeurs différentes.

Cette distinction étant faite, c'est la deuxième approche que nous suivons dans cette thèse.

Cette thèse se concentre sur l'Ingénierie des Interfaces Homme-Machine (IIHM).

Nous considérons les concepteurs des IHM comme aussi importants que les utilisateurs. Cette approche fait du sens notamment dans les domaines d'applications qui nécessitent la production d'IHM à faible coût. Considérons le domaine de la gestion du contenu (Content Management Systems - CMS en anglais). Ce domaine est en pleine expansion actuellement, notamment avec l'avènement du Web 2.0. Dans ce domaine d'application les « concepteurs » ont des compétences limitées en informatique. Le fait qu'ils puissent facilement créer des applications et des IHM donne lieu au succès même de l'approche. Les IHM générées ne sont pas particulièrement innovantes, leur ergonomie n'est peut-être pas parfaite, mais ce n'est pas le problème principal. Il s'agit tout au contraire de démocratiser la production d'IHM afin de répondre à des demandes de plus en plus nombreuses. Ainsi, selon les domaines d'applications considérés, la balance entre « favoriser l'utilisateur » et « favoriser le concepteur » peut changer.

Mais *quid* alors de l'ingénierie de tels systèmes complexes ? Comment faire face, dans le contexte de l'informatique ambiante, à une telle montée en puissance, en diversité et donc en combinatoire ? L'ingénierie de tels systèmes est un vrai défi. Non seulement il s'agit de combiner de plus en plus de technologies, mais il faut s'assurer que malgré cette complexité accrue les systèmes restent simples à développer et à utiliser. Le développement de tels systèmes n'est possible qu'en combinant expertises de multiples intervenants. De ce point de vue, il s'agit avant tout d'un problème d'ingénierie.

2 Contexte d'utilisation & Contexte d'ingénierie

La dualité entre utilisateurs et ingénieurs (d'IHM) va se retrouver tout au long de cette thèse. Les premiers *utilisent* les IHM, les seconds les conçoivent et les développent. De même on parle respectivement de contexte d'utilisation et de contexte d'ingénierie. Nous parlons *d'acteurs* pour désigner l'ensemble des personnes influant et agissant sur le logiciel : de *l'ingénieur* (ou concepteur) à *l'utilisateur final*.

Avec l'informatique ambiante, les contextes d'utilisation se diversifient.

On appelle *contexte d'utilisation* le triplet :

- *utilisateur* (néophyte, expert, fatigué, attentif, etc.)
- *environnement d'utilisation* (rue, bureau, maison, transports en commun, etc.)
- *plate-forme d'interaction* (PC, PDA, téléphone, console de jeu, montre, etc.)

La variété des contextes d'utilisation, en cardinalité et imagination requise, commence à être difficilement maîtrisable à la conception. De plus, la multiplicité des technologies qu'il faut maîtriser et mettre en œuvre pour réaliser des systèmes informatiques, et en particulier des IHM, pose le problème du contexte d'ingénierie.

On appelle *contexte d'ingénierie* le triplet :

- *ingénieur* (spécialiste métier, spécialiste IHM, etc.),
- *environnement d'ingénierie* (équipe de développement, entreprise, travail individuel, réunion de travail, etc.),
- *plate-forme d'ingénierie* (UML avec Rose, UML sur tableau blanc, outils de conception, de développement, Visual Studio, Eclipse, etc.)

Le besoin de considérer les deux *contextes d'utilisation* et *d'ingénierie* est d'autant plus criant en informatique ambiante. Cette nouvelle problématique joue un rôle important dans la motivation de nos travaux. Nous en présentons donc ci-dessous les principales caractéristiques.

3 Problématique : informatique ambiante

L'informatique ambiante promet un accès ubiquitaire à l'information. Se posent alors des problèmes de complexité et d'échelle : du très petit (*capteur*) au très grand (systèmes combinant des milliers voire des millions de capteurs). Mais on constate également un besoin de dynamicité du traitement de l'information et en particulier d'une réaction éventuelle aux stimuli captés au niveau du monde environnant (contexte d'utilisation).

Cette partie est en deux volets, le premier présente la complexité technologique croissante des systèmes en informatique ambiante. Le deuxième volet explique le besoin d'avoir des systèmes toujours plus réactifs.

3.1 Hétérogénéité et complexité technologique

Si l'on veut préserver un système simple et souple pour *l'utilisateur* (c'est-à-dire en lui laissant l'opportunité de choisir quelles plates-formes utiliser quel que soit l'environnement), la complexité est reportée sur *l'ingénieur*. Ainsi le défi de l'ingénieur est de masquer la complexité à l'utilisateur.

C'est pour cela que dans le cadre de cette thèse nous portons autant d'attention au concepteur d'IHM qu'à l'utilisateur.

Plusieurs technologies et architectures ont émergé autour des problématiques induites par l'informatique ambiante : charge de réseaux, répartition des calculs, miniaturisation, nombreuses sources de capteurs. Lorsqu'il faut développer de tels systèmes, outre les aspects d'échelle, beaucoup de connaissances techniques et technologiques sont à mettre en œuvre. C'est le *contexte d'ingénierie*, en particulier les *plates-formes de conception* qui sont alors variées et variables.

Aujourd'hui, mettre en œuvre un système pour qu'il réponde aux exigences de l'informatique ambiante est une tâche complexe. En effet pour réaliser de tels systèmes, on s'appuie sur bon nombre de *spécialités d'ingénierie* allant dans le sens de la diversification des *langages* informatiques [SLE08, 2008], [Favre, et al., 2007]. Le temps où le développement d'un logiciel se faisait en un seul langage et impliquait un seul programmeur est révolu. Désormais, il faut faire face au développement global et collaboratif des logiciels, contexte dans lequel l'hétérogénéité des technologies et des langages est particulièrement importante. Nous allons aborder de manière explicite et originale cet aspect dans le cadre de thèse.

3.2 Dynamicité

Au début de l'informatique, un programme était réalisé par un ingénieur pour un contexte prévu, pour un besoin déterminé et pour une machine donnée. C'était de l'*ingénierie sur mesure*.

Plus récemment, dans le cadre du génie logiciel, l'approche dite *ligne de produits* a été proposée. Une *ligne de produits* offre, pour de multiples contextes prévus par l'ingénieur (téléphone, PC, PDA,…), différentes *variantes* de l'application. C'est lors du processus de conception que le choix de la variante se fait selon les besoins et le contexte prévu d'utilisation. La sélection de la variante appropriée est faite donc de manière « statique » par un intervenant autre que l'utilisateur. Ainsi l'approche *ligne de produit* n'est pas compatible avec la dynamicité requise lors de l'utilisation du produit logiciel : elle ne permet pas de répondre aux changements opportunistes que pourrait décider l'utilisateur.

Avec l'informatique ambiante, le contexte d'usage est changeant à *l'utilisation* tout comme lors de *l'ingénierie*. La dynamicité, c'est-à-dire les modifications d'une IHM nécessaires, lors de l'utilisation d'un système informatique, est une caractéristique importante que l'on considère dans cette thèse. On pourrait imaginer une solution dans laquelle l'utilisateur peut tout réaliser sans contrôle, mais aussi sans aide. Nous pensons au contraire, que bien que, la dynamicité doive être prise en compte, il ne faut pas pour autant perdre l'intérêt d'une conception dirigée par des spécialistes du logiciel et

des IHM. C'est en unifiant les concepts de l'ingénierie et de l'utilisation et en les adaptant à chacun des acteurs qu'on se propose de faire face aux problématiques soulevées par l'informatique ambiante.

4 Adaptabilité, Malléabilité et Plasticité

Se donner les moyens d'adapter les systèmes interactifs implique, comme nous l'avons vu, une charge importante pour l'ingénieur. Il ne s'agit ni de réaliser des produits sur mesure pour un utilisateur donné, ni de développer des lignes de produits classiques. Il s'agit de rendre les produits « malléables », de sorte que l'utilisateur puisse lui-même adapter le produit au contexte d'utilisation. Plus particulièrement nous définissons ci-dessous ce qu'on entend par « système adaptable », « système malléable » et dans le contexte des IHM, les « IHM plastiques ». Dans le cadre de cette thèse sur l'IIHM la malléabilité est vue comme un moyen de mettre en œuvre la plasticité.

4.1 Système adaptable

Comme nous l'avons introduit, cette thèse parle de systèmes adaptables. Nous utilisons une définition inspirée de cette notion en Génie Logiciel (GL).

Définition : Un ***système adaptable*** est un système pouvant être « modifié » **lors** de son utilisation.

L'adaptabilité n'est pas un concept propre à l'informatique. A titre d'illustration, considérons comme systèmes les « vélos » ; un vélo pour un (jeune) enfant ne possède qu'une seule vitesse et par conséquent ne peut pas s'adapter à la fatigue de l'enfant ou au terrain lors de son utilisation. Ici on suppose que la fonctionnalité du vélo est de permettre à l'utilisateur de se rendre d'un point donné à un autre. C'est ce que permet le vélo d'enfant, bien que celui-ci ne soit pas "adaptable".

En revanche la plupart des vélos pour adultes possèdent des vitesses qui rendent la motricité adaptable. De même, certaines hauteurs de selle peuvent être modifiées pour baisser rapidement le centre de gravité lors d'une descente en VTT. Cette modification du système ne faisant pas partie des fonctionnalités de base du vélo, on parle d'adaptation.

Dans la définition de l'adaptation, nous insistons sur l'aspect dynamique, c'est-à-dire au moment de l'utilisation. Par la suite, nous affinons les moyens et les objectifs (comment, quoi et pourquoi) qui nous intéressent pour assurer l'adaptation des systèmes interactifs.

4.2 Système malléable

Nous introduisons dans cette thèse le concept de malléabilité. Ce concept a une définition particulière en métallurgie que nous ne réutilisons pas ici. La malléabilité est pour nous un *moyen* de rendre un système adaptable.

Définition : Un ***système malléable*** est un système exposant des points de contrôle ou de variation permettant l'adaptation du système dans le respect de certaines plages de valeurs et de contraintes.

4.2.1 Points de malléabilité

Nous parlerons de **points de malléabilité** pour faire référence aux « points de contrôle ou de variation ». Par exemple, si l'on considère le pied d'un appareil photo : ces points de malléabilité sont représentés sur la figure 1 (par un ensemble de vis, rotules et poignées. Ils permettent et limitent (plages de valeurs/contraintes) la rotation et l'élévation de l'appareil. Le trépied peut ainsi être adapté au contexte d'usage.

Figure 1. Système « pied d'appareil photo » malléable.
Points de malléabilité encerclés.

Dans l'exemple du vélo, les points de malléabilité sont l'attache rapide pour régler la selle ainsi que le dérailleur pour les vitesses. C'est en effet par le dérailleur que l'on change les vitesses en sélectionnant pignons et plateaux différents. Le dérailleur a lui-même des « plages de valeurs limites » sans lesquelles la chaîne finira par sauter.

Figure 2. Dérailleur de vélo

4.2.2 Interfaces de malléabilité

Par rapport à l'adaptabilité, qui est la capacité d'un système à être modifié lors de son utilisation, la caractéristique de la malléabilité est le fait d'être basée explicitement sur des interfaces pour contrôler l'adaptation : nous les appelons des interfaces de malléabilité.

Définition : Une ***interface de malléabilité*** est un ensemble explicite de points de contrôle rendant le système malléable. Comme toute interface, il s'agit du sous-système permettant l'interaction avec un autre système (humain ou non).

La dernière partie concernant l'acteur interagissant avec l'interface de malléabilité requiert une discussion. Une interface de malléabilité pour un humain doit alors représenter les actions possibles sur le système malléable dans le respect des compétences de l'acteur.

Toujours dans notre exemple concernant le vélo, la poignée de vitesse qui contrôle le dérailleur est une IHM de malléabilité. Cette poignée sert aussi à l'utilisateur pour avoir un retour d'information sur sa vitesse (voir figure 3).

Figure 3. Poignée de vitesse grip-shift (VTT) :
les chiffres 0 à 9 indiquent la position de la chaîne sur le pignon (vitesse).

Il existe des interfaces systèmes/systèmes. Par exemple la fourche télescopique est une interface de malléabilité système à système (axes coulissant, ressorts, etc.) permettant d'adapter automatiquement (car mécaniquement) le vélo aux accidents du terrain. Le guidon quant à lui est une IHM « classique » pour diriger le vélo. En effet, la fonctionnalité intrinsèque d'un vélo est de pouvoir se rendre où l'on veut, ce n'est pas le changement des vitesses.

Notons que jusque là nous avons évoqué la malléabilité pour l'utilisateur. Mais ce concept est plus général car on peut l'envisager pour d'autres acteurs. Certaines interfaces de malléabilité pourraient être destinées à différents intervenants. Par exemple, une molette de réglage des freins peut être utilisée par le vendeur du vélo pour personnaliser le freinage. Le niveau d'expertise du vendeur est différent. Il est naturel d'avoir des interfaces dépendant des niveaux d'expertise.

4.2.3 Malléabilité des systèmes informatiques : une malléabilité augmentée

Les exemples ci-dessus correspondent tous à des systèmes physiques et donnent une idée intuitive mais partielle de ce que recouvre le terme de malléabilité de cette thèse. En informatique, la malléabilité traduit les capacités d'un logiciel à s'adapter selon les axes des contextes d'*ingénierie et d'utilisation*. Ces axes sont rendus explicites à l'acteur qui les manipule (utilisateur, ingénieur ou le système lui-même) pour configurer le logiciel à la fois en conception mais aussi à l'exécution.

En informatique, les points de malléabilité du logiciel ont une représentation, ce sont les *modèles*. C'est ici qu'intervient l'Ingénierie Dirigée par les Modèles, technique importante en Génie Logiciel, comme on le voit par la suite. Dans les systèmes physiques comme les vélos, les représentations issues de la conception, autrement dit les différents plans et modèles du vélo sont d'une nature bien différente du système, car totalement basées sur le concept d'information, ce qui n'est pas le cas pour le vélo. Au contraire en informatique, les modèles sont de la même nature intrinsèque que le système qu'ils modélisent. Par conséquent on peut reporter l'adaptation du modèle sur le système et

inversement. Les modèles sont ainsi *actifs* dans la mesure où ils permettront d'adapter le système dynamiquement, et inversement ils agissent comme des vues sur le système si celui-ci se modifie. Dans le monde physique c'est comme si en modifiant les plans du vélo on pouvait modifier le vélo et inversement en modifiant le vélo on modifiait les plans !

Contrairement aux systèmes physiques, la malléabilité des systèmes informatiques est donc augmentée, car la malléabilité d'un logiciel est permise par la représentation et la manipulation des savoirs et savoir-faire de ce même logiciel. C'est comme si l'on fournissait des vélos avec l'ensemble des plans de conception et de fabrication : cet aspect est abordé plus en détail par la suite.

4.3 Adaptation contrôlée et IHM plastiques

L'adaptabilité est la capacité d'un système à être modifié ; la malléabilité est un moyen permettant l'adaptabilité ; mais pourquoi adapter ? Dans quelles conditions ? Comme nous allons le voir se pose alors le problème de « l'adaptation contrôlée » et de là découle la notion d'IHM plastique.

4.3.1 Adaptation contrôlée

Rendre les systèmes adaptables, par des points de malléabilité, ne suffit pas pour réaliser un système utile et utilisable. Il faut se poser la question : pourquoi rendre cette partie malléable ? À quoi s'adapter ? Dans quelles limites ? C'est ici qu'interviennent les notions de contexte et de qualité.

Dans le cas du cyclisme, le contexte est formé par de multiples aspects tels que la fatigue du cycliste (utilisateur), les montées et descentes, le fait de rouler de nuit ou de jour (environnement) ; avec des pneus gonflés ou non, etc. (plate-forme). L'adaptation au contexte se doit de préserver certains critères de qualité qu'il s'agit de déterminer : ce peut être un certain confort (fournir moins d'effort), un certain rendement (pouvoir aller plus vite), ou tout autre critère important du point de vue de l'utilisation du système.

L'adaptation peut être « manuelle », c'est-à-dire contrôlée par l'utilisateur. C'est le cas lorsque le cycliste monte sa selle grâce à un point de malléabilité prévu pour lui. L'adaptation peut aussi être automatisée et donc contrôlée par un système non humain. En effet sur certains vélos récents, il existe un appareil capable de changer de vitesse automatiquement (voir figure 4) en fonction du contexte. Si le cycliste est fatigué, en montée, on lui propose une vitesse « plus petite » donnant ainsi du couple et lui permettant de fournir un effort constant.

Figure 4. Dérailleur automatique Shimano :
prix ‘or’ 2008 de la meilleur innovation et design (IDEA)

4.3.2 IHM plastiques

Lorsque la *malléabilité* concerne un système interactif et garantit les propriétés ergonomiques dans sa manipulation, alors la malléabilité est un moyen d’assurer ce qui est appelé « plasticité ». Dans ce contexte, la malléabilité doit, par conséquent, apporter les moyens nécessaires à la *représentation et la manipulation* des propriétés spécifiques à l’IHM dans le but de pouvoir *adapter* ces IHM.

L'un de nos objectifs particuliers est de se focaliser sur le traitement de l’adaptation des IHM au *contexte d’utilisation* (plateforme, environnement, utilisateur). C'est cette notion qui est appelée plasticité.

Définition : la ***plasticité*** traite de l’adaptation des IHM à leur *contexte d’utilisation* dans le respect de certaines propriétés [Thevenin, 2001].

Les propriétés *d’utilisabilité* [Nielsen, 1994, Seffah, et al., 2004, Bastien, et al., 1993] sont des exemples de qualités attendues pour une IHM plastique. Elles sont la représentation « des plages de valeurs » (voir malléabilité) lorsqu’elles sont mesurables.

En outre, dans le cadre de notre thèse, le *contexte d’ingénierie* de la plasticité est lui-même pris en compte : il s’agit de fournir au concepteur des moyens efficaces de traiter et comprendre, dès les phases de conception, les enjeux spécifiques du domaine des IHM plastiques. Autrement dit, il est nécessaire de donner à l'ingénieur les outils pour modéliser certaines parties du contexte d'usage.

4.3.3 Le scénario « HHCS »

Pour illustrer la notion de plasticité, considérons un scénario, le scénario « HHCS » (Home Heating Control Service). Ce scénario est repris dans différents chapitres de cette thèse. Il montre l'évolution d'une IHM simple et « plastique » (figure 4).

Jean est dans le tramway lorsqu'il réalise avoir oublié de baisser le chauffage de sa maison en partant. Pas de panique, grâce à l'application HHCS, il va réparer cet oubli. Il saisit son PDA, et grâce à une incarnation de HHCS s'exécutant sur celui-ci, il parcourt virtuellement sa maison de pièce en pièce et ajuste si nécessaire les thermostats. Il n'a malheureusement pas terminé lorsque le tramway atteint la station à laquelle il descend. Jean suspend la tâche, puis poursuit les réglages de température en arrivant au bureau, mais en utilisant cette fois son PC qui lui offre une surface d'affichage bien plus large. Pour cela, le système interactif HHCS se *redistribue* : il migre totalement sur la nouvelle plate-forme qu'est le PC. Cette migration déclenche alors un *remodelage* de l'IHM : le plan de la maison se substitue aux boutons radio initiaux.

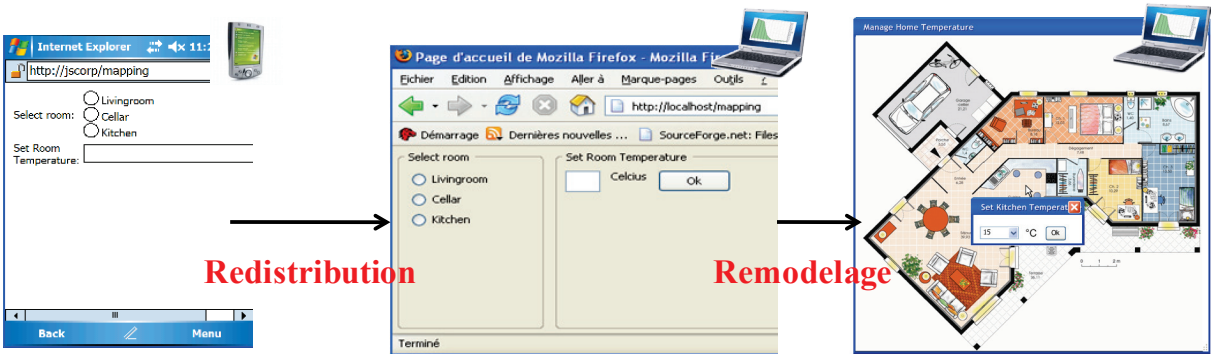

Figure 4. HHCS (Home Heating Control System), un système de chauffage plastique couvrant les deux leviers de plasticité : redistribution et remodelage.

Remodelage et *redistribution* sont les deux leviers de la plasticité. Ils peuvent être réalisés à la conception et par conséquent se baser sur des IHM préfabriquées. Mais dans le cadre de l'informatique ambiante le besoin d'adaptation peut aussi émerger opportunément.

Dans l'optique d'assurer la *plasticité,* la *malléabilité* des systèmes interactifs consiste à laisser manipuler les variables du système par tous les acteurs (de l'ingénieur à l'utilisateur final). Nous devons procurer les moyens de modéliser cette connaissance et sa variabilité selon plusieurs points de vue. C'est pourquoi nous nous sommes naturellement tournés vers l'Ingénierie Dirigée par les Modèles (IDM).

5 Approche : Ingénierie Dirigée par les Modèles (IDM)

L'ingénierie Dirigée par les Modèles (IDM) est une approche récente en génie logiciel visant à modéliser explicitement les différents aspects d'un système. On peut considérer l'IDM, vaste domaine de recherche, selon différentes perspectives. Dans cette thèse, nous utilisons l'IDM principalement comme un moyen d'expliciter les savoirs et savoir-faire (en IHM) et de les rendre « opérationnels » aussi bien que « communicationnels ». Ces différents concepts sont introduits brièvement ci-dessous.

5.1 Modèles communicationnels et modèles opérationnels

La notion de modèle, en tant que « représentation partielle d'un système complexe », est ancienne en informatique. Pour bien comprendre le changement de philosophie par rapport aux approches de modélisation telles que Merise ou SADT, il est pratique de distinguer les *modèles contemplatifs* des *modèles productifs* [Bézivin,2005]. La première catégorie correspond aux modèles destinés à une interprétation par des humains (les concepteurs d'un logiciel) ; la seconde catégorie ; aux modèles destinés à être interprétés ou transformés par des machines (pour de la génération de code).

Ces deux catégories ont été souvent opposées par les pionniers de l'IDM entre autres, car il s'agissait, pour l'IDM, de se démarquer des approches « contemplatives » historiquement prédominantes. Au final le message est que l'IDM veut promouvoir des modèles utiles, qu'ils permettent de générer du code ou non.

Le cas idéal est celui où un modèle est à la fois *communicationnel* et *opérationnel* ; autrement dit il doit permettre à la fois la communication entre acteurs humains et suffisamment précis pour être opérationnel du point de vue de la machine. Ces deux dimensions doivent être réconciliées autant que possible. Comme nous allons le voir par la suite, nous utilisons le concept de « vue », concept à la frontière entre IHM et IDM. Les vues fournissent les moyens de rendre les modèles *perceptibles* et *interactifs*. Nous allons également utiliser l'IDM pour expliciter les savoirs et savoir-faire, et ce aussi bien dans un but communicationnel qu'opérationnel.

5.2 Explicitation des savoirs *via* des modèles et des métamodèles

La caractéristique de l'IDM est que cette approche préconise l'utilisation systématique de langages explicites et instrumentés pour décrire les modèles et les rendre opérationnels. En effet si un langage n'est pas défini suffisamment précisément, les modèles ne le sont pas non plus et ne peuvent donc pas être opérationnels.

Par ailleurs, on peut utiliser *plusieurs formes de modélisation* pour plus de lisibilité selon l'acteur considéré : ingénieur, utilisateur, machine. C'est au travers de ces modèles que l'acteur peut expliciter son « savoir » ou exploiter d'autres savoirs. Par exemple, on va privilégier un diagramme

UML pour un spécialiste du génie logiciel, un arbre des tâches pour un spécialiste de l'IHM, alors qu'un fichier XML est plus approprié à un traitement par la machine.

Nous voyons également dans cette thèse qu'il est intéressant de distinguer les savoirs à plusieurs niveaux. Les *modèles* permettent de décrire les savoirs sur un système particulier, alors que les *metamodèles* (c'est-à-dire les modèles qui modélisent des langages de modélisation) permettent de décrire des savoirs plus généraux ; par exemple concernant le domaine de l'IIHM plutôt qu'une IHM en particulier.

Quoi qu'il en soit, expliciter les savoirs ne suffit pas lorsque l'on considère des systèmes capables d'adaptation (automatique ou sous le contrôle de l'utilisateur) : encore faut-il avoir le savoir-faire pour l'adaptation.

5.3 Explicitation des savoir-faire *via* des transformations

Historiquement les *transformations* n'ont pas été définies dans les prémisses de l'IDM [Gerber, et al., 2002]. Pour autant, elles sont la clé de voûte de l'IDM. Elles permettent de manipuler les modèles, fait nécessaire à la mise en œuvre de la *malléabilité*. A travers les *transformations* de modèles, on peut générer du code (dynamiquement ou non), assurer la cohérence du système lors de modifications, vérifier des propriétés, etc. De manière générale, la *transformation* de *modèles* est un moyen d'exprimer le savoir-faire de l'acteur qui les conçoit.

Dans bien des cas, le savoir-faire est implicite en IHM. Ce sont des experts qui le détiennent et bien évidemment il n'est pas facile ni de le généraliser, ni de le rendre explicite. La conception d'IHM est élevée d'un coté « artistique », en tout cas pour certaines applications. Comme nous le verrons par la suite nous nous intéressons aux IHM « systématiques » plutôt qu'« originales ». Ceci dit la malléabilité des IHM offre un compromis entre ces deux extrêmes, compromis permettant d'adapter des IHM plutôt systématiques, et de les rendre ainsi plastiques.

5.4 Vers l'ingénierie dirigée par les langages

Dans l'optique de la *malléabilité des systèmes interactifs,* il nous faut prendre en compte la diversité des contributions récentes en IHM : nouvelles technologies et spécialisations à des domaines particuliers. De nouvelles techniques d'interaction induisent bien souvent de nouveaux *langages* (ce sont les savoirs du domaine de l'IHMI dont nous parlions auparavant). Pour assurer une malléabilité sur ce plan technologique, nous nous proposons d'aller vers *l'Ingénierie des Langages Informatiques* (ILI), suite logique de l'IDM qui se focalise sur les langages supportant la modélisation.

C'est par l'expression systématique des *langages* et des « traductions » entre langages qu'on peut assurer la cohésion de l'ensemble des technologies et des acteurs impliqués dans la *malléabilité*.

6 Plan et Objectifs du mémoire

Le mémoire suit les principaux requis pour considérer la malléabilité des systèmes interactifs. Chacun des différents objectifs fait l'objet d'un chapitre. Concrètement le document est décomposé comme suit.

- **Chapitre II. *Etat de l'art de l'IIHM basé modèles.*** Ce chapitre décrit comment on est passé progressivement de la programmation d'IHM à l'Ingénierie des IHM (IIHM) basé modèles. On établit ensuite un état de l'art à partir d'un ensemble représentatif des outils existants dans ce domaine.

- **Chapitre III. *IDM pour l'expression des savoirs et savoir-faire.*** ici nous présentons les concepts de l'IDM ainsi qu'un certain nombre de techniques et outils utile pour expliciter savoirs et savoir-faire.

- **Chapitre IV*. Extraction de savoirs et savoir-faire à partir d'outils d'IIHM.*** Le chapitre II recense un certain nombre d'outils d'IIHM, mais chaque outil y est décrit informellement. Dans ce chapitre, au contraire les éléments d'IDM introduits en chapitre III sont utilisés pour extraire des « metamodèles » et des « transformations » à partir de ces outils. Chaque outil est considéré comme un système à étudier et on propose donc une *vision locale à un outil*. Le chapitre suivant bâtit une *vision globale à un domaine*.

- **Chapitre V*. Vers une cartographie des savoirs et savoir-faire en IIHM***. Alors que traditionnellement les états de l'art d'un domaine particulier prennent la forme d'un document textuel, nous montrons comment utiliser les concepts de l'IDM pour modéliser les états de l'art et établir une cartographie d'un domaine. Bien évidemment, l'approche que nous esquissons est illustrée sur le domaine des IIHM.

- **Chapitre VI. *Mega-IHM : malléabilité par l'utilisateur des savoirs et savoir-faire.*** Dans les chapitres IV et V savoirs et savoir-faire sont explicités et « formalisés » (au sens de l'IDM). Ils restent néanmoins de nature communicationnelle. Autrement dit, ils servent à comprendre l'essence même de chaque outil (vision locale, chapitre IV), ou l'agencement des concepts dans un domaine (vision globale, chapitre V). Il ne faut pas négliger ces aspects car c'est à partir de là qu'on peut bâtir l'expertise dans un domaine particulier. Dans ce chapitre, nous allons plus loin. Nous allons voir comment savoirs et savoir-faire peuvent être rendus opérationnels en les embarquant dans des systèmes et en les rendant modifiables à l'exécution *via* des IHM appropriées. Ce chapitre présente ainsi le concept de mega-IHM et son utilisation pour la malléabilité des IHM.

- **Chapitre VII.** ***Malléabilité : un outil pour la plasticité des IHM.*** Les concepts de malléabilité et de méga-IHM dépassent le cadre de la plasticité des IHM. Nous montrons cependant dans ce chapitre comment ces concepts peuvent être mis à profit pour adapter des IHM au contexte tout en préservant un certain nombre de qualités.

- **Chapitre VIII.** ***Le démonstrateur MARA (Model At Runtime Architecture).*** Les chapitres précédents proposent différents concepts. Dans ce chapitre nous allons montrer leur mise en œuvre à partir d'un démonstrateur appelé **MARA** (Model At Runtime Architecture). Ce démonstrateur est basé sur une architecture rendant la malléabilité possible grâce à l'utilisation et la sélection de modèles, metamodèles et transformations durant l'exécution même du système. Les IHM sont ainsi produites « à la volée » en fonction du contexte. Nous présentons dans ce chapitre cette architecture, puis quelques éléments de réalisation, et finalement une démonstration exposant comment une méga-IHM peut être utilisée pour de la plasticité d'IHM.

- **Chapitre IX.** ***Conclusion et perspectives.*** Finalement différentes conclusions sont tirées à partir de ces travaux et, comme nous allons le voir, cette thèse ouvre de nombreuses perspectives de recherche.

Chapitre II : Etat de l’art sur l'Ingénierie des IHM Homme-Machine basée Modèles

Cette thèse se déroule dans le domaine de l'Ingénierie des IHM. Nous en dressons un bref état de l'art. Soulignons de nouveau la distinction entre Ingénierie des IHM (IIHM) et IHM innovantes (IHMI). C'est le premier champ de recherche qui est étudié ici. Autrement dit nous ne traitons pas, dans ce chapitre, la description de techniques d'interaction innovantes (multimodalité, visualisation innovantes, nouveaux dispositifs, d'interacteur, etc.). Nous parlerons des modèles, langages et outils pour l'IIHM.

Tout d'abord, nous donnons un bref historique montrant comment le développement d'IHM est passé progressivement d'écriture de code de bas niveau à une activité d'ingénierie basée modèles (section 1). Nous voyons ensuite les différents modèles utilisés classiquement en IHM ainsi que les cadres de référence mettant en jeu ces modèles (section 2). Nous étudions quelques systèmes phares dans le domaine de l'IIHM. Nous avons sélectionné un ensemble varié de systèmes de conception représentatifs des grandes tendances (section 3). Une synthèse comparant les caractéristiques de ces systèmes est présentée en section 4. Finalement, ce chapitre se termine par l'identification des points de faiblesse des systèmes existants quant à la malléabilité. Cela nous permet de lister l'ensemble des points à considérer (section 5).

1 L'Ingénierie des IHM : un bref historique

L'Ingénierie des IHM est relativement récente si on la place dans le contexte de l'histoire de l'informatique. Il faut noter aussi que l'utilisabilité des systèmes n'a pas été une préoccupation première en informatique. Qui plus est, le concept même d'utilisateur, au sens où on l'entend aujourd'hui, a mis relativement longtemps à émerger. Pour bien comprendre la dualité utilisateur / ingénieur il nous faut donc brièvement reconsidérer les différentes étapes ayant mené à la situation actuelle en IIHM.

1.1 Les « mainframes », le temps des opérateurs et les IHM « invisibles »

Dans les années 50, les premiers systèmes informatiques étaient basés sur des IHM rudimentaires de types câbles, interrupteurs et voyants. Il faut dire qu'à cette époque tous les efforts se concentraient sur la puissance de calcul, l'expression des algorithmes, la fiabilité du matériel, etc. L'homme devait s'accommoder à la machine, et non pas l'inverse. D'ailleurs, on parlait d'*opérateurs* car c'est un personnel spécialisé qui interagissait avec la machine, aussi bien pour déposer et récupérer les lots de cartes perforées que pour vérifier le fonctionnement physique de la machine. Le temps de calcul était extrêmement coûteux et ni les programmeurs, ni a *fortiori* les utilisateurs n'interagissaient avec

la machine. C'était le temps des « centres de calcul », temples fermés réservés aux seuls « initiés ». Le concept d'IHM n'existait pas, ou tout au moins il n'était pas explicité. Ainsi l'interactivité n'a pas toujours existé en informatique.

1.2 Les premiers systèmes de dialogues

Il faut attendre les années 60 et 70 pour que les ordinateurs, encore très coûteux, puissent être utilisés en « temps partagé ». Grâce à cette innovation, de multiples « opérateurs », puis « programmeurs », puis « utilisateurs » ont pu peu à peu utiliser les ordinateurs de l'époque de manière parallèle et interactive. L'interaction se faisait typiquement via des terminaux composés pendant longtemps du couple clavier / imprimante. Puis est apparu dans les années 70 le terminal à écran.

D'un point de vue logiciel, il s'agissait alors de développer des systèmes de dialogues « ligne à ligne ». La programmation du dialogue était une activité qu'un programmeur d'application pouvait encore assumer en sus des fonctionnalités de l'application elle-même ; en tout cas tant il s'agissait d'IHM de dialogues relativement simples. A ce stade la dualité programmeur / utilisateur pouvait débuter car les applications informatiques commençaient à être « utilisables » et accessibles à des tiers, les *utilisateurs*. Cette époque marque aussi le début du génie logiciel. On savait déjà qu'il était fort complexe de développer des applications, mais on était encore bien loin de parler d'IHM comme un élément essentiel de tous systèmes informatiques.

1.3 Stations de travail, micro-ordinateurs et dispositifs d'interaction

Vers la fin des années 70 et surtout dans les années 80, l'apparition de la microélectronique bouleverse le paysage informatique. Les « stations de travail » commencent à se répandre peu à peu et la micro-informatique fait exploser les domaines d'applications envisageables. Avec « l'ordinateur de bureau » puis « l'ordinateur individuel », concepts révolutionnaires, on assiste peu à peu à une démocratisation de l'informatique. Cette démocratisation est cette fois freinée, non plus par les aspects matériels (le coût des machines va sans cesse en décroissant alors que parallèlement leur puissance augmente), mais au contraire par l'utilisabilité des applications. Avec la microélectronique, le nombre et la variété des périphériques d'entrées / sorties explosent. Le programmeur se doit de gérer, en plus du code applicatif, la variété des périphériques et leurs possibilités sans cesse croissantes. Il s'agit alors de programmation bas niveau pour piloter par exemple le contrôleur d'affichage.

```
        push es
        mov ax,0a000h
        mov es,ax
        xor dx,dx

again:  mov ax,4f05h
        xor bx,bx
        int 10h
        xor di,di
        mov al,1
        mov cx,0ffffh
        rep stosb
        inc dx
        cmp dx,8
        jb again
        pop es
```

Figure 1. Code assembleur affichant un écran bleu[1] (VESA 2.0, 800*600).

Dans cette phase, la gestion des interactions à un bas niveau d'abstraction consomme de plus en plus d'énergie pour des programmeurs. De plus, le résultat en termes d'utilisabilité n'est pas forcément au rendez-vous. On s'aperçoit que le problème de l'Interaction Homme-Machine n'est pas tant un problème de programmation de périphériques que d'utilisabilité. La balance entre la machine et l'utilisateur commence à changer, en tout cas dans la communauté naissante puis grandissante de l'IHM. Des laboratoires comme Xerox travaillaient depuis longtemps sur ces aspects, mais pour bon nombre d'informaticiens, le domaine des IHM n'existait tout simplement pas ou ne faisait pas partie de « l'informatique ».

D'un point de vue pragmatique, il devenait de plus en plus clair que la démocratisation de l'informatique ne pourrait se faire qu'à condition que les utilisateurs sans expérience de l'informatique puissent utiliser les applications. Visicalc (l'ancêtre d'Excel), et dans une moindre mesure Wordstar (un traitement de texte), sont des applications phares du début des années 80. Ces applications ont joué un rôle important dans le développement de l'informatique de bureau.

1.4 Des pilotes aux API et aux boites à outils

Le besoin de pilotes (drivers) de périphériques est apparu assez rapidement. Mais pour des problèmes de performances, les périphériques ont longtemps dû être gérés au niveau de l'assembleur ou avec des primitives systèmes relativement basses. La gestion en « temps réel » de la souris et des écrans graphiques consommait en effet une partie non négligeable de la puissance de calcul des machines.

C'est dans le début des années 90 et avec l'avènement de machines plus puissantes et moins onéreuses, que l'utilisation des IHM graphiques devient le standard *de facto* en tant que paradigme d'interaction pour toute application. Ce paradigme baptisé WIMP (Windows, Icons, Menu, Pointer)

[1] http://forum.hardware.fr/hfr/Programmation/ASM/programmation-graphique-600-sujet_52073_1.htm

est basé sur des techniques déjà anciennes (1973, la souris à Xerox ; 1984, le Macintosh). Cependant, dans les années 90, cette modification généralisée du paysage informatique n'a eu lieu que grâce à la disponibilité d'applications graphiques de plus en plus nombreuses.

Il est bon de noter qu'en parallèle de l'ordinateur « outil » (de calcul ou de travail) on retrouve les jeux vidéo. Ces jeux proposent dès 1971 des interactions originales : dates des premières bornes d'arcades avec notamment le jeu « Computer Space ». En 1972, c'est une petite révolution, Nolan Bushnell, co-auteur de Computer Space fonde Atari et sort la première console de jeu ainsi que le célèbre Pong, à l'époque très original car il s'agit de déplacer un palet pour renvoyer la balle dans le camp de l'adversaire. Les jeux vidéo montrent les prémisses de l'interaction avancée (IHMI).

Par ailleurs, les jeux vidéo sont une industrie lucrative qui devient dès 1981 non négligeable en termes de marché avec 5 milliards de dollars de chiffre d'affaires aux Etats-Unis. De nos jours encore, ce sont les consoles et jeux vidéo qui sont vraiment à la pointe en termes d'interaction : la console WII propose d'interagir grâce à des accéléromètres ; pour faire « un coup », il suffit d'agiter le contrôleur.

D'une certaine manière, cette prolifération des applications n'a pu avoir lieu que parce que les techniques de développement d'IHM graphiques se sont considérablement améliorées *via* la définition d'*IHM de programmation* (API en anglais) de plus en plus haut niveau, de plus en plus simples à utiliser. Les savoirs et les savoir-faire en IHM ont été rendus plus accessibles au programmeur lambda, non pas parce qu'ils ont été formalisés, mais plutôt parce que les APIs ont été accompagnées de documentation / livres / enseignements sur le thème du développement des IHM, problème devenu alors explicite.

En fait, l'apport des API peut être considéré selon deux perspectives correspondant à la dualité programmeur / utilisateur :

- *bénéfices pour le programmeur*. L'utilisation d'API de haut niveau a donné la possibilité aux programmeurs de réaliser des applications graphiques de plus en plus facilement. Bien que les premières applications soient de type « spaghettis » mêlant code applicatif et code d'IHM, peu à peu des savoir-faire ont commencé à émerger ainsi que des architectures de référence. Les APIs ont contribué à une certaine systématisation du code des IHM. La programmation d'IHM dans des langages de « haut niveau » (java, C, C++, C#, TCL) se voit renforcée par la notion de boîtes à outils souvent basées sur le concept d'objet. Le programmeur sélectionne des objets d'interaction (interacteurs ou widgets) tels que boutons, fenêtres, ascenseurs, etc. Il fixe ensuite leurs paramètres et les assemble. Les boîtes

à outils sont en général bien intégrées aux langages de programmation (par exemple SWING en Java).

```
    using System;
    using System.Drawing;
    using System.Windows.Forms;
    public class Form1 : Form
    {
          Button buttonTest;
          public Form1()
          {
               this.Text = "Hello World";
               this.Size = new System.Drawing.Size(300, 100);
               this.buttonTest = new Button();
               buttonTest.Location = System.Drawing.Point(110, 20);
               buttonTest.Size = new System.Drawing.Size(80, 30);
               buttonTest.Text = "Test";
               buttonTest.Click += new System.EventHandler(buttonTest _Click);
               this.Controls.Add(buttonTest);
          }
          private void buttonTest _Click(object sender, EventArgs evt)
          {
               MessageBox.Show("Clic sur bouton", "Clic sur bouton");
          }
}
```

Figure 2. Code C# déclarant une fenêtre avec un bouton « test ».

- *bénéfices pour l'utilisateur*. L'utilisation par le programmeur de boîtes à outils standardisées est également un avantage pour les utilisateurs. En effet les applications des années 80 se caractérisent dans un certain nombre de cas par des IHM au « look » complètement ésotérique, variant très largement d'une application à une autre en fonction du goût et des capacités du programmeur. L'utilisation systématique de boîtes à outils a permis au contraire une certaine homogénéisation. C'est un exemple où l'aspect *systématique* est un avantage par rapport à des IHM plus « originales ».

1.5 Conception visuelle d'IHM

L'utilisation d'API pour définir des IHM simplifie l'écriture du code des applications interactives, mais comme le suggère le code C# présenté ci-dessus (figure 2), le code qu'il s'agit d'écrire n'est pas franchement « intéressant ». En fait, l'intérêt de standardiser des interacteurs ne réside pas uniquement dans l'abstraction qu'il donne au programmeur ; il donne également la possibilité de concevoir visuellement des IHM et de générer le code correspondant, en tout cas partiellement. Cette fois-ci l'IHM n'est plus « programmée » mais au contraire « dessinée » par composition d'interacteurs préexistants. Dans la figure suivante, l'IHM en construction est représentée au centre de la copie d'écran. Elle est assemblée par « glissés-déposés » à partir des interacteurs disponibles dans la palette de droite.

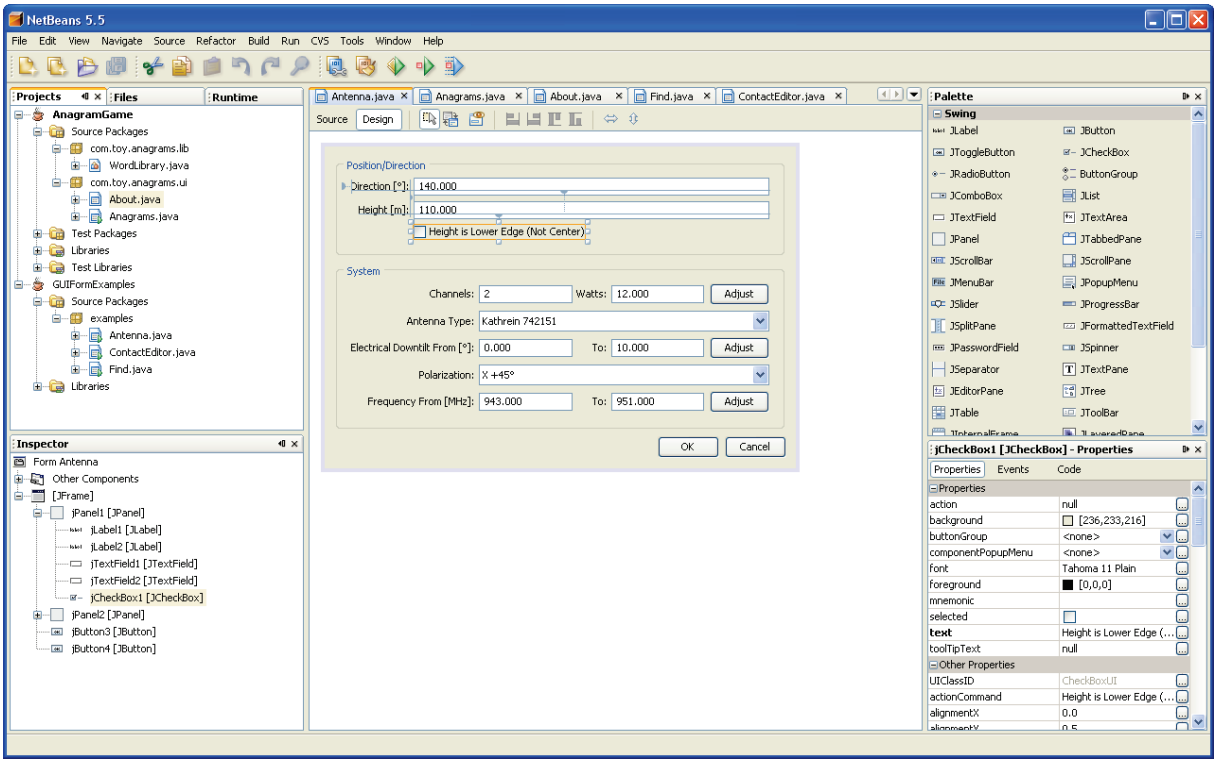

Figure 3. Editeur d'IHM JAVA/Swing avec NetBeans.

1.6 De la Programmation des IHM à l'Ingénierie des IHM

Mais qui au juste est censé utiliser un environnement tel que celui proposé ci-dessus ? Bien évidemment, on peut imaginer qu'il s'agit là du programmeur et il est alors possible de parler de « programmation visuelle ». Mais on peut également considérer ici d'autres rôles, peut-être implicites, mais pourtant ouvrant la voie à une vision différente sur le développement d'IHM.

Sachant qu'un tel environnement ne requiert pas de connaissance en programmation (il s'agit simplement d'assembler visuellement l'interface), il peut être utilisé par un *concepteur d'interface* qui ne serait pas forcement un *programmeur*. Cet aspect est important car il est fort probable qu'un programmeur n'a pas de compétences en termes d'utilisabilité. Par ailleurs, alors que l'interface pourrait être construite par un *ingénieur*[2] grâce à cet outil, elle pourrait être validée par un *ergonome*, etc.

Désormais l'IHM n'est plus seulement vue comme un problème de programmation, mais comme un véritable problème d'*ingénierie* impliquant la prise en compte de multiples *points de vue*. Autrement dit on passe progressivement de la *programmation des IHM* par un programmeur, à l'*Ingénierie des IHM* via la collaboration de différents ingénieurs aux expertises diverses et variées.

[2] Par la suite le terme ingénieur est utilisé pour représenter de manière générique les différents types d'acteurs pouvant intervenir dans le cadre de l'ingénierie du logiciel.

Il a en effet été reconnu que programmer des IHM, même au niveau boîte à outils, n'était pas aisé [Myers, et al., 1999]. Cette difficulté vient entre autres de l'apprentissage de l'API ou de la complexité (parfois l'absence de flexibilité) des architectures découlant de l'utilisation d'une boîte à outils. Cela devient beaucoup plus facile si l'on recourt à des outils d'aide à la conception, telles que des spécifications de plus haut niveau [Phanouriou, 2000].

L'aide fournie dans un contexte d'ingénierie peut être de plusieurs formes, allant de l'analyse de validité d'une interface conçue, à la génération du code. Les différents outils proposent soit l'utilisation d'un (ensemble de) *langage*(s) *spécifique*(s), soit un *cadre de travail* guidant la conception.

Dans [Phanouriou, 2000], huit grandes classes de *langages* pour les IHMs sont identifiées. Les plus significatives sont les réseaux d'états-transitions (UML), les grammaires, les langages de contraintes, les langages déclaratifs (HTML, XUL), les IHM de bases de données (IHM pour des requêtes : Microsoft Access),…

En fait, ces langages spécifiques permettent de décrire ce qu'on appellera des modèles (pris ici au sens large), c'est-à-dire des représentations partielles des IHM. Les modèles ne sont pas une préoccupation récente en interaction homme-machine, que ce soit au niveau de l'IHM avec par exemple les fichiers de configurations de X11 ou les outils **WYSIWYG** (What You See Is What You Get, figure 3 ci-dessus) jusqu'aux systèmes de génération d'IHM basés sur des spécifications bien plus abstraites se focalisant par exemple sur les besoins même de l'utilisateur.

Il existe ainsi différentes approches basées sur des modèles plus ou moins abstraits couvrant les différentes activités du cycle de vie tel qu'on le connaît en Génie Logiciel. Dans le cadre de cette thèse nous nous intéressons plus particulièrement aux approches orientées modèles qui proposent un haut niveau d'abstraction et permettent la construction d'IHM par la définition de multiples points de vue.

2 Approches orientées modèles pour les IHM

Les approches orientées modèles font partie du paysage de l'IIHM depuis un certain temps déjà. Après une première génération d'approches basée sur des modèles divers et variés (section 2.1), nous allons voir qu'un certain consensus est apparu autour d'une taxonomie permettant d'organiser ces différents modèles (section 2.2). Finalement, cette taxonomie a été étendue sous la forme d'un cadre de travail plus général, prenant en compte les problématiques liées à l'adaptabilité, la plasticité et la prise en compte du contexte d'utilisation (section 2.3).

2.1 Premières approches orientées modèles

Les premiers systèmes de conception orientés modèles se sont focalisés sur la génération d'IHM pour la visualisation de bases de données ou de production d'IHM ligne à ligne. Ainsi, ce sont des savoirs et des savoir-faire spécifiques à des domaines d'application particuliers qui ont pu être mis en œuvre.

Ces systèmes ne définissent pas explicitement la manière d'interagir mais utilisent comme heuristique de génération des patrons d'interaction. Pour les bases de données par exemple, on sait que toute interface doit permettre de réaliser les opérations classiques de création, suppression, modification et de visualisation de requêtes. De nos jours, de nombreux systèmes appliquent ce même principe et sont très largement utilisés par le grand public sur internet. C'est le cas des gestionnaires de contenu (Content Management Systems), des blogs, etc.

Alors que les approches décrites ci-dessus sont plutôt orientées données, une part importante des contributions fait en revanche référence à la notion de *tâche utilisateur* [Griffiths, et al.], [Mori, et al., 2004], etc. Par *tâche utilisateur*, on entend le couple <But, Procédure> où le *but* désigne l'état souhaité et la *procédure* la façon attendue pour atteindre ce but. Cette fois-ci les systèmes orientés tâches se focalisent sur l'interaction pour une application donnée. Par conséquent, ils ne se basent pas sur des tâches génériques comme c'était le cas dans les approches basées données. Les tâches « créer », « supprimer », etc. sont remplacées par des tâches dépendantes de l'application. Si certaines approches n'utilisent pas explicitement le mot « tâche », la grande majorité d'entre elles se focalise tout de même sur les buts et procédures des utilisateurs. La priorité est alors donnée à l'utilisateur – on parle de conception centrée utilisateur - contrairement aux approches traditionnelles en Génie Logiciel.

2.2 Cadre de référence pour les IHM basées modèles

Si tous les auteurs et outils de conception n'utilisent pas les mêmes modèles et la même terminologie, en revanche un consensus apparaît dans les années 90 autour d'une taxonomie de modélisation. Cette taxonomie met en avant quatre grandes classes de modèles : (1) *les modèles de tâches* ; (2) *les modèles de domaine*, appelés aussi (ou combiné avec) les *modèles d'application* ; (3) les *modèle de présentation abstraite*, appelé parfois dialogue ; (4) les *modèles de présentation concrète*. Comme on peut le voir dans la figure suivante, cette taxonomie se retrouve dans l'étude de da Silva [Silva, 2000]. Ce dernier ajoute le *modèle de présentation finale*. Nous décrivons ci-dessous successivement chaque type de modèle en donnant un exemple qui se base sur l'application HHCS introduite dans le chapitre I.

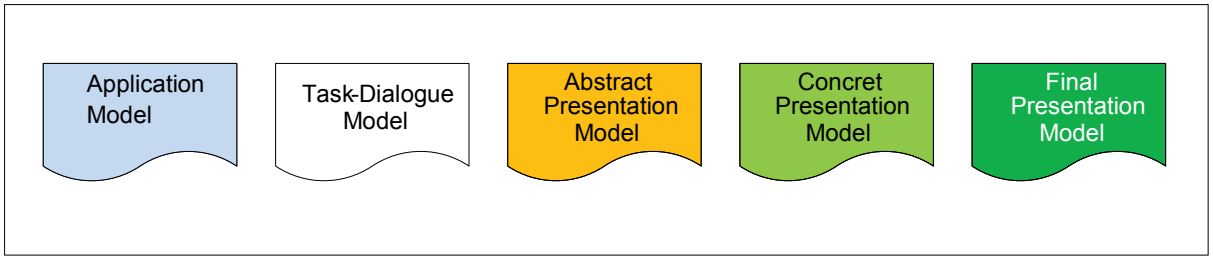

Figure 4. Cadre de travail originel basé sur [Silva, 2000].

- *Modèle d'Application.* Le modèle d'application fait référence aux modèles classiquement utilisés en GL lors de la conception d'un système. En IHM on s'intéresse ici aux « concepts du domaine » pertinents pour l'utilisateur (observables et/ou manipulables). C'est pour cela qu'on parle parfois du « modèle de domaine ». Par exemple dans l'application HHCS, les concepts importants sont les concepts de *maison*, de *pièce*, de *température*, etc. Ces concepts apparaîtront dans le modèle de domaine.

- *Modèle de Tâche*. Le modèle de tâche représente les *buts* et les *procédures* que l'utilisateur doit réaliser pour interagir. Ce modèle peut être un simple fil conducteur linéaire des actions à suivre par l'utilisateur mais il peut aussi être décomposé hiérarchiquement. Cette décomposition donne lieu aux arbres des tâches qui décrivent les buts et sous-buts (du plus abstrait au plus concret) et ceci récursivement, alors que les feuilles de l'arbre représentent des actions concrètes. Selon certaines définitions de modèles de tâche ce sont les tâches effectives que l'utilisateur doit réaliser.

  En ce qui concerne le cas d'étude HHCS, le but de l'utilisateur est de *gérer la température de sa maison.* Dans la réalité, il effectue le réglage du thermostat en passant de pièce en pièce. Ainsi, si l'on transpose cette approche en un système informatique, l'utilisateur d'HHCS doit *sélectionner une pièce* **puis** *affecter* (ou non) *la température désirée* pour la pièce sélectionnée. Cette analyse de l'activité sur le terrain se consigne en un modèle faisant apparaître trois tâches (figure 5) et un opérateur :

  - une tâche racine « Gérer la température de la maison » (*Manage Home Temperature*).
  - une première tâche fille pour sélectionner la pièce désirée (*Select Room).*
  - une seconde qui permet de régler la température de la pièce sélectionnée (*Set Room Temperature).*
  - un opérateur « *puis* » symbolisé par []>>. Exprimé en langage CTT, il désigne la séquence avec passage de paramètres entre sous-tâches.

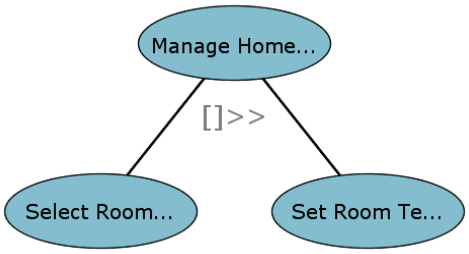

Figure 5. Modèle de tâche de l'application HHCS (gestion de température).

- *Modèle de présentation abstraite*. Ce modèle décrit l'ensemble des espaces d'interaction dont on n'a pas encore fixé les interacteurs. Autrement dit il est indépendant du style, de la disposition, etc. Ce modèle est issu à l'origine des *IHM graphiques* pour lesquelles il définit les conteneurs qui donnent lieu par la suite aux notions de fenêtres, de panneaux, etc. Pour les *IHM sonores* ou les *IHM vocales*, c'est un espace temporel qui est considéré. De manière générale, on parle plutôt d'*espace de dialogue* pour désigner les présentations abstraites, l'espace pouvant être physique ou temporel.

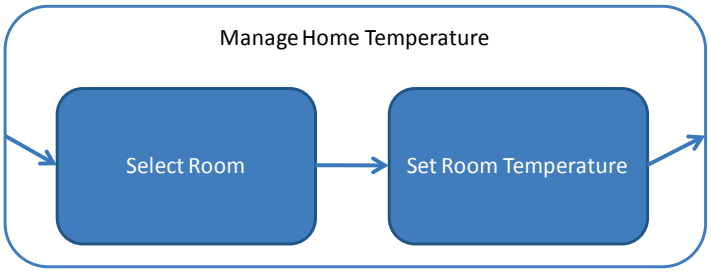

Figure 6. Modèle de présentation abstraite pour l'application HHCS.

- *Modèle de présentation concrète*. Ce modèle fixe les choix de l'IHM en termes de (1) *stylistique*, souvent assurée par des artistes ; (2) *d'interacteurs*, ce sont les objets qui composent l'interface et (3) *la disposition* (layout) des interacteurs, qu'elle soit spatiale ou temporelle. Le modèle de présentation concret est le plus souvent une maquette, mais cette dernière peut aussi être dépendante de l'implémentation qui en est faite.

- *Modèle de présentation ou interface finale.* Il décrit le rendu effectif à l'utilisateur. Il modélise l'exécution du code de l'interface utilisateur avec tous les détails associés. La figure ci-dessous présente un exemple d'interface finale utilisant la plate-forme Mozilla pour accéder à l'application HHCS. Cette IHM correspond au modèle de tâche présenté dans la figure 5.

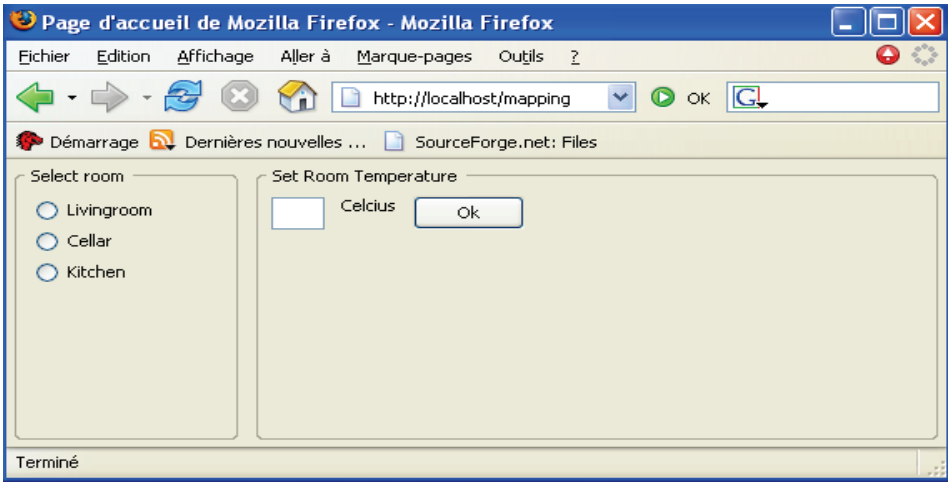

Figure 7. Exemple d'IHM finale pour l'application HHCS sur Mozilla.

2.3 Cadre de référence pour les IHM adaptatives et plastiques

La taxonomie précédente a été reprise dans le domaine de la plasticité et enrichie pour intégrer les nouvelles préoccupations de contexte d'usage et d'adaptation. Le cadre de référence CAMELEON présenté ci-dessous en est un exemple [Thevenin, et al., 2003].

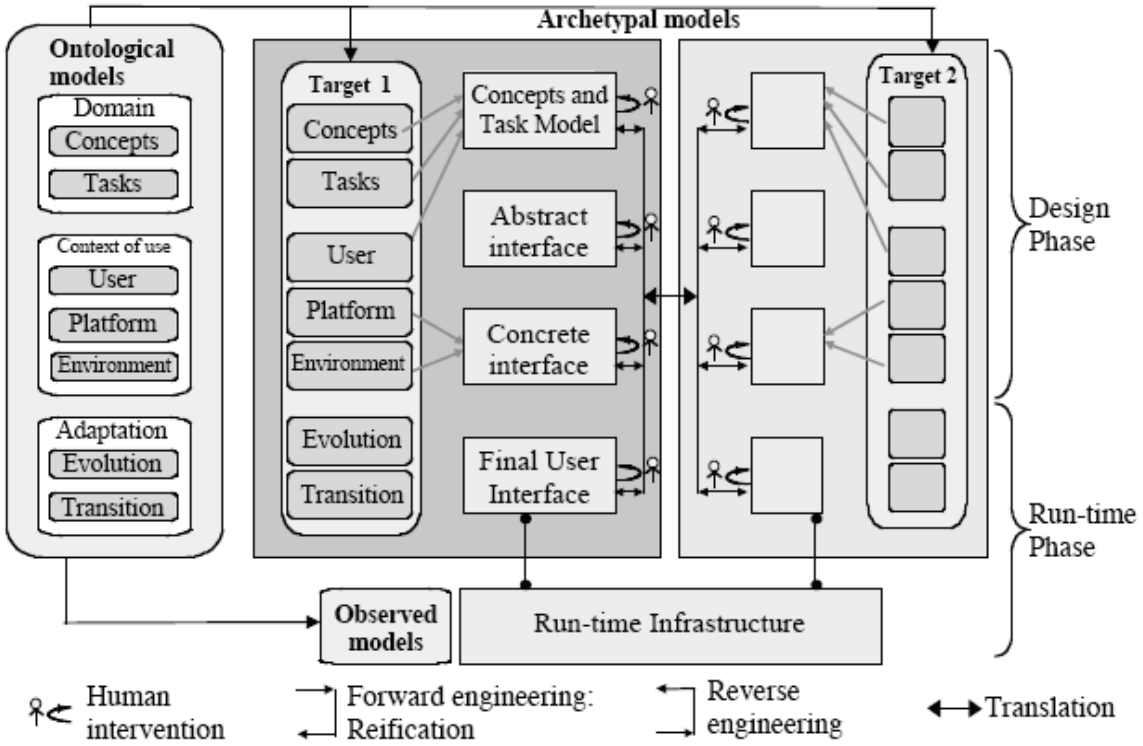

Figure 8. Cadre de travail CAMELEON : Modèles et processus de transformation.

Dans le cadre de travail, on retrouve les différents types de modèles énoncés section 2.3 (au centre de la figure), mais on voit également apparaître d'autres modèles. C'est le cas des modèles décrivant le contexte d'usage, lequel permet de modéliser les trois éléments du contexte : l'utilisateur, la plate-forme, et l'environnement (à gauche de la figure).

Il y a eu pendant longtemps une dichotomie entre les progrès des techniques d'interaction innovantes et l'ingénierie des IHM basée modèle. Cependant, comme le suggère la figure 8, on voit apparaître de plus en plus le besoin de modèles dans le contexte de ces IHM avancées. Ces modèles, d'abord « contemplatifs » et « communicationnels », se veulent de plus en plus « productifs » et « opérationnels ». Désormais on cherche à utiliser les modèles d'IHM pour valider celles-ci, pour prédire certaines caractéristiques, pour générer des IHMs, etc. On est bien loin de la programmation des IHMs telle que nous l'avons décrite au début de ce chapitre. Modèles et IHM innovantes ne sont pas du tout en contradiction, bien au contraire. Par exemple la notation ASUR permet de décrire des systèmes mixtes et s'intègre dans une approche générative [Gauffre, et al., 2007].

Quoi qu'il en soit, si on revient à la figure 8, qui reste très informelle, le cadre de travail proposé reste « contemplatif ». On peut voir cette figure comme une cartographie dans laquelle on peut placer différents travaux de recherche se focalisant sur un point particulier. La figure sert également à « expliquer » une approche globale à mettre en œuvre.

Le contenu des modèles n'est pas précisément défini. Il en est de même pour les transformations entre modèles (représentées par des flèches dans le diagramme). Celles-ci sont supposées être réalisées par des acteurs humains, ou tout au moins via une intervention humaine. Autrement dit, la figure 8 fournit un cadre structurant pour réaliser des IHM plastiques, mais les savoirs et savoir-faire qu'il est nécessaire de déployer ne sont pas représentés explicitement.

C'est à ce défi que nous nous attaquerons. Pour cela, nous allons établir un état de l'art des outils *concrets* permettant de développer des IHM à partir de modèles. Ce sera à partir de ces outils que nous allons, dans le **chapitre III**, extraire des savoirs et savoir-faire explicites.

3 Environnements et outils de conception et de génération d'IHM

Comme la majorité des outils récents s'appuient sur l'analyse des tâches utilisateur, cette section se base sur une énumération non exhaustive de ces principaux outils. Si les formats des outils diffèrent (arbres XML, grammaires, graphes), ils restent fidèles, au moins en partie, au cadre de travail défini dans la section précédente. En revanche, chaque outil est garant d'une spécificité ou d'une utilisation particulière que nous allons mettre en avant dans cette partie.

Nous faisons d'abord le point concernant la génération d'IHM (section 3.1). Puis nous introduisons les outils de conception d'IHM basés sur les modèles de tâches ou sur des modèles proches (section 3.2), les outils spécifiques basés sur la génération d'IHM (section 3.3), et enfin les outils plus avancés prenant en compte un cadre de référence de modélisation (section 3.4). Ces derniers proviennent soit de la communauté IHM (section 3.4.1), soit de contacts avec la communauté Génie Logiciel (3.4.2).

3.1 Génération d'IHM, une idée ancienne mais controversée

La génération d'IHM est un thème suffisamment controversé pour lui dédier une section dans cette thèse. L'idée est déjà ancienne et il convient de distinguer différentes classes d'approches, plutôt que de les mettre toutes dans la même catégorie sans discernement.

Les premiers outils ont visé la *génération directe d'IHM* à partir de descriptions de bases de données [MasterMind] en se fondant sur des tâches d'interactions figées (Création, Suppression, Modification, etc.).

Si les modèles sont utilisés par la plupart des acteurs académiques en IHM, au moins à titre d'illustration ou de documentation, en revanche les transformations et outils de génération créent de grands débats. En 1999, Myers [Myers, et al., 1999] statue sur les échecs des premiers systèmes de génération automatique.

Cependant il se refuse à condamner les approches génératives mais les reconsidère au vu des progrès des recherches connexes, en particulier en Génie Logiciel. Myers lui-même explore d'ailleurs la génération automatique d'IHM [Nichols, et al., 2002]. Cette approche *spécifique à un domaine*, par exemple les systèmes multimédia dans le cadre de l'approche de Nichols, capitalise un savoir-faire de conception désormais bien rodé.

Les approches industrielles tirent également parti de la génération automatique ou semi-automatique pour les domaines dans lesquels elles ont une expertise éprouvée. Après avoir passé des années à concevoir des applications, on commence à détacher des patrons (création, suppression,…) puis on en automatise la mise en œuvre. Un exemple marquant de cette tendance est la PME NetFlective qui, forte de son expérience sur « + de 40 logiciels spécifiques » [NetFlective], propose un cadre de travail et un moteur de génération de code. Elle offre entre autres la génération d'IHM par l'intermédiaire de diagrammes d'activité UML et de stéréotypes (par exemple <<Screen>>).

Des approches spécifiques à l'IHM, centrées données, ont également vu le jour. Par exemple le système Leonardi [Lyria] adopte une approche de génération d'IHM multi-cibles à partir de modèles

issus de base de données. Enfin, plus simplement c'est aussi le cas des CMS [Joomla.org], des blogs, etc.

Enfin dans le monde de la recherche, on retrouve des modélisations plus centrées utilisateur, qui pour autant ne condamnent pas les origines de la génération automatique. Récemment des cadres de travail ont été réalisés à partir du savoir en conception et modélisation des IHM (cf. CAMELEON). Notons en particulier l'initiative du consortium UsiXML qui propose un ensemble de langages et d'outils pour réaliser la conception et la génération d'IHM.

3.2 Outils basés « tâche »

Dans le monde des IHM basées modèles, on compte différents outils basés sur la description de tâches utilisateurs. Ces outils n'utilisent pas systématiquement le terme de « tâche » mais leur préoccupation est la même : décrire l'interaction en fonction de buts et procédures pour les utilisateurs. La plupart du temps ces outils décrivent aussi les concepts du domaine qui sont manipulés par les tâches. Ci-dessous nous décrivons brièvement CTTe, KMADe, Trident, Tadeus et DiaTask selon la perspective modèle de tâche.

CTTe [Mori, et al., 2004]. CTTe est l'acronyme pour Concurrent Task Tree Editor. Il s'agit d'un outil d'édition de modèles centré tâche, bien qu'il permette de définir aussi les concepts du domaine. Dans l'outil CTTe on peut réaliser un ensemble de tâches décomposées hiérarchiquement (figure 9 ci-dessous).

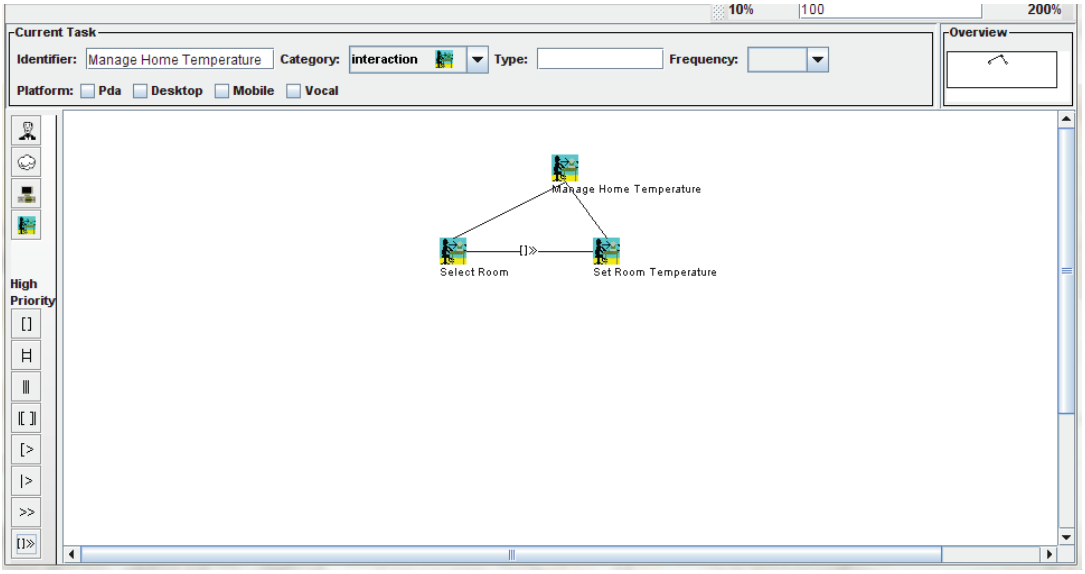

Figure 9. Modèle de tâche dans l'éditeur CTTe.

Les tâches à un même niveau de décomposition sont agencées entre elles par des opérateurs (l'opérateur []>> sur la figure correspond à la séquence avec passage d'information). Les opérateurs,

basés Lotos, permettent de définir séquences, parallélisme, concurrence, et complémentarité entre tâches utilisateur.

CTTe permet en outre d' « attacher » un ensemble de plates-formes aux tâches utilisateur. La relation tâche-plate-forme représente les restrictions de réification de la tâche sur les plates-formes cibles. Par exemple, certaines tâches de consultation ne sont pas visibles sur une plate-forme mobile par manque d'espace d'affichage. CTTe est l'outil de référence pour le format CTT défini par ces mêmes auteurs.

Associé à **CTTe**, l'outil **Teresa** permet la génération d'IHM à partir d'un modèle de tâche CTT. **Teresa** a subi de nombreuses modifications. La génération dans **Teresa** est pilotée par la décomposition en espaces de travail de certains sous-arbres types. On note la possibilité de générer des IHM *multi-modales* ainsi que la migration dynamique de tâches sur des plates-formes cibles.

KMADe[Baron, et al., 2006]. Alors que **CTTe** est l'éditeur pour CTT, **KMADe** est associé à la notation MAD [Scapin90]. **KMADe** permet la conception d'arbres des tâches et facilite la spécification d'un ensemble de variables ayant trait à l'ergonomie. Derrière la spécification de **KMADe** il y a beaucoup d'informations destinées à la documentation et à la simulation.

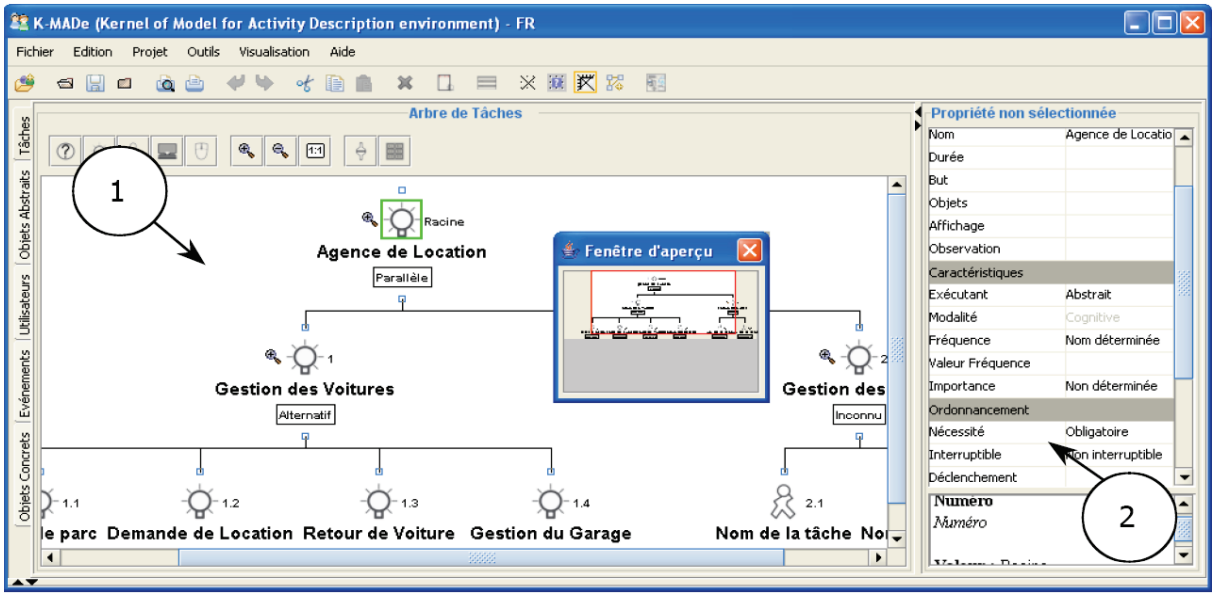

Figure 10. **KMADe**. En 1, l'espace d'édition de l'arbre des tâches : en 2, l'édition des attributs d'une tâche.

Trident[Bodart, et al., 1995]. Trident est un outil de conception et de génération d'IHM utilisant en entrée un modèle de chaîne d'activité. Ce modèle décrit le flux de données entre

l'ensemble des fonctions et leurs paramètres pour définir la logique de l'interaction. La couverture des chaînes d'activité regroupe le modèle de tâche et le modèle de concept.

La génération d'IHM de **Trident** est une génération directe, à partir du modèle « de chaîne d'activité », en Objets d'Interaction Abstraits (indépendants de l'environnement cible), puis en Objets d'Interaction Concrets (Interacteur). Le savoir-faire de **Trident** est, comme la plupart des outils de génération des années 80-90, adapté uniquement aux applications de type formulaire. Les règles de générations sont diluées dans le code de l'outil. Elles sont figées pour une boîte à outils donnée. Cette approche générative est donc de type « boîte noire ».

Tadeus[Schlungbaum, et al., 1996]. Cet outil de conception d'IHM semi-automatisée permet de définir les modèles de tâche et concepts du domaine ainsi qu'un modèle de dialogue. Le modèle du dialogue présenté dans **Tadeus** est une alternative au modèle de tâche CTT.

DiaTask[Reichart, et al., 2004]. Tout comme Tadeus, dont il s'inspire, DiaTask modélise des graphes de dialogue. **DiaTask** a subi plusieurs évolutions durant sa conception. Au départ la modélisation était centrée sur le modèle de dialogue (**Tadeus**). Par la suite, **DiaTask** se voit enrichi des différentes tâches exprimables dans un espace de dialogue (figure 11 ci-dessous). **DiaTask** s'est aussi enrichi d'un éditeur de modèles d'interacteur, chacun des interacteurs pouvant être lié à un espace de dialogue.

A partir d'un modèle de dialogue, **DiaTask** permet la génération d'une maquette d'IHM (simpliste) pour faire du prototypage rapide. Mais **DiaTask** permet aussi l'intégration d'IHM développées « à la main » dans un autre outil et leur liaison au modèle de dialogue. L'originalité de **DiaTask** repose, par conséquent, sur le fait de pouvoir intégrer des parties faites à la main lors de la génération automatique du modèle d'interacteur.

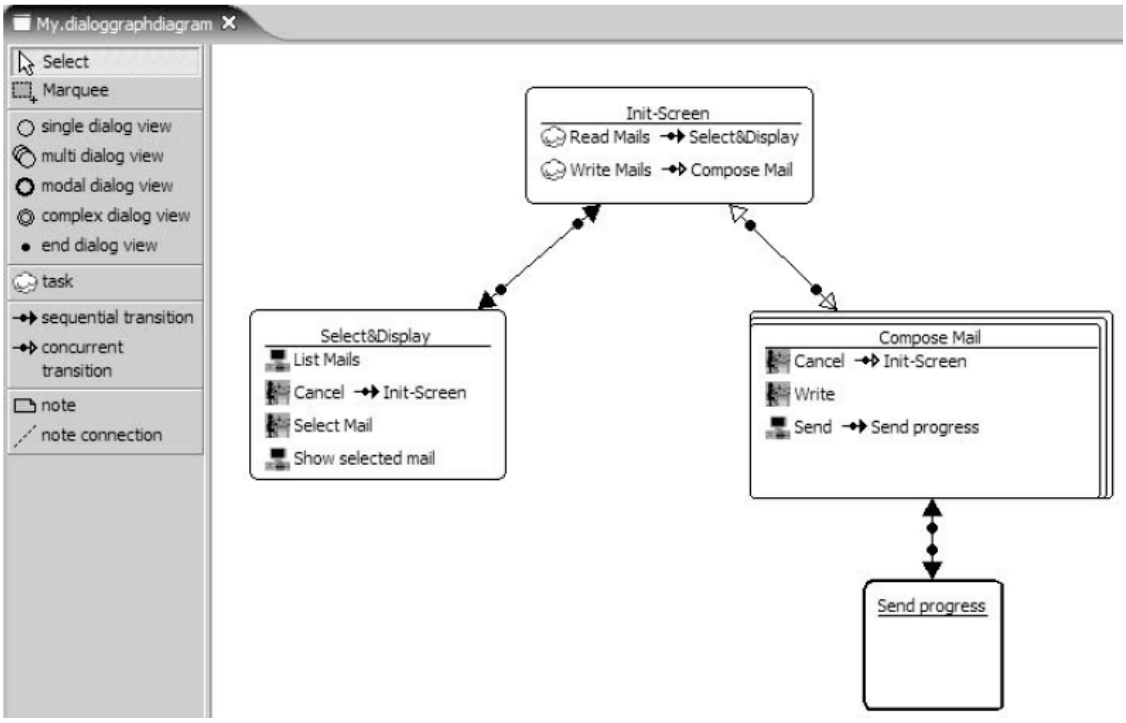

Figure 11. Edition du diagramme de dialogue.

3.3 Outils spécifiques à la génération automatique d'IHM

Les outils présentés dans cette section proposent des modèles de conception et de génération spécifiques à une problématique particulière. Le but de ces outils est de faciliter la génération automatique d'IHM dans le domaine pour lequel ils sont prévus. Nous présentons à titre illustratif Supple et Huddle.

*Supple [*Gajos, et al., 2004*]*. L'outil **Supple** utilise un langage de spécification légèrement différent de CTT. Il s'agit d'une description fonctionnelle de l'interface se focalisant sur les fonctions des données manipulées plutôt que sur les buts des utilisateurs. **Supple** embarque aussi une description des capacités en termes de boîtes à outils des plates-formes ciblées.

En outre **Supple** permet la génération d'IHM de contrôle sur différentes plates-formes à partir d'une description fonctionnelle et d'une description des capacités des plates-formes d'accueil. La description fonctionnelle de l'IHM se focalise sur les fonctions des données manipulées tandis que la description du dispositif embarque l'ensemble des interacteurs disponibles pour cette plate-forme. Le type d'interacteur est sélectionné en fonction du type de données manipulé dans le modèle de décomposition fonctionnelle. Par exemple un entier de [1-10] donnera lieu à une liste déroulante, un booléen sera représenté par une case à cocher, etc.

*Huddle [*Nichols, *et al., 2006]*. **Huddle** est un outil proche de **Supple**, mais il est destiné plus spécifiquement aux télécommandes universelles pour des plates-formes multimédia, par exemple

des récepteurs satellites, téléviseurs, magnétoscopes, lecteurs DVD, etc. **Huddle** utilise un langage pour décrire les capacités des ports des dispositifs cibles. Pour chacun de ces ports, **Huddle** décrit un ensemble de flots d'actions à réaliser : c'est une forme de description de tâches sans hiérarchie arborescente.

Huddle est basé sur des heuristiques offrant une couverture multi-plate-forme dans le domaine multimédia ; c'est-à-dire qu**'Huddle** permet d'obtenir une IHM pour contrôler différents dispositifs (magnétoscope, lecteur DVD) mais pas forcement sur différents dispositifs (téléphone ou PC). **Huddle** propose [Nichols, et al., 2006] une gestion de la cohérence entre un même type de dispositifs : différentes marques par exemple.

3.4 Outils basés sur des cadres de modélisation

Les premières générations d'outils étaient basées sur un ou deux modèles, comme par exemple les modèles de tâches, mais il est apparu peu à peu que différents points de vue pouvaient être explicités. Dans cette partie nous décrivons des outils basés sur des langages plus complexes et regroupant un ensemble de sous-modèles. Ces langages donnent plus de souplesse en termes de description et de génération.

Nous pouvons distinguer deux catégories d'outils. D'une part ceux correspondant à la poursuite des travaux dans la communauté IHM ; d'autre part ceux satisfaisant à une intégration des concepts et techniques de Génie Logiciel. Dans la première catégorie on présente à titre d'illustration **ArtStudio**, **IdealXML** et **DynamoAid**. Bien que les versions les plus récentes de **Teresa** soient basées sur le cadre de référence CAMELEON, cet outil ne sera pas mentionné de nouveau pour éviter les redites. L'approche « Génie Logiciel » sera représentée par **UMLi**, **Mantra** et **EMode**.

3.4.1 Outils inspirés des modèles IHM

La définition de cadre de référence tel que CAMELEON a montré l'importance de multiple modèles en IHM. Elle a aussi donné lieu à différents outils.

ArtStudio[Thevenin, 2001]. **ArtStudio** est basé sur le cadre de travail CAMELEON. L'éditeur de tâche **d'ArtStudio** est inspiré de celui de **CTTe**. **ArtStudio** propose en outre l'édition des modèles d'espaces de travail et d'IHM concrètes. Cependant cet outil ne couvre pas les mises en correspondance entre les différents modèles qu'il propose.

Le point fort **d'ArtStudio** ne réside pas dans son environnement de modélisation mais dans la génération d'IHM multi-cibles. Il embarque en effet un ensemble de règles pour choisir l'interface la plus appropriée à la plate-forme cible. **ArtStudio** propose les mêmes capacités que **Teresa** en termes

de génération multi-cibles. La faiblesse de ce système réside dans l'indéterminisme du choix des heuristiques de génération.

IdealXML [Montero, et al., 2005]. Le langage UsiXML [Limbourg, et al., 2004] regroupe un ensemble de modèles correspondant au cadre de référence CAMELEON. Dans la version étudiée ici (figure 10), **IdealXML** propose plusieurs éditeurs pour les modèles de tâches et concepts du domaine ainsi que pour les modèles d'IHM abstraites. Certaines versions **d'IdealXML** permettent même l'édition de mise en correspondance entre modèles. **IdealXML** embarque un simulateur d'arbre des tâches ainsi qu'un générateur d'espaces de dialogue.

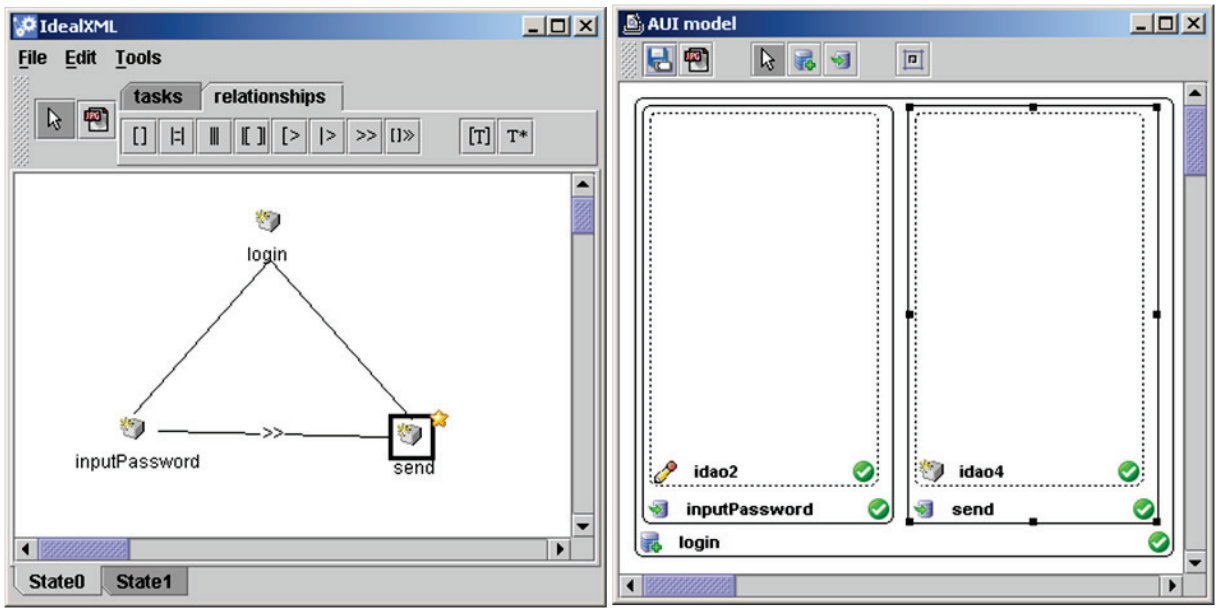

Figure 12. A gauche, l'éditeur de tâche **d'IdealXML**,
à droite, le modèle d'IHM abstraite.

La génération, pour la version **d'IdealXML** étudiée ici, se limite au modèle d'IHM abstraite. Cependant, couplé aux outils de génération du consortium UsiXML, on peut à partir d'un modèle provenant **d'IdealXML** générer du code pour différentes technologies. Notons particulièrement l'outil **transformiXML** qui réalise des transformations de modèle à modèle.

Le point fort de **transformiXML** est d'être découplé d'un environnement de modélisation et de laisser ainsi le choix de l'ordre et de la portée des transformations. Ainsi on peut réaliser des affinages de modèles de tâche par transformations successives. Le second point fort de cet outil de transformation est l'édition et la personnalisation des transformations, rendant les outils de génération beaucoup plus pertinents.

*Dynamo-Aid [*Clerckx, et al., 2004*].* Cette approche définit à la fois un processus et une architecture basés sur des modèles pour réaliser des IHM sensibles au contexte. L'outil **Dynamo-Aid** prend en compte un ensemble de modèles de conception et de contextes issus du cadre de référence **Dygme** [Coninx, et al., 2003], cadre proche de **CAMELEON**. Il décrit en particulier un modèle de micro-dialogue. La particularité de cet outil est d'offrir des mises en correspondance entre modèles afin de conserver les choix de conception. Cet outil propose des « *mappings* » entre éléments abstraits (tâches) et éléments concrets (interacteurs). Le cadre applicatif de cet outil comprend entre autres l'arrivée et la disparition de services pouvant étendre ou réduire les fonctionnalités de l'application.

Dans le cadre de prototypage rapide, **Dynamo-Aid** facilite la simulation de contexte et la re-conception dynamique. L'exécution de l'interaction peut être suivie sur le modèle de tâche correspondant. D'un point de vue « génération de code » les auteurs ne se sont pas clairement prononcés. Les heuristiques sont en tout cas propres à l'outil, lequel propose plus une aide qu'une automatisation du processus de conception : on établit l'ensemble des variantes qui sont ensuite affiliées au contexte.

3.4.2 Systèmes basés Génie Logiciel

Les modèles sont au cœur du Génie Logiciel depuis longtemps, et en particulier plus récemment au centre de l'ingénierie dirigée par les modèles. Cette approche, dont les caractéristiques principales sont présentées dans le chapitre suivant, a donné lieu à des retombées dans le monde de l'Ingénierie des IHM. Trois exemples sont présentés ci-dessous.

*UMLi[*Silva, et al., 2003*].* UML est un standard *de facto* d'un point de vue industriel, pourquoi ne pas l'utiliser ? C'est ce raisonnement qui à été poursuivi par **UMLi**. **UMLi** est un langage défini par un ensemble de profils **UML** spécifiques à l'interaction, c'est-à-dire qu'**UMLi** propose une extension d'**UML** pour spécifier des IHM. Il n'y a pas d'outils dédiés spécifiquement à **UMLi**, tout outil **UML** pouvant gérer les profils convient. **UMLi** à été instrumentalisé sous **ArgoUML**, voir figure 13, où l'on spécifie une présentation abstraite à l'aide du profil **UMLi**. Le langage **UMLi** couvre les concepts du cadre de travail originel en IHM basée modèle.

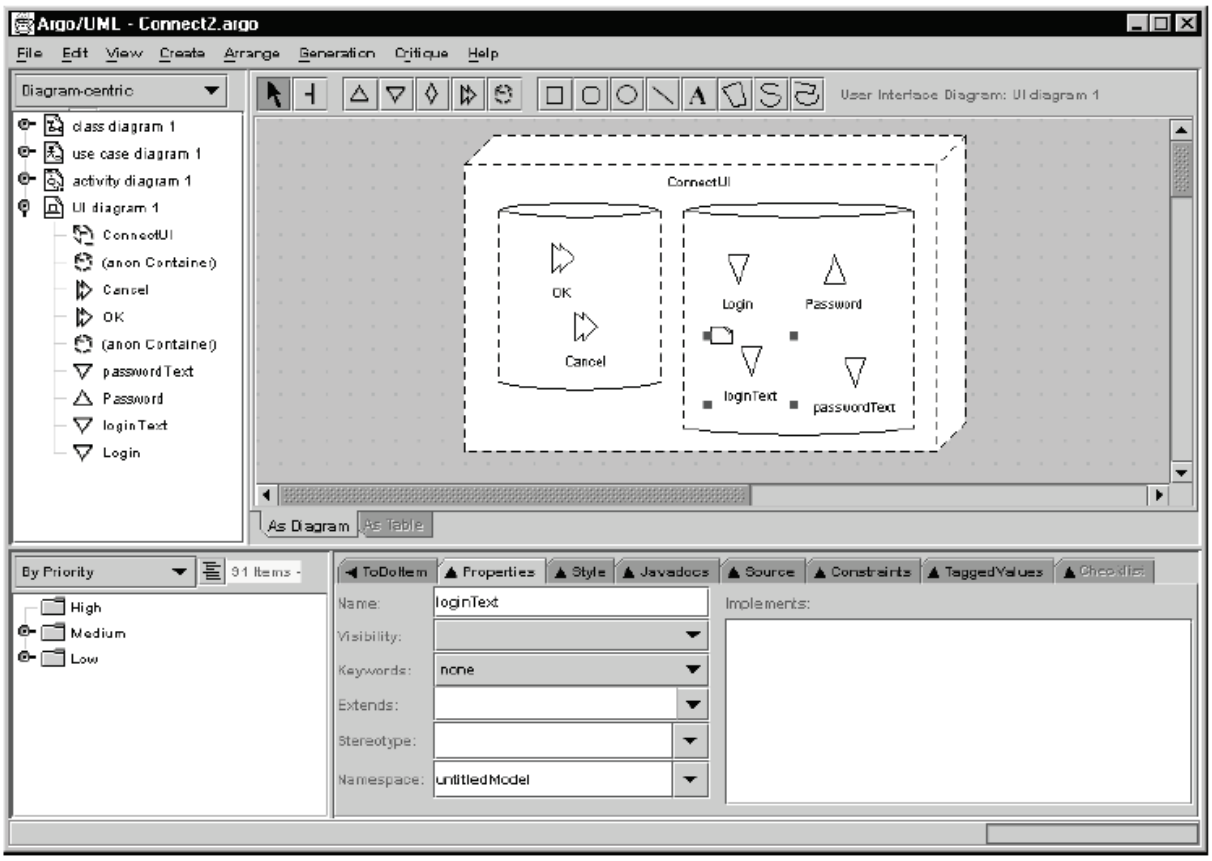

Figure 13. L'outil Argo UML personnalisé avec le profil **UMLi**.

*Mantra [*Goetz, et al., 2006*]*. Cet outil est basé sur une approche IDM de la conception d'IHM centrée tâche utilisateur (basé sur CTT). Dans **Mantra,** le modèle de dialogue est généré semi-automatiquement à partir du modèle de tâche. Par la suite ce modèle de dialogue est transformé en code exécutable sur un ensemble de plates-formes cibles. **Mantra** est inspiré **d'ArtStudio** et du cadre de travail CAMELEON. Par contre, **Mantra** utilise non pas des techniques *ad-hoc* d'implémentation, mais au contraire des outils de transformations provenant de la communauté **IDM**, en particulier **ATL**. L'approche générative de **Mantra** suit un principe de génération directe à l'instar de CAMELEON. **Mantra** permet à partir d'un modèle de dialogue commun de générer des IHM vers des cibles hétérogènes tout comme **ArtStudio**.

*EMODE[*Behring, et al.*]*. Le consortium allemand du projet ITEA EMODE (Emode, 2005) a présenté un ensemble de langages basés sur UML, un peu à la manière **d'UMLi**. Les langages d'**EMODE** couvrent les différents aspects du cadre de référence CAMELEON, en insistant également sur la partie Application (Modèle du noyau fonctionnel, modèles de services). Les modèles **EMODE** couvrent également la partie multi-modalité et adaptation. La génération d'IHM se fait alors par un outil externe (non défini), à l'instar du consortium **UsiXML**.

4 Synthèse des outils basés modèles

Nous avons vu que différents outils basés modèles ont été proposés à différentes époques, en utilisant différentes approches pour résoudre des objectifs variés. Le développement de tels outils constitue un ensemble de savoirs et de savoir-faire qu'il est nécessaire de capitaliser. Mais ceci n'est pas simple. Ci-dessous nous proposons une brève synthèse, tout d'abord en considérant les savoirs manipulés par chaque outil, puis les savoir-faire permettant la transformation et génération d'IHM.

4.1 Synthèse concernant les savoirs représentés par les outils basés modèles

Pour développer une IHM il est rapidement apparu qu'on devait considérer le système à construire selon des points de vue complémentaires. Les différents modèles et langages mis en œuvre par chaque outil représentent chacun une parcelle de ce savoir. Le tableau 1 ci-dessous décrit la couverture des outils étudiés en se basant sur le cadre de référence présenté dans ce chapitre. Les quatre premières colonnes allant de *Tâche* à *Interacteur* font référence aux principaux modèles de conception en IHM. Les deux colonnes suivantes, *Utilisateur* et *Plate-forme*, décrivent en partie le contexte d'usage. L'*Environnement* n'est pas représenté ici car il n'est couvert explicitement que par un seul outil. Finalement la dernière colonne résume certaines spécificités propres à certains outils.

	Tâche	Concept/ Application	Dialogue	Interacteur	Utilisateur	Plate-forme	Specificités
CTTe	oui	oui				oui	
Tadeus	oui	oui	oui	oui			
Trident	oui	oui	oui	oui			
DiaTask	oui	oui	oui	oui			
KMADe	oui	oui			oui		Evaluation
ArtStudio	oui	oui	oui	oui		oui	Environnement
IdealXML	oui	oui	oui				
Supple	oui	oui			oui	oui	Boites à outils
Huddle	oui	oui				oui	Entrée/sortie plates-formes
Dynamo	oui	oui	oui	oui	oui	oui	Micro Dialogue Smulation contexte
Mantra	oui	oui	oui	oui		oui	Langages explicites
Emode	oui	oui	oui	oui	oui	oui	Langages explicites

Tableau 1. Classification des langages supportés par les outils par rapport à leur couverture sur le cadre de travail CAMELEON.

Notons que les anciennes générations d'outils de modélisation d'IHM (**Tadeus**, **Trident**) se sont focalisées sur la description Tâches & Concepts. C'est aussi le cas d'outils plus récents comme

KMADe qui s'intéresse à l'évaluation d'IHM, **Supple** et **Huddle,** lequel permet une génération multi-cibles. Il est à remarquer que bien peu d'outils considèrent le contexte d'usage tel que défini dans (Thevenin, et al., 2003) : ce sont **ArtStudio**, **Emode** et **Dynamo-Aid**. Le modèle utilisateur est aussi très peu considéré ; à visée de documentation dans **KMADe**, il sert en revanche de trace dans **Supple**.

Par ailleurs, seuls les outils les plus récents, grâce à une forte relation avec la communauté IDM, sont conformes à des langages définis explicitement par des metamodèles. Ce concept est défini dans le chapitre suivant, et il constitue un élément important pour représenter des savoirs. En fait la plupart de ces outils sont des éditeurs de modèles accompagnés éventuellement d'un moteur de génération de code, ou à défaut, de réification des modèles.

4.2 Synthèse concernant les savoir-faire inclus dans les outils de génération d'IHM

Le tableau 2 résume les avantages et inconvénients de chacun des outils de génération d'IHM. La première colonne fait référence à l'édition possible des tâches de l'application : dans le cas de CMS ces tâches sont fixées. La colonne n°2 indique si la génération peut se faire par étape, avec éventuellement un réajustement du concepteur. La colonne n°3 fait référence à l'intégration possible de code de modèles d'IHM concrètes réalisées à la main. La colonne n°4 s'intéresse à la multiplicité des cibles choisies (en terme de dispositifs, de langages de programmation, de boîte à outils). La 5^{ième} colonne, quant à elle, se préoccupe des transformations à l'exécution (ou interprétations des modèles). La dernière colonne est relative à la mise à disposition d'un outil d'édition des heuristiques de génération.

	Tâche éditable	Génération Directe	Intégration de code/CUI	Multi cibles	Transfo. dynamique	Transfo. éditable
Leonardi		Oui	?	Oui		Oui (3)
Trident	Oui	Oui				
Tadeus	Oui	Oui				
DiaTask	Oui	Oui	Oui			
Teresa	Oui			Oui	Oui (2)	
Art Studio	Oui		?	Oui		
Supple	Oui (1)	Oui		Oui		
Huddle	Oui	Oui		Oui		
Mantra	Oui			Oui	?	
TransformiXML	Oui		Oui	Oui	Oui	Oui

(1) Définie sous forme de fonctions

(2) Dans le cadre de migrations

(3) Uniquement dans la version **Acceleo**

Tableau 2. Classification des outils de Génération d'IHM.

Les outils de génération d'IHM sont souvent indissociables des outils de modélisation. Une grande majorité des outils propose une génération directe à partir d'un ou plusieurs modèles. Mais bien souvent ces outils généralistes sont limités aux cas simples et ne proposent aucune alternative ; c'est d'ailleurs ce qui leur a valu une mauvaise réputation dans le domaine des IHM. Des outils plus récents, parfois commerciaux, proposent une approche spécifique à un domaine d'application (Domotique, E-commerce, etc.). En reconsidérant les approches génératives dans un cadre spécifique, les outils reçoivent un meilleur accueil. En effet, il est plus aisé de couvrir un domaine d'application pour lequel on maitrise une catégorie particulière d'IHM cible. Par exemple, l'outil **Huddle** qui permet la génération de certaines IHM dédié (contrôleur son, vidéo, etc.) est présent dans les conférences majeures en IHM (UIST, CHI).

En revanche, très peu d'outils proposent d'ouvrir leurs heuristiques de génération pour en faciliter la personnalisation. Pourtant, c'est uniquement par cette personnalisation des transformations de modèles que les spécialistes du métier peuvent capitaliser leur savoir-faire. On note cependant que la tendance actuelle est d'externaliser les transformations ; l'outil **transformiXML** devient un véritable outil de conception et de génération d'IHM. L'état de l'art ne montre globalement que peu d'adaptabilité aux nouvelles techniques d'interaction. Les langages sont figés par les outils, soit à cause d'une vision spécifique soit par nécessité. Il apparaît donc nécessaire d'ouvrir à l'édition

l'ensemble des éléments couvrant la génération automatique, à savoir : les langages, les transformations et même les modèles des outils.

5 Synthèse : état de l'art et besoins pour la malléabilité

La malléabilité des systèmes interactifs requiert :

1. Une grande souplesse face aux langages de modélisation et de génération utilisés (par exemple l'interopérabilité).
2. Une capacité à automatiser les transformations (réponses à des stimuli d'adaptation).
3. Un ensemble de valeurs pour les transformations effectuées afin de conserver le rationnel de ces transformations.

5.1 Souplesse face aux langages

L'état de l'art montre des lacunes en termes de souplesse autour de la modélisation : on voit des systèmes dont la préoccupation est d'abord centrée sur la qualité du rendu final en ciblant un langage très spécialisé (**Huddle, Supple**). Ce qui fait la force de tels systèmes (qualité de rendu) en fait aussi la faiblesse car ils sont limités aux applications pour lesquels ils sont prévus. Aucun mécanisme de modification (ou d'ajout d'éléments) dans les langages n'est mis en place dans ces outils.

L'interopérabilité est un domaine très peu couvert : **IdealXML** permet un import des modèles réalisés avec **CTTe**. Il faut noter que le modèle de tâche **d'IdealXML** est très proche du modèle de **CTTe**. Mais de manière générale, il y a très peu de passerelles entre deux langages fondamentalement différents décrivant la même préoccupation. Notons cependant l'initiative [Bruening, et al., 2007] qui vise à promouvoir une passerelle entre langages du GL et de l'IHM.

Très peu d'outils proposent d'externaliser les heuristiques de transformations : elles sont souvent diluées dans le code de l'éditeur de modèle. Cependant les approches les plus récentes, inspirées des progrès en GL et en IDM, proposent d'utiliser des outils externes de transformation (**TransformiXML**, **Mantra**, **Emode**).

5.2 Transformation dynamique à l'exécution

Rares sont les systèmes qui offrent des transformations à l'exécution (**Teresa** et **TransformiXML**). Ils sont motivés par les besoins pour la plasticité des IHM. De manière générale les systèmes de génération automatique se sont surtout concentrés sur l'amélioration des processus de génération (en termes de qualité du rendu).

La dynamicité des systèmes interactifs implique la conception d'une infrastructure réactive. La plupart des outils de l'état de l'art ne sont que des éditeurs/générateurs. En revanche, il nous faut

considérer des outils comme **Dynamo-Aid** : s'il n'exprime pas clairement les transformations, il offre une infrastructure pour tester dynamiquement les réactions des IHM (ensemble de variantes) au changement de contexte.

5.3 Valeurs

Seules les plus récentes initiatives [Stanciulescu, et al., 2007] s'intéressent à faire porter des valeurs sur les règles de génération. Il reste encore un travail important en termes de valeur pour le choix des bonnes transformations. La conservation du rationnel des transformations, c'est-à-dire une trace liant les modèles sources et cibles (par exemple tâche liée aux espaces) est une préoccupation également récente dans les systèmes d'aide à la conception d'IHM comme [Clerckx, et al., 2004]. Cependant ce dernier ne considère pas pour le moment l'ergonomie.

Les *mappings* ou mises en correspondances servent à conserver le rationnel de conception et préserver une « valeur » globale à l'application. En revanche aucun système ne propose clairement d'identifier des valeurs sur les mises en correspondance ; ces valeurs représentent entre autres la qualité « ergonomique » d'une IHM.

Nous avons vu ici comment on peut réutiliser les avancées des systèmes basés modèles. Voyons maintenant l'état de l'art en IDM pouvant nous aider à combler les lacunes que nous avons rencontrées ici.

Chapitre III : Ingénierie Dirigée par les Modèles pour l'expression de savoirs et de savoir-faire

Ce chapitre introduit les notions nécessaires en Ingénierie Dirigée par les Modèles (IDM) pour lequel il fait office d'état de l'art. Les concepts introduits ici sont illustrés par des technologies et des outils provenant de l'IDM.

1 Le savoir en IDM : Modèles, Langages, Metamodèles

Nous présentons d'abord les origines de l'IDM, à savoir les Architectures Dirigées par les Modèles. Ensuite nous faisons une rapide revue des concepts de l'IDM ainsi que des outils servant à les mettre en œuvre.

1.1 Architecture Dirigée par les Modèles (MDA)

Bien qu'initialement ce soit le standard **MDA** (Model Driven Architecture) de l'**OMG** (Object Management Group) qui ait donné lieu à l'**IDM**, cette dernière en dépasse largement le cadre. Le « **MDA** » est un standard industriel [OMG-MDA] ayant pour point de départ la séparation des modèles selon leur indépendance aux plates-formes (Platform Independant Model ou **PIM**) *versus* les modèles spécifiques aux plates-formes (Platform Specific Model ou **PSM**). Dans le contexte de l'**OMG**, la notion de plate-forme fait référence aux intergiciels comme EJB, DotNEt, etc. L'**OMG** a donc abandonné l'idée d'une plate-forme unique et unifiée (Corba). Il faut être capable de capitaliser les savoirs indépendamment des plates-formes logicielles. Selon la vision de l'**OMG**, c'est la raison d'être des **PIM**. Ces modèles abstraits sont ensuite transformés en **PSM** via des *transformations* explicites qui prennent en compte un modèle de description de la plate-forme **PDM** (Plate-forme Description Model).

Dans le cadre de la conception des IHM, un **PIM** possible peut être un arbre des tâches décrivant, pour HHCS l'application de gestion de la température de la maison. Par exemple, un **PDM** est une description du PDA. Enfin le code de l'IHM s'exécutant sur le PDA cible est un modèle **PSM** possible.

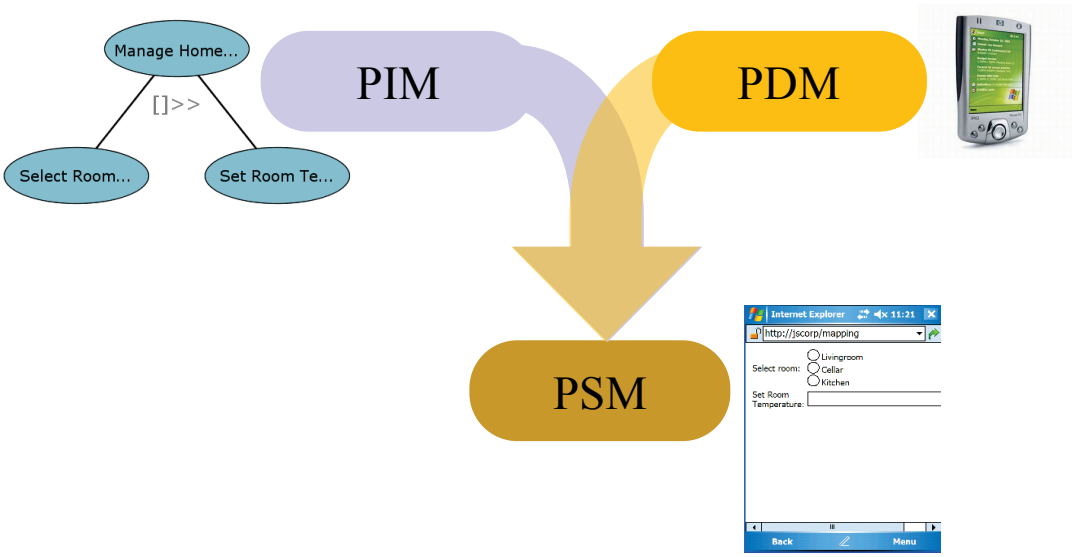

Figure 1. Processus de développement MDA : éléments principaux et artefacts sur l'exemple de la gestion de température à la maison.

Bien que l'approche préconisée par l'**OMG** soit séduisante, les standards produits par cette organisation sont souvent complexes, parfois centrés vers des domaines d'applications ou des technologies particulières comme les intergiciels.

De plus, les concepts sous-jacents sont souvent mal définis : dans notre exemple, un arbre des tâches est dit indépendant de la plate-forme si le concepteur l'a voulu comme tel. Mais il pourrait ne pas l'être, dans des environnements comme **CTTe** [Mori, et al., 2004] qui permettent de réaliser la décoration des tâches par la plate-forme ciblée. En pratique, les modèles de tâche peuvent être considérés comme **PIM** ou **PSM**. La séparation **PIM/PSM** n'est donc pas claire même dans les modèles de spécification de haut niveau.

1.2 MDA vers IDM

Nous différencions l'**IDM** ou Model Driven Engineering (MDE) en anglais du **MDA** par le fait que la séparation **PIM/PSM** n'est que conceptuelle : il faut se concentrer sur ce qui fait sens d'un point de vue Génie Logiciel. Cette distinction a fait l'objet d'un rapport de l'Action Spécifique **MDA** du CNRS [ASMDA]. Le reste de cette partie est consacré à l'étude des concepts **IDM**.

Les notions inhérentes au **MDA** n'apparaissent pas forcément comme nouvelles dans le cadre de beaucoup d'approches, tant en Génie Logiciel qu'en IHM. L'approche **MDA** se focalise sur des représentations abstraites du logiciel : les modèles. A partir de ces modèles on peut générer du code ou se servir du modèle pour raisonner : par exemple analyser les dépendances dans le code. Cette idée n'est pas spécifique au **MDA**, c'est aussi le but de la plupart des approches existantes en Génie Logiciel. Par ailleurs le terme modèle n'est pas une nouveauté en ingénierie logicielle. Les raisons du

scepticisme à l'égard du **MDA** sont détaillées par [Favre, 2004]. L'**IDM** ne se focalise pas sur un ensemble de standards et de techniques spécifiques mais se veut intégratrice de savoirs et savoir-faire en génie logiciel aussi bien que pour les domaines métiers. L'**IDM** ne remet donc pas en cause des initiatives comme XML, UML, etc. mais propose un cadre conceptuel fédérateur pour considérer ces différentes technologies. On parle alors d'espaces techniques [Kurtev, et al., 2003]. L'**IDM** considère autant la conception que l'évolution des logiciels sur tous ces aspects (sécurité, interaction, etc.).

1.3 Modèles

Il n'existe pas de définition unifiée de la notion de *modèle.* Cependant on peut trouver un consensus autour de la relation entre un *modèle* et le *système étudié* et de leur complémentarité [Bézivin, 2005]. La relation entre un système et le *modèle* étudié est notée μ [Favre, 2004] : on dit qu'un *modèle* est une représentation simplifiée d'un système. Cette représentation (*modèle*) est utilisée pour répondre à des questions à la place du système.

Par exemple un diagramme de séquence donne l'ensemble des objets impliqués dans une cascade d'appels de méthodes. Le modèle permet par conséquent au développeur de comprendre le système, sans avoir à décrypter le code. Dans l'exemple de la figure 2, à gauche, le modèle (de tâche) sert d'abstraction à l'IHM sur le PDA.

Les modèles peuvent revêtir différentes natures ; on dit alors qu'ils sont contemplatifs ou productifs selon qu'ils sont respectivement destinés à la compréhension d'un système ou bien à la production automatique de code.

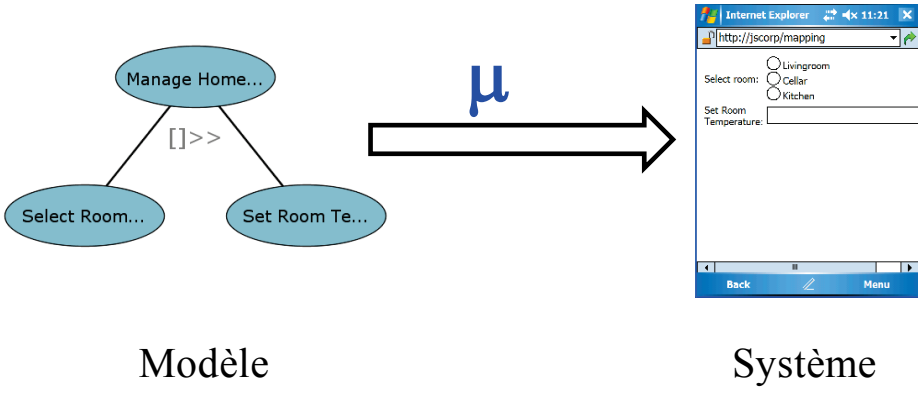

Figure 2. Un modèle de tâche représente l'IHM sur le PDA.

La notion de modèle ne se suffit pas à elle-même. Un modèle est exprimé dans un langage particulier : le langage de modélisation. Un modèle est une phrase (ou simplement un mot) de ce langage.

1.4 Metamodèles

Un *metamodèle* est un modèle de langage de modélisation. La relation qui unit un *metamodèle* à un modèle est appelée *EstConformeA* et notéχ. Le concept de *metamodèle* donne en intension un moyen de manipuler le *langage* de modélisation et ainsi de comprendre et de capitaliser les savoirs en modélisation. On utilise la relation AppartientA (ε) pour exprimer la relation entre le modèle et le langage dans lequel il est écrit.

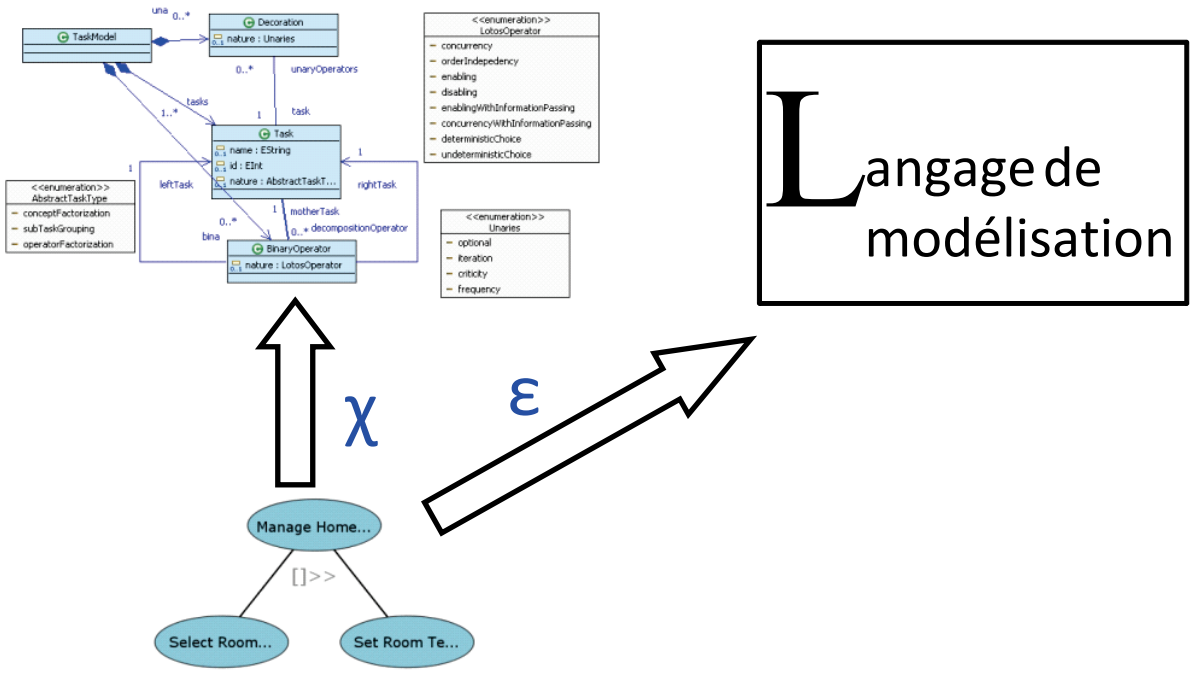

Figure 3. Un modèle de tâche conforme à un *metamodèle* de tâche. Ce *metamodèle* décrit tous les éléments et agencements d'éléments composant un modèle de tâche.

Le *metamodèle* lui-même est écrit dans un *langage*. La figure ci-dessus montre un metamodèle décrit en UML (donc lui-même conforme à UML/classe). Le *modèle* auquel est conforme un *metamodèle* est appelé *metametamodèle*.

1.5 Langages Spécifiques au Domaine

Avec l'arrivée de la *métamodélisation* en informatique, (certains domaines ont d'ailleurs comme l'IHM une longue tradition orientée *modèle*) on constate une spécialisation des savoirs se traduisant par l'émergence de langages spécifiques. On parle de Langages Spécifiques au Domaine (**LSD**) ou en anglais **DSL** (Domain Specific Language) qui traduisent une expression particulière des besoins liés aux métiers. Par exemple, en IHM on utilise traditionnellement les arbres des tâches plutôt que les diagrammes d'état UML pour exprimer l'interaction.

1.6 Metametamodèles

Un *metametamodèle* est le modèle du *langage* dans lequel le *metamodèle* est exprimé. Dans le cas de la figure 3, le langage considéré ici est UML/classe. Bien sûr cette cascade de meta pourrait ne jamais s'arrêter. On considère alors les langages capables de réflexivité, c'est-à-dire capables de se décrire eux-mêmes, comme la grammaire EBNF.

Le plus important n'est pas de trouver des niveaux absolus, comme le montre la pyramide du **MDA** (figure 4), mais de savoir de quoi on parle en relatif : est-ce un *modèle*, un metamodèle, etc. On parle alors de niveau de modélisation Mx. Si on parle de modèle Mi, le modèle Mi+1 est son metamodèle.

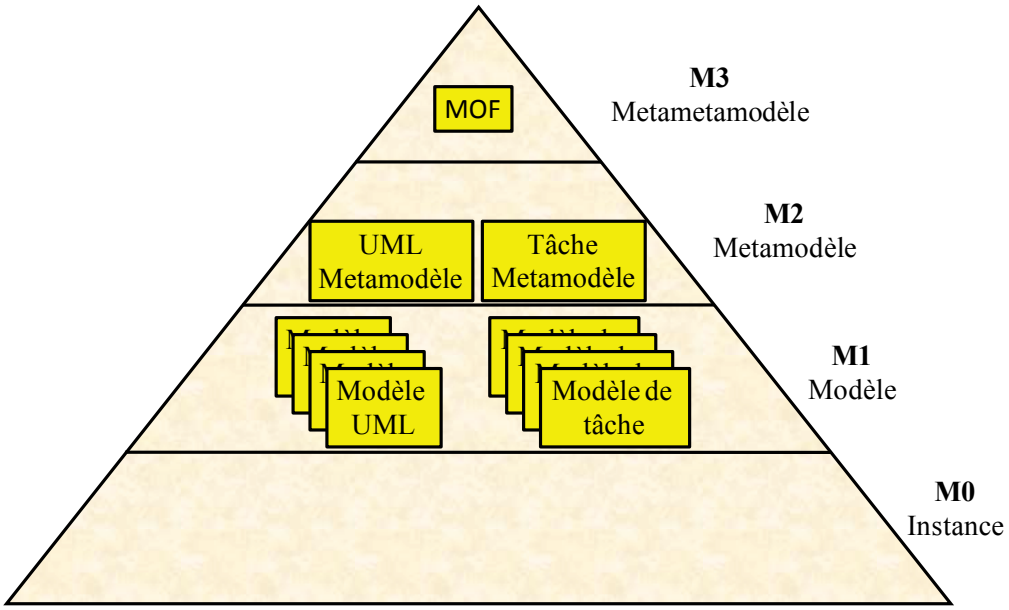

Figure 4. Les quatre niveaux de la pyramide du MDA inspiré de [Favre, 2004]

Si on reste dans une conception à quatre niveaux comme la pyramide de la figure 4, il faut tout exprimer dans un langage commun : par exemple la *MetaObject Facility* (**MOF**). La **MOF** sert de langage de modélisation commun. Elle permet la comparaison, la manipulation des modèles et metamodèles. Mais ces concepts de *metametamodèles*, metamodèles et modèles s'appliquent également aux technologies « non-**MDA** » comme les langages de programmation, les documents XML, etc. On parle alors d'espaces techniques [Kurtev, et al., 2003] dans lesquels les spécialistes, les *ingénieurs,* s'expriment de préférence comme on le fait avec notre *langue* maternelle. Un ingénieur utilise plutôt C que Java s'il a le choix dès lors qu'il a plus d'expérience dans ce langage.

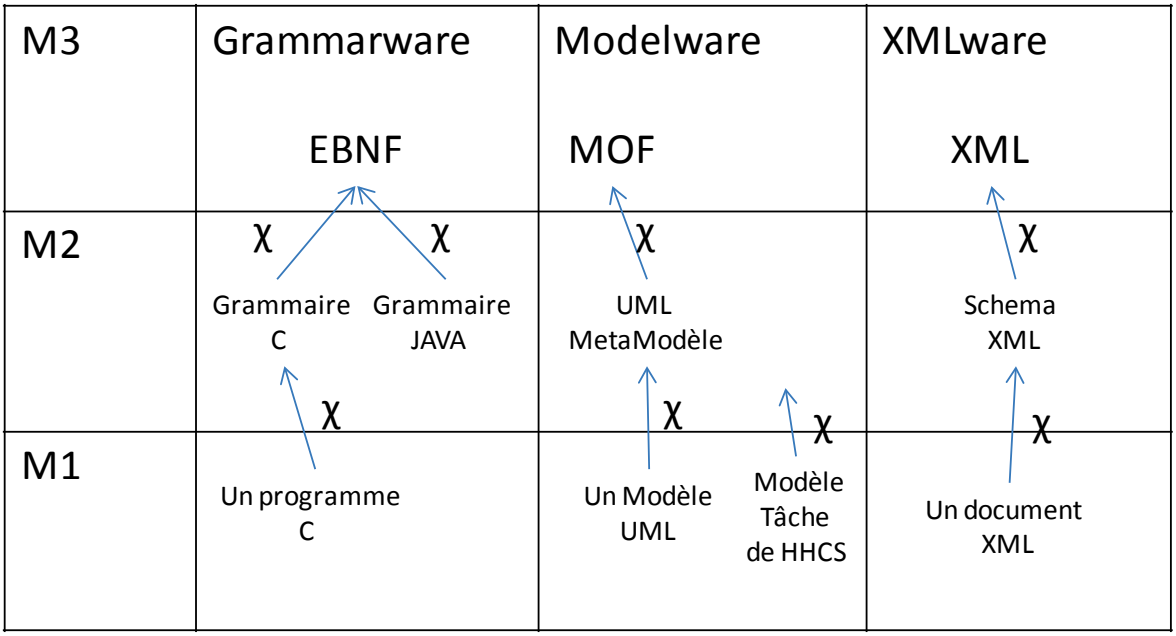

Tableau 1. Différents espaces Techniques : Grammaire (EBNF), Document (XML), IDM (MOF).Tableau inspiré du rapport de l'AS MDA [ASMDA].

Le tableau 1 présente les concepts du MDA (niveaux de modélisation et relations entre modèles). Ce tableau est une vue simplifiée afin de comprendre le principe d'espaces techniques. Un document XML est aussi directement conforme à XML : il y a donc ambigüité possible entre les M2, M3 et M1. En outre chacun de ces espaces n'est pas un îlot isolé mais peut être relié par des transformations : par exemple de java vers XML grâce à la sérialisation.

1.7 Standards et outils de metamodélisation

L'**OMG** a défini la **MOF** [OMG-MOF] comme langage standard de *metametamodélisation*, c'est-à-dire un langage commun à tous les langages de modélisation. La **MOF** définit un ensemble de blocs de base (Element, Factory Class, Instance, etc.) et de propriétés (réflexivité, extensibilité) pour concevoir des langages ou *metamodèles*. Les éléments déterminés par la **MOF** sont très proches d'UML et des concepts de l'Objet plus généralement.

Eclipse Modelling Framework (**EMF**) est une implémentation JAVA qui se veut proche de **MOF** [EMF]. **EMF** fournit les outils pour créer des metamodèles, générer des éditeurs de modèles correspondants, ainsi que le code Java. La persistance des modèles est assurée par un fichier **XMI** (*XML metadata interchange*) spécifique à **EMF**.

DSL tools [DSL-TOOLS] est à Microsoft ce qu'**EMF** est à IBM ou plus précisément au consortium Eclipse. C'est une implémentation qui permet de créer ses propres éditeurs de modèles et de

générer du code. L'acronyme **DSL** fait référence au concept de langage spécifique au domaine (section 1.5) qui a été vu précédemment.

Le *MetaData Repository* (**MDR**) est une implémentation des recommandations du **MOF** [MDR]. Il consiste en un réceptacle de métadonnées (écrites en **XMI**, un fichier XML standardisé pour transférer des données) afin d'assurer la persistance des modèles. MDR est entièrement compatible avec le standard **JMI** de Sun et il est intégré dans NetBeans.

Kermeta [kermeta] est un langage de *metametamodélisation* développé à l'Université de Rennes. Dans Kermeta, un langage est aussi défini pour écrire des transformations, ainsi que nous le verrons plus tard. Il dispose d'un environnement de développement de metamodèles basé sur **MOF** dans un environnement Eclipse. Il permet non seulement de décrire la structure des metamodèles, mais aussi leur comportement.

MOFLON [MOFLON] est une initiative de l'université de Darmstadt (Allemagne) pour unifier le monde des *transformations de graphes* avec celui de l'IDM. Le *metametamodèle* utilisé par ce système est basé sur **UML** 1.x auquel on peut ajouter un ensemble de contraintes.

GME (Generic Modeling Environment) [GME] est un environnement permettant de créer des modèles spécifiques à un domaine. **GME** est développé par l'université Vanderbilt (USA). L'élaboration de *metamodèles* est axée sur un diagramme de classes UML et de contraintes OCL.

Kernel Metametamodel (**KM3**) développé par l'université de Nantes est un langage de *metametamodélisation* se basant aussi sur les concepts de la **MOF**. **KM3** est une syntaxe textuelle permettant de définir des metamodèles sur la base de « déclaration de classe java ». Elle ne conserve que les classes, les références et les attributs. Elle sert entre autres de modèle pivot entre **EMF** et **DSL** de Microsoft [Bézivin, et al., 2005].

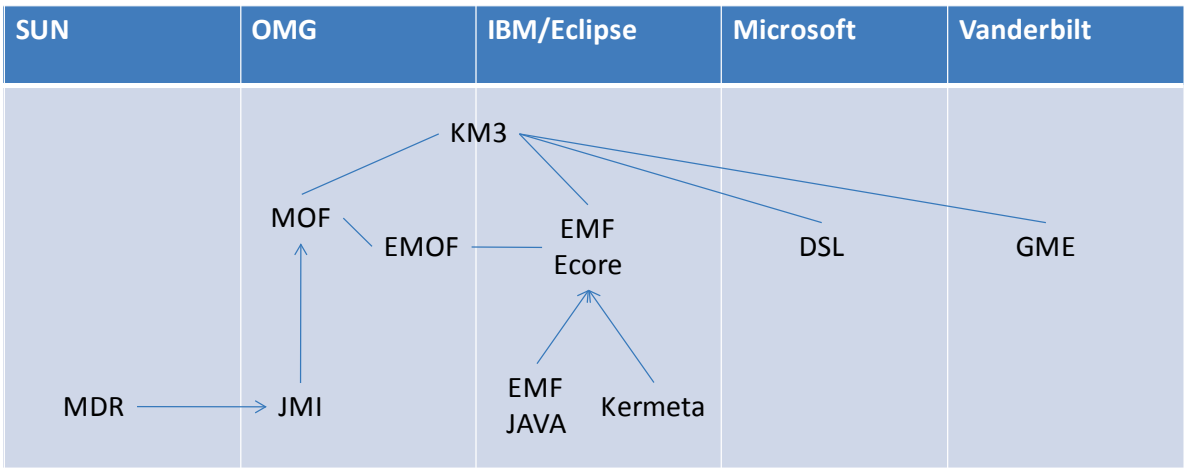

Tableau 2. Résumé des outils de metamodélisation.

Le tableau 2 propose un graphe des relations (« repose sur pour les flèches et pont technologiques pour le traits) entre les outils précédemment définis et leur affiliation. Notons que KM3 est transversal à tous ces clivages et permet d'établir des ponts entre ces différentes implémentations.

2 Savoir-faire en IDM : Transformations

Le cœur de l'**IDM**, d'abord implicite dans le **MDA**, repose sur la représentation explicite des transformations.

2.1 Transformation

La notion de transformation a mis du temps avant d'être une notion de premier ordre en **IDM** [Gerber, et al., 2002]. C'est grâce aux transformations que l'on peut passer de **PIM** à **PSM** pour le **MDA**. Selon [Mens, et al., 2004], une *transformation* peut être vue comme un ensemble de règles qui décrivent comment un ou des modèles sources sont transformés en modèles cibles. Les règles d'une *transformation* se basent sur les *metamodèles* (ou *langages*) dans lesquels sont exprimés les modèles. Une *transformation* reprend donc les « mots » du *metamodèle* source pour lui faire correspondre des « mots » du *metamodèle* cible. De la même manière que les *modèles* ne sont pas considérés comme une nouveauté en informatique, la notion de *transformation* n'est pas née avec l'**IDM**. En effet, les compilateurs de code, XSLT pour XML sont des illustrations de transformation hors de l'espace technologique du « *modelware* ».

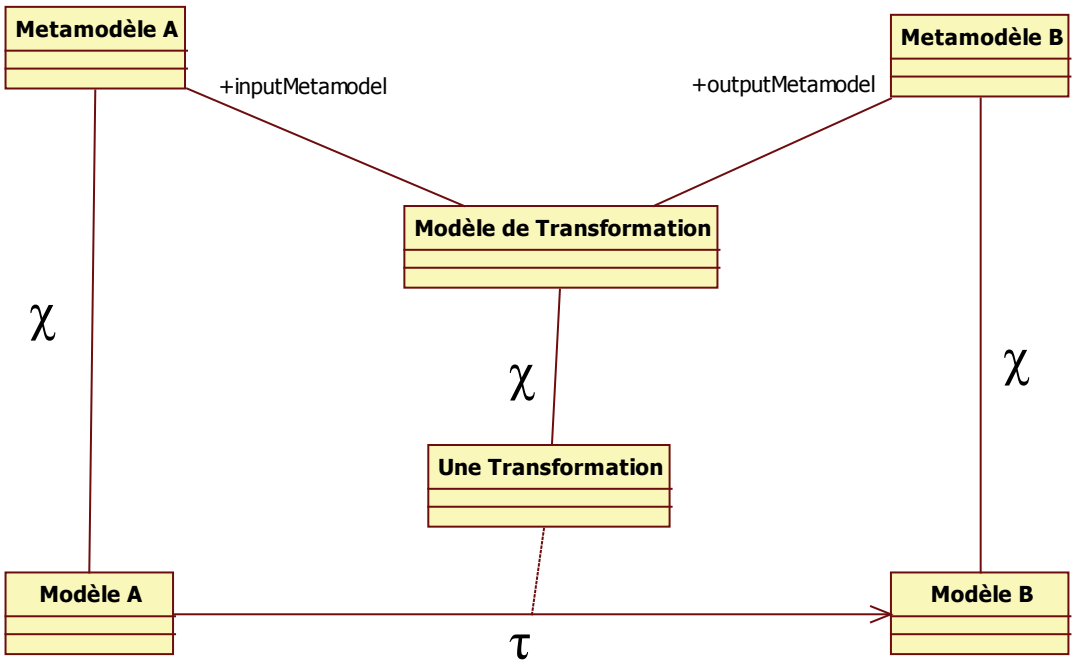

Figure 5. Une transformation est écrite pour un ensemble de metamodèles sources A vers un ensemble de metamodèles cibles B. La relation « est transformée en » est notée τ.

On identifie deux types de transformations : modèles vers modèles et modèles vers code (ou modèles vers texte et inversement). Si le principe peut sembler le même pour ces deux cas, il convient de différencier le fait de rester dans une même technologie et le transfert vers un autre espace de technologie. Par exemple XSLT permet deux types de transformations : de XML à XML et de XML à texte. Dans ce deuxième cas, on peut réaliser une transformation de la technologie XML à la technologie Java.

Aujourd'hui, les technologies ne peuvent pas évoluer en autarcie : la conception d'un site internet peut déjà faire appel à de nombreuses technologies différentes : flash, javascript, HTML, Ajax, etc. Il faut donc être capable de donner des ponts et des mises en correspondance pour les différentes technologies étudiées. Il est intéressant de proposer des mécanismes similaires pour passer d'une version à l'autre, d'un type de modélisation à un autre etc. ceci dans le but de conserver les spécificités de chacune des technologies ainsi que les savoir-faire des ingénieurs liés à des technologies particulières.

2.2 Transformation de transformations

Les transformations de modèles sont un moyen de passer d'une technologie à l'autre, mais elles sont elles-mêmes écrites dans une technologie particulière : les transformations sont aussi des modèles. On parle de transformation d'ordre supérieur (High Order Transformation, **HOT** en anglais) pour faire

référence aux transformations qui génèrent des transformations. Ce concept n'est pas nouveau, il est présent dans les compilateurs pour l'optimisation de code par exemple : ces transformations de plus haut niveau offrent alors des algorithmes de compilation spécifiques au code à traiter.

Les transformations d'ordre supérieur sont particulièrement utiles pour couvrir *l'évolution*, les variations ou les traitements spécifiques qui peuvent être faits à la volée lors de l'exécution d'une *transformation*. On va par exemple utiliser des transformations de transformations dans le cadre d'un changement de version : on est capable, à partir de la différence entre deux metamodèles (version1 et version 2), de générer les transformations permettant de convertir tous les modèles version 1 en version 2.

2.3 Standards et outils de transformation de modèles

L'**OMG** a émis un ensemble de spécifications pour les « langages de transformation de modèles » : le standard **QVT** (Query, View, Transformation) [OMG-QVT]. Il définit plusieurs types de syntaxes du langage : déclarative, impérative, hybride (QVT-relation, etc.) ainsi qu'une sémantique (QVT-Core). Actuellement les outils de transformation de modèles implémentent une partie de ces spécifications.

l'Atlas Transformation Language (**ATL**) développé à l'université de Nantes propose un dérivé des standards, en outre c'est un composant du projet Eclipse Model2Model [Eclipse-M2M]. Par ailleurs **ATL** ne permet pas dans l'implémentation actuelle de développer des transformations réversibles comme le suggère la norme **QVT**. Cependant, **ATL** permet l'élaboration de requêtes (Query) à travers lesquelles on peut faire un ensemble de vérifications sur le modèle. **ATL** est un langage très utilisé, à la fois dans le monde académique et industriel, et il tire parti d'une syntaxe proche d'OCL. **ATL** s'appuie sur les *metametamodèles* **EMF** et **MDR**.

Medini QVT [QVT] est développé par IKV++ et se veut une implémentation du standard QVT-relation (syntaxe déclarative). Il permet la définition de transformations itératives, l'élaboration d'une trace et la définition de transformations bidirectionnelles.

Kermeta : [Kermeta] Dans l'environnement de modélisation **Kermeta** il y a aussi des transformations de modèles. La syntaxe de transformations en **Kermeta** est proche de **JAVA** et dispose d'une structure uniquement impérative pour écrire des transformations.

Viatra2 est un moteur de transformation de modèle qui se dit compatible avec **QVT**. Le but de **Viatra2** est de rendre les méthodes formelles « invisibles » en les cachant dans des transformations de modèles automatisées [Horvath, et al., 2006]. Ce langage inclut deux formalismes pour exprimer ses transformations de modèles : les transformations de graphes et les machines à états abstraites.

Dans le domaine des transformations de graphes, il y a beaucoup de travaux antérieurs au **MDA** et à l'IDM. Cependant des études visent à concilier les *Triples Grammaires de Graphes* **TGG** (utilisées pour décrire des transformations de graphes) avec le standard **QVT** [Greenyer, 2006].

Les **TGG** sont utilisées dans l'outil **MOFLON** et définissent des transformations bidirectionnelles de manière déclarative (comme le préconise le standard **QVT**). En outre **MOFLON** utilise un autre langage de transformation (**SDM)**, de nature impérative et qui décrit les transformations à l'aide de diagrammes d'activités.

2.4 Mises en correspondance

Les mises en correspondance sont proches dans leur définition des transformations de modèles : elles lient des éléments de modélisation et définissent par ce biais des transformations de modèles.

Cependant les mises en correspondance font plus que décrire des transformations. On veut lier deux modèles pour obtenir une troisième entité qui possède de nouvelles propriétés : on parle alors de tissage de modèles ou de metamodèles. Une mise en correspondance peut être une trace de transformation. Cette trace propose alors le lien entre le modèle source et le modèle cible de la transformation. Elle permet également de voir quelles règles sont utilisées pour réaliser cette transformation.

Les mises en correspondance sont, avec les modèles, un autre moyen d'exprimer des points de malléabilité ; cette fois-ci, non plus sur les modèles eux-mêmes mais sur la relation entre les modèles, et ceci à tous les niveaux de modélisation. On peut ainsi réaliser une « cascade » de transformations à partir des mises en correspondance pour, par exemple, assurer la cohérence d'une application lors de sa modification sous malléabilité.

Dans **AMW,** [Didonet Del Fabro, et al., 2005] les mises en correspondance ou tissages expriment différents scenarii : la *comparaison entre metamodèles, la traçabilité, l'annotation de modèle et l'interopérabilité* entre *modèles*. Un metamodèle de mise en correspondance est fourni dans **AMW**, ainsi qu'un mécanisme d'extension de ce metamodèle pour couvrir de nouveaux scénarii. Le plugin **AMW** procure également tout l'outillage pour réaliser des mises en correspondance et générer les transformations **ATL** qui sont décrites dans le modèle de mise en correspondance.

MWDL (Milewski, et al., 2005) est un langage de description de tissage de modèles. Il permet la mise en correspondance entre différents aspects des modèles. Grâce à cette description, les auteurs simplifient la réalisation (en utilisant des modèles de grains très fins) et la réutilisation de transformation.

Le tisseur de modèle **WEAVR** [Cottenier, et al., 2007], initiative de Motorola, est un module à Telelogic **TAU** [Telelogic, 2005] permettant de décrire des profils d'Aspect pour UML. On peut également visualiser et proposer des tissages. Ces derniers, semblables aux *tissages* d'*aspects*, définissent directement dans le modèle à tisser, un ensemble de points d'action ou de jonction portant les conditions d'activation des *aspects*. Les actions portées par ces aspects sont modélisés dans **WEAVR** sous forme de graphes états-transitions. Contrairement aux tisseurs de modèles précédents, **WEAVR** est aussi basé sur la dynamique des mises en correspondance.

C-SAW [Gray, et al., 2001] est un cadre de travail pour le tissage de modèles, basé sur des contraintes et fourni dans l'outil **GME**. L'idée de base de **C-SAW** est de faire ressortir les contraintes croisées dans un ensemble de modèles : l'utilisateur de **C-SAW** choisit une transformation correspondant aux scenarii de tissage qu'il désire réaliser. Ce framework utilise donc les mêmes concepts qu'**AMW** : tissage de modèles et génération de transformation. Cependant cette dernière transformation peut être réalisée par un ensemble d'aspects avec **AspectJ**.

Le « **constraint based-modelling** » [White, et al., 2008] propose des mises en correspondance en s'inspirant des tissages de la programmation par aspects. Appliqué aux modèles, on parle de modélisation par aspect (Aspect Oriented Modelling **AOM**). Il autorise le tissage de différents modèles (qui sont des perspectives) par l'intermédiaire d'un modèle composite. L'originalité de ce travail réside dans l'utilisation d'un résolveur de système à contraintes pour choisir la mise en correspondance la plus appropriée.

Basé sur l'outil **GME,** le **constraint based-modelling** facilite l'enchaînement les outils de transformation proposés (**C-SAW**, **AMW**, **WEAVR**)**.** Il prend en compte ce qui manque à ces outils de tissages : le support des contraintes globales du système.

Transformations et mises en correspondance lient concepts, outils et modèles en permettant le passage d'une technologie à une autre. C'est la dernière dimension qui manquait pour introduire le concept de *megamodélisation* : unification de tous les éléments de la modélisation.

3 Megamodélisation

La *megamodélisation* pourrait se résumer à mettre en relation l'ensemble des concepts de l'IDM autour d'un même modèle unificateur. Mais cela ne se limite pas à un modèle où ne figurent que les concepts de « metamodèle, modèle, langage et systèmes » avec leurs relations respectives : τ, χ, μ. Il faut également prendre en compte l'ensemble des artefacts ou instances produits pour chacune de ces entités et relations.

3.1 Megamodèles

Nous proposons un *megamodèle* simple, représentant les éléments et relations de base qui le composent. Ce premier *megamodèle* (figure 6) sert aussi de *metamodèle* pour organiser les éléments de modélisation d'un domaine donné (modèle, metamodèle etc.). Bien sûr, ce *megamodèle* étant lui-même un (meta)modèle, il peut se définir par lui-même.

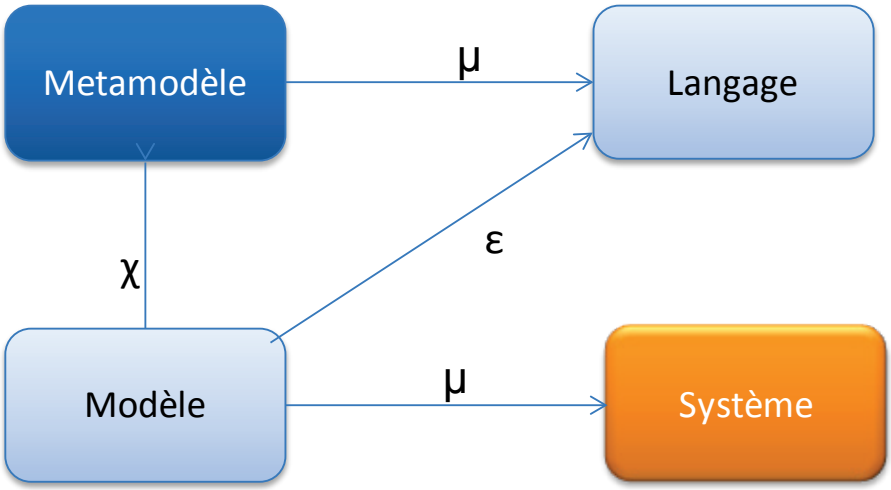

Figure 6. *Megamodèle* mettant en relation (μ, χ, ε) les concepts de base en IDM.

Par exemple, si nous projetons cette notion générale, de megamodèle, sur le domaine de l'IHM, et plus particulièrement sur le modèle de tâche, nous obtenons le graphe synthétique de la figure 7 (reprenant les figures des parties précédentes).

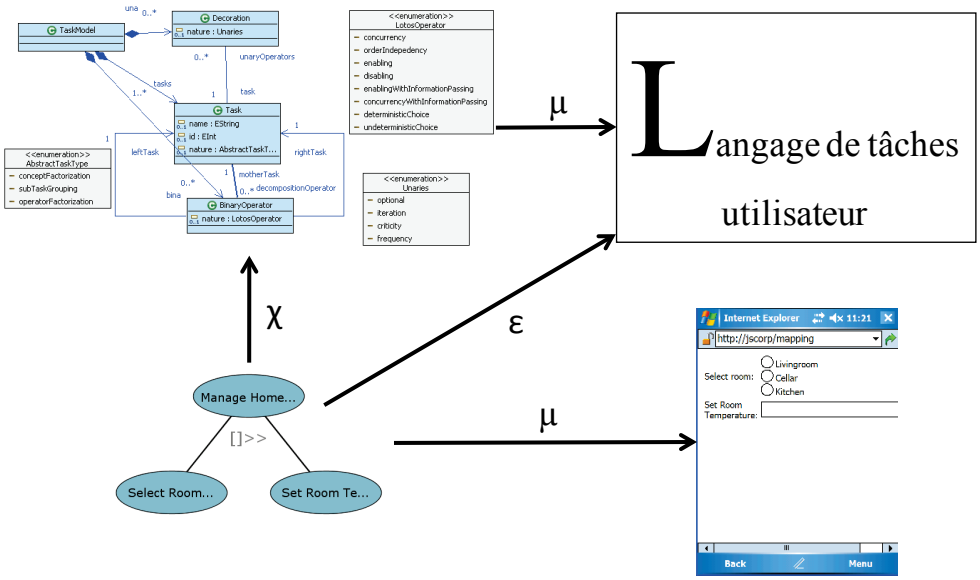

Figure 7. *Megamodèle* appliqué au *modèle* de tâche de l'IHM, conforme au *metamodèle* de *megamodèle* de la figure 7.

A ce graphe de *megamodèle,* il manque encore la relation *estTransforméEn* (τ). L'instanciation de ces transformations dans un *megamodèle* peut représenter les différents processus d'ingénierie appliqués à ce domaine. Par exemple, en IHM, nous avons une succession de transformations qui partent de l'arbre des tâches pour générer l'IHM de l'application. Un *megamodèle* peut ici représenter l'ensemble des éléments, des relations d'un domaine particulier ; en d'autres termes il est le *modèle* des *savoirs* et *savoir-faire* pour ce domaine.

Les notions *d'espaces techniques* et de *langages* sont également prédominantes avec des questions pratiques : comment passer de C à java, de java à .Net, etc. Encore une fois c'est la relation (τ) qui permet d'aller d'un *langage* à un autre pour lier différentes perspectives métiers (Base de données avec l'IHM). On parle alors de *Zoo* de *metamodèles* et de *transformations*. Ces zoos sont eux-mêmes des *megamodèles* plus ou moins larges couvrant plusieurs ponts entre *langages*.

3.2 Les plates-formes

Nous considérons ici tous les systèmes qui répondent aux ensembles d'éléments proposés, à savoir metamodélisation, modélisation, transformation et tissages de modèles (ou mises en correspondance).

	OMG (spécifications)	AMMA (Inria/Nantes)	MOFLON (fujaba)	GME (Vanderbilt)	DSL Tools (Microsoft)
MetaModèle	MOF	KM3	UML1.X	GME 5.0	MS-DSL
Transformation (Model to Model)	QVT	ATL	TGG (déclaratif) SDM (impératif)	GReAT	Pas de technique explicite !
Transformation (Model to text)	QVT	ATL (Query) TCS	Velocity	?	MS Templates
Tissage		AMW	?	C-SAW	

Tableau 3. Couverture des plates-formes en termes d'éléments et de relations du *megamodèle.*

Les quelques systèmes et normes présentés dans le tableau 3 présentent les notions (éléments et relations) proposées dans le *megamodèle* pour l'IDM. Ces outils couvrent l'élaboration de

metamodèles, de modèles et la description de modèles de transformations. Il faut cependant noter que la plate-forme **AMMA** propose un module explicite (**AM3**) de gestion de *megamodèles*.

4 Synthèse

Nous avons vu dans ce chapitre quels sont les concepts et technologies que nous pouvons mettre en œuvre pour exprimer les savoirs et savoir-faire en informatique. C'est par le biais du concept de modèle (dont la structure est clairement exprimée dans un metamodèle) que nous pouvons formuler et manipuler les savoirs. Les transformations quant à elles nous permettent de préciser et de manipuler les savoir-faire.

Nous avons tous les points de contrôle (modèles et transformations) qui nous permettront de mettre en œuvre la malléabilité : nous pouvons manipuler savoirs et savoir-faire. Les mises en correspondance et règles de transformations nous servirons ici pour exprimer les plages de valeurs limites de la malléabilité.

Nous voyons enfin qu'il y a matière à travailler avec la notion de *megamodèle*. Des initiatives récentes visent à regrouper les connaissances en termes de modèles [(Zoomm), (AtlanticZoo)] metamodèles et transformations. L'importance est donnée aux cartographies des savoirs et savoir-faire d'un domaine précis : *megamodèle*. Ces initiatives font la promotion de passerelles entre ces îlots de connaissances.

Dans le cadre de ce travail, il nous reste maintenant à montrer comment appliquer l'IDM à l'IHM (**chapitre IV**) et motiver la création d'une cartographie (megamodèle) du domaine de l'ingénierie de l'IHM (**chapitre V)**.

Chapitre IV : Extraction des savoirs et savoir-faire à partir d'outils

Au chapitre précédent nous avons vu comment exprimer les savoirs et savoir-faire avec les notions IDM : *modèles, langages metamodèles et transformations*. Dans ce chapitre, nous montrons comment appliquer ces notions au domaine de l'IHM. Nous présentons l'intérêt d'une telle démarche pour extraire les savoirs et savoir-faire mais aussi pour pouvoir comparer des outils/modélisations.

Dans un premier temps, nous voyons comme extraire les savoirs des outils (du **Chapitre II**) : réaliser des metamodèles à partir de l'extraction. Ceci est illustré par un exemple (section 1.1) et un début de méthodologie pour extraire des metamodèles (sections 1.2 et 1.3).

Dans un deuxième temps, nous comparons les outils au travers des metamodèles que nous avons préalablement extraits. Cette comparaison donne lieu à une généralisation.

Enfin, la dernière partie « ouvre » notre approche vers l'extraction de savoir-faire : les transformations de modèles. Les règles de transformation d'un outil, **IDEALXML**, illustrent cette partie.

1 Extraction de savoirs

Dans un premier temps, nous examinons une manière d'extraire les metamodèles utilisés par les outils proposés dans l'état de l'art. Nous faisons ensuite les liens entre les différentes parties des IHM des outils et les éléments composant le metamodèle du modèle construit. Tout au long de cette partie nous traitons l'exemple de **CTTe** pour illustrer une méthode d'extraction.

1.1 Cas d'étude : Extraction du metamodèle de CTTe

Nous étudierons, ici, le metamodèle du langage CTT tel qu'il est proposé par l'outil **CTTe**. Cette extraction de metamodèle est appelé une « retro-conception dirigée par les modèles ». Ce type d'approche est proposé dans **Modisco** (Model Driven Reverse Engineering) dont le but est d'extraire des modèles et metamodèles à partir de systèmes propriétaires (*legacy*).

1.1.1 « Tâches »

Tout d'abord nous étudions la partie édition principale de **CTTe** : elle permet d'ajouter des tâches, des opérateurs entre tâches et de définir par construction une structure arborescente.

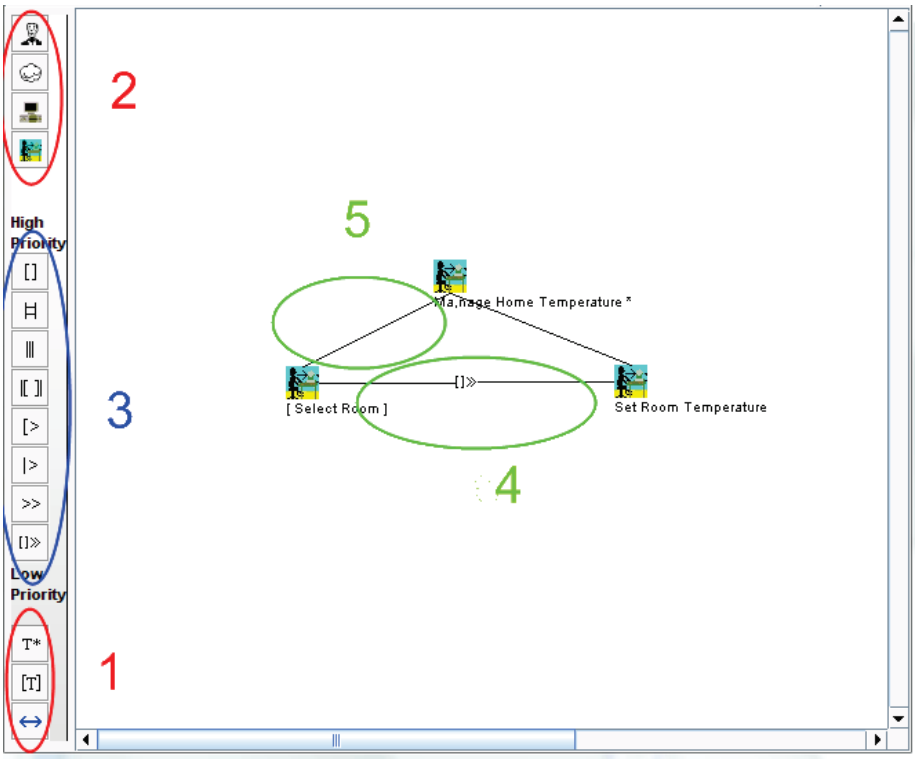

Figure 1. Partie édition de l'arbre des tâches dans **CTTe** : éléments principaux présentés en 1, 2, 3 (tâches et opérateurs binaires et unaires) ; éléments de structure présentés en 4 et 5.

Les éléments identifiés dans la figure précédente nous permettent de commencer à élaborer des hypothèses de retro-conception. Nous identifions trois énumérations (modélisant des collections finies d'éléments) correspondant aux zones 1, 2 et 3. Ces zones définissent respectivement des types d'opérateurs unaires (sous le libellé « *Low Priority* ») à appliquer aux tâches (appelées aussi décorations) : optionnel, itératif ou de connexion, des *catégories* de classes et enfin des *opérateurs binaires* (sous le libellé « *High Priority* ») entre ces tâches.

A partir des éléments identifiés en 4 et 5 dans la zone de travail principale de l'éditeur, on déduit la structure de l'arbre CTT. La décomposition d'une tâche en sous-tâches (cercle n°5 sur la figure 1) conduit à une relation (*subTask*) sur la classe *task* : une tâche peut avoir de 0 (tâche feuille) à N sous-tâches.

Par construction, on pose des opérateurs entre tâches d'un même niveau de décomposition. Nous suivons la méthodologie de **CTTe** qui indique que si une tâche est décomposée, elle l'est forcément au moins par deux sous-tâches qui sont reliées entre elles par des opérateurs binaires. Cette relation semble séquentielle : de la tâche gauche vers la droite. Nous réalisons donc une classe opérateur (*Operator*) avec une relation orientée : tâche gauche vers opérateur puis opérateur vers tâche droite.

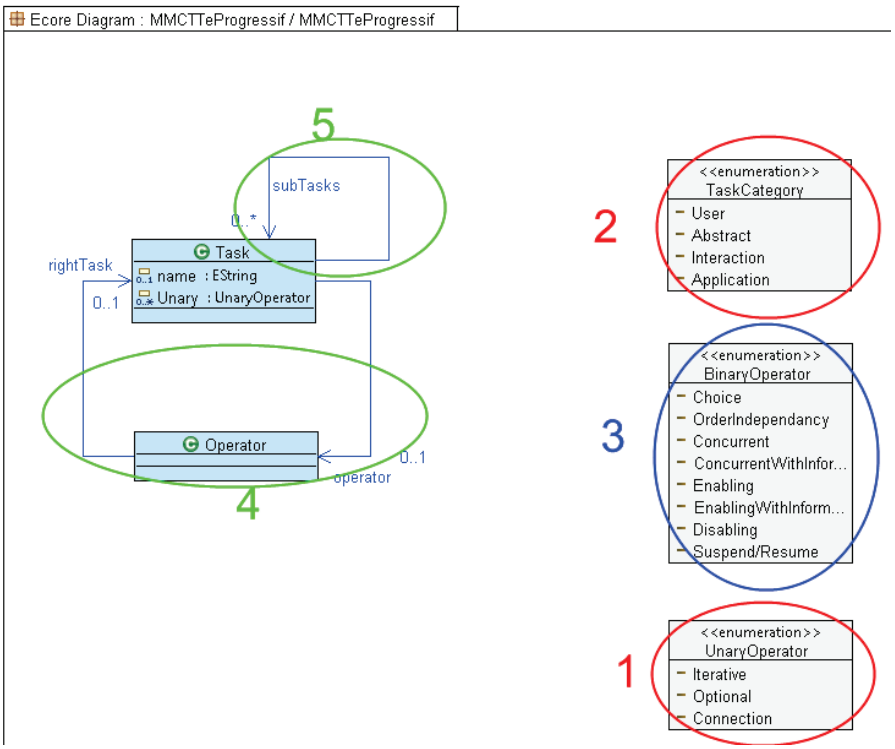

Figure 2. Hypothèses sur l'extraction du metamodèle des différents éléments de la figure 1.

Ce metamodèle, certes simple, nous permet de décrire une première version d'arbre des tâches qui correspond à ce qu'on peut construire avec seulement l'interface de la figure 1. Cependant **CTTe** propose d'autres fonctionnalités : la définition d'attributs pour une tâche, des objets, des performances proposées à l'utilisateur dans la fenêtre « *task properties* » de la figure 3.

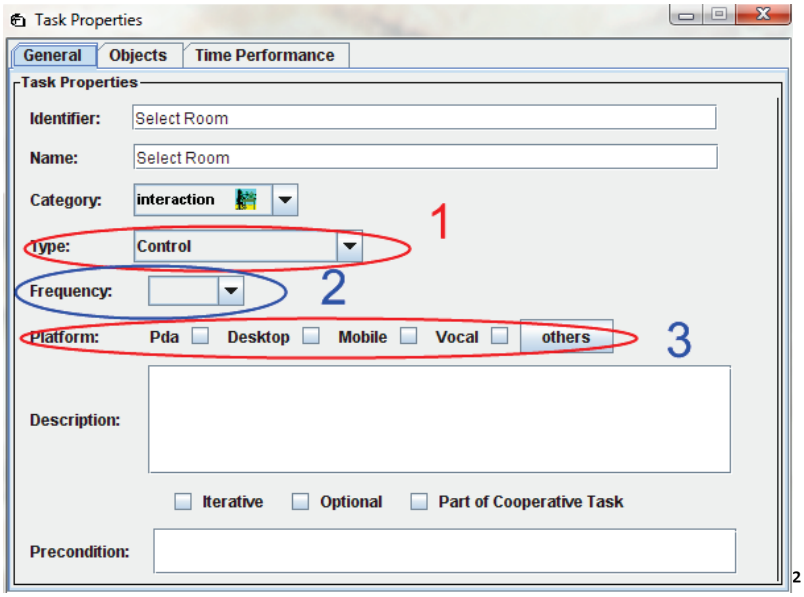

Figure 3. Fenêtre d'édition d'une tâche (« *task properties* »).

En choisissant une catégorie de tâche, on accède à un ensemble de valeurs pour le type de tâche (figure 3 point 1). Ces valeurs correspondent à une sous-catégorie : ceci nous amène à reconsidérer notre modélisation précédente (figure 2) : la catégorie de tâche ne peut plus être modélisée par une simple énumération de valeurs car des informations sont associées à cette catégorie. On développe alors une hiérarchie de classe avec un héritage : les catégories deviennent des classes particulières. Les types de tâches (Point 1 de la figure 3) deviennent des sous-classes d'interaction : ils se retrouvent ici dans les classes numérotées 1 de la figure 4. Dans ce diagramme, nous n'avons pas représenté les éléments « *identifier* », « *description* » et « *précondition* » mais ils font partie du modèle.

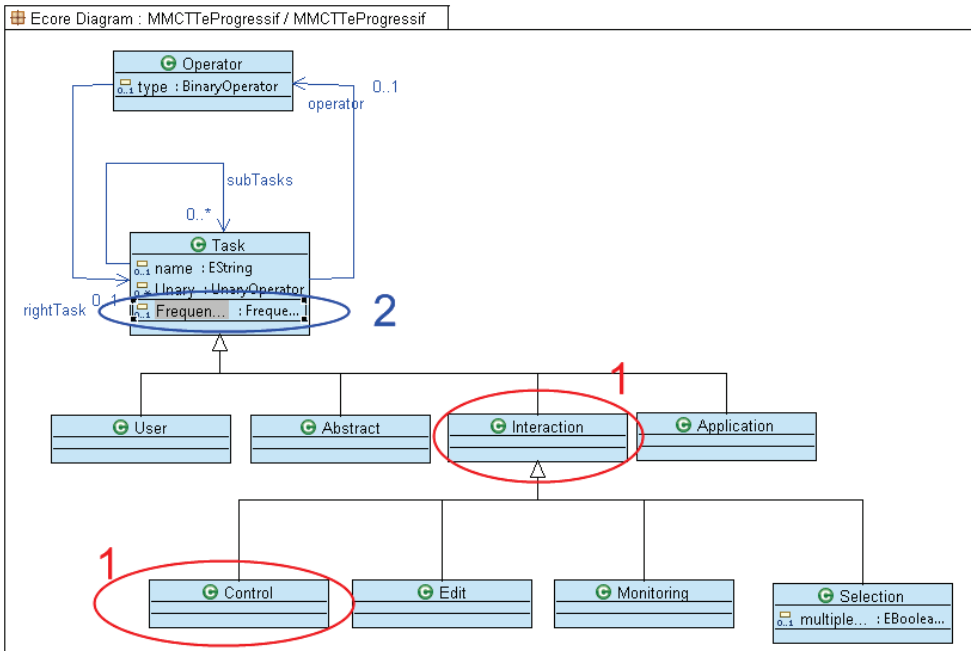

Figure 4. Modèle corrigé des catégories et sous-catégories de tâches. Ici seule la catégorie interaction est développée.

1.1.2 « Plates-formes »

La place de la plate-forme pour **CTTe** est particulière : elle permet de filtrer les tâches pouvant s'exécuter sur un type de plate-forme. Le type de plate-forme est extensif (le bouton « other », figure 3, permet de créer son propre type). Les radio-boutons correspondent alors à des booléens : une manière simple de modéliser ces booléens est de réaliser pour chaque type de plate-forme un attribut dans la classe tâche. C'est la manière la plus directe de représenter ce type d'interacteur dans le metamodèle.

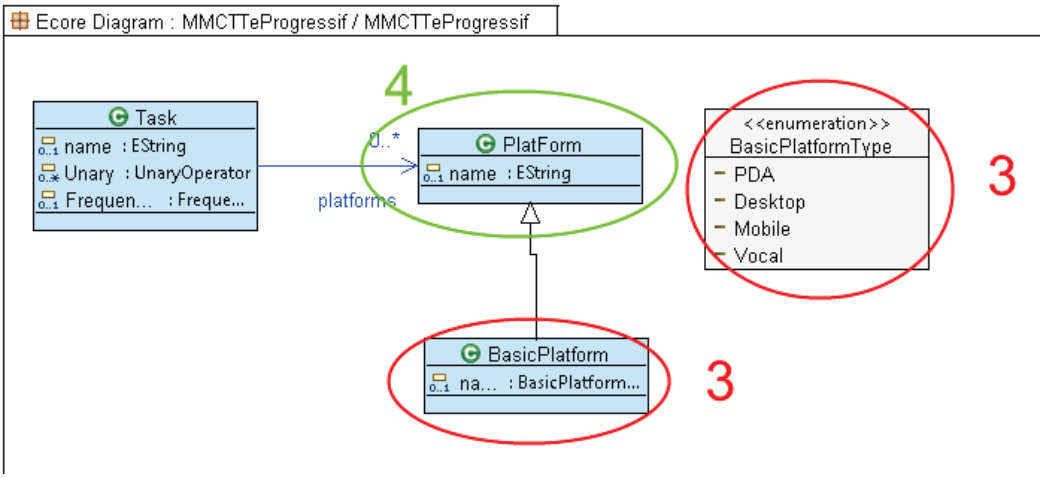

Figure 5. Vue de la relation tâche-plate-forme. Le point 3 représente les plates-formes de base, le point 4 les plates-formes éventuellement ajoutées par l'utilisateur (figure 6).

Une autre façon de modéliser cela consiste en une classe *plate-forme* (figure 5). Cette fois-ci, cocher une case revient à réaliser la relation entre une instance de la classe tâche (*task*) et une instance de la classe plate-forme (*platform*) qui possède un nom (son type). Un nouveau type de tâche n'est ici défini que par une chaîne de caractères (voir figure 6).

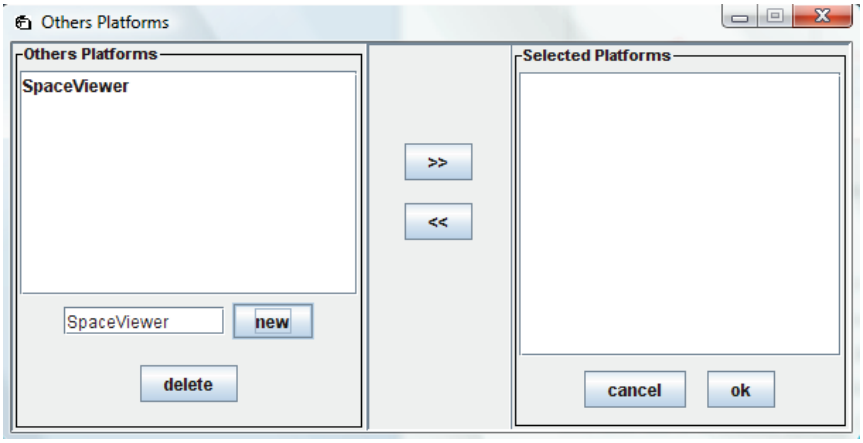

Figure 6. Création d'une nouvelle plate-forme « SpaceViewer », identifiée uniquement par son nom.

1.1.3 « Objets » du domaine

L'opération de retro-conception est à répéter pour les concepts du domaine appelés objets dans **CTTe** (voir onglet « objects » de la figure 7). Ces concepts sont, comme les plates-formes, liés aux tâches car ils sont présents dans la fiche d'édition d'une tâche particulière (figure 3).

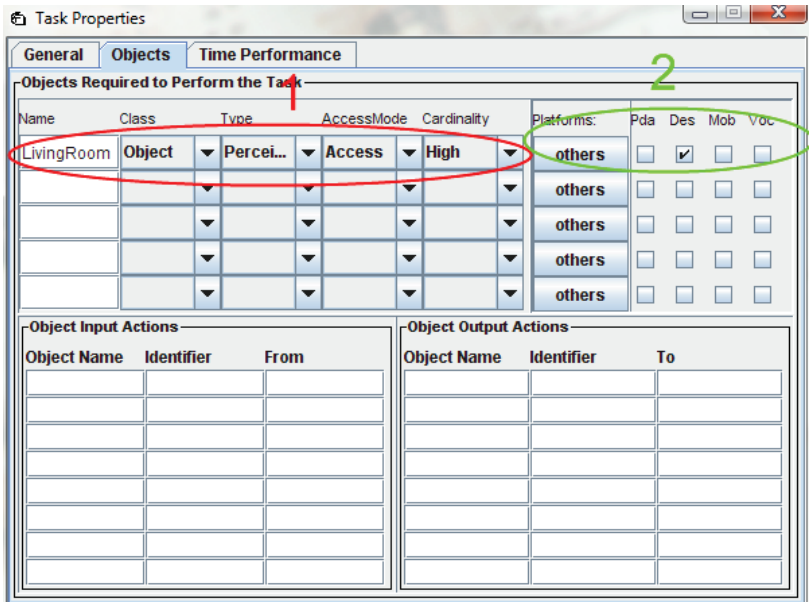

Figure 7. Onglet de gestion des « objets » (concepts) du domaine.

La copie d'écran de la figure 7 montre une zone d'édition (zone cerclée 1) « d'objets » du domaine modélisé. On peut alors éditer un ensemble d'objets en donnant leur nom, leur classe à choisir parmi : Objet (générique), texte, entier, etc. (types primitifs). Cette partie ne permet pas la définition précise des classes de ces objets : par exemple, le fait de savoir qu'une pièce a une température, une luminosité, etc. En revanche, on peut définir plusieurs types : l'objet est perceptible et fera parti de l'IHM ou, au contraire, uniquement un modèle de l'application. Enfin le type d'accès par l'IHM est lui aussi défini : soit en affichage (*access*), soit en modification (entrée de valeurs). La cardinalité, quant à elle, ne définit pas une valeur précise mais un ordre de grandeur du nombre d'objets manipulés par rapport à la tâche.

Ceci conduit à une nouvelle classe objets (figure 8) dans notre metamodèle qui possède comme attributs tous les champs à remplir de la zone cerclée 1.

La partie cerclée numérotée 2 comprend les plates-formes pour lesquelles, les concepts sont valables à l'instar de ce qui a été proposé pour les tâches. Nous créons donc une association liant un objet (*Object*) à une plate-forme (figure 8, zone n°2). Enfin la zone cerclée numéro 3 entoure la relation d'une tâche vers N objets (classe *Object*), probablement limitée à 5 d'après l'interface figure 7.

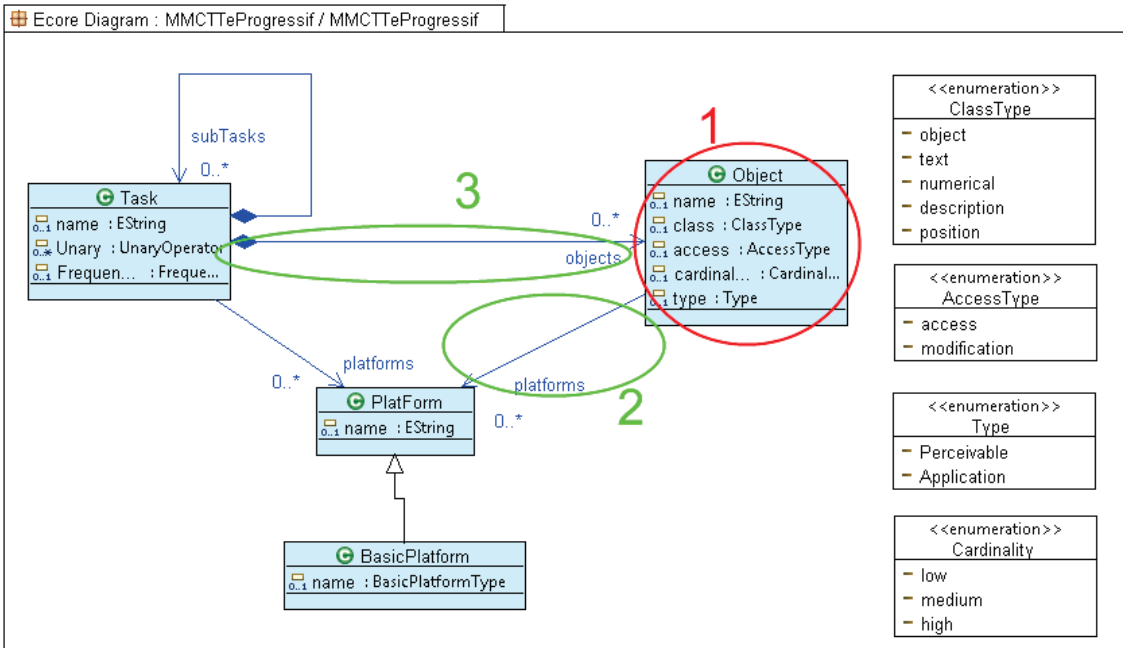

Figure 8. Vue sur le metamodèle de la relation Tâche, Plate-forme et Objet du domaine.

1.2 Généralisation

Bien que la section 1.1 présente une étude de cas sur **CTTe,** nous avons également mené dans cette thèse, la généralisation d'une approche de retro-conception de langage à partir de l'IHM de l'éditeur en donnant un ensemble de règles de bonne pratique. Ici nous proposons une approche méthodologique, parfois semi-automatisable, de rétro-conception à partir des IHM des outils.

On a obtenu à partir de l'éditeur de **CTTe** (un graphe et un ensemble de formulaires d'édition) un premier metamodèle. C'est donc à partir d'un modèle d'interacteur de l'IHM de l'éditeur (élément concret de l'interface ou copie d'écran) que l'on va déduire le metamodèle du métier traité par l'éditeur. On donne ici une solution sous forme de patron de rétro-conception de metamodèle : plusieurs variantes sont cependant possibles.

Motivation : Extraire le metamodèle du type d'interacteur «un seul choix dans une liste» ; exemple de ce type d'interacteur : liste déroulante, cases à cocher exclusives, radio boutons,… Cette liste est contenue dans une « fenêtre » faisant référence à un élément du graphe.
Cas limite **:** Cet interacteur « liste » agit sur d'autres paramètres ou en combinaison avec d'autres « listes ». (C'est notamment le cas dans l'exemple de la figure 9)
Solution : Création d'une classe d'énumération des valeurs possibles contenues comme éléments de cet interacteur. Les attributs de la classe correspondant à cet élément du graphe ont une cardinalité de 1 maximum.

Structure:

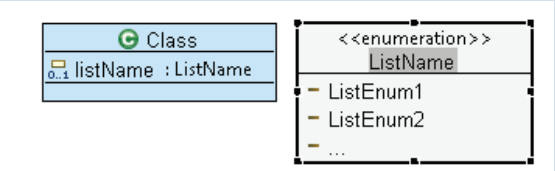

Figure 9. Modèle de la structure de rétro-conception.

Conséquence **:** Un type énuméré est créé comme attribut pour la classe représentant un élément du graphe. Il permet l'instanciation d'une valeur valide pour l'attribut de cette classe.

Exemple: Liste déroulante (figure 9) « Class » qui devient un attribut *class* pour la classe Object (onglet *Objects*) dont les valeurs sont données par l'énumération « ClassType » (figure 10 bis en bas).

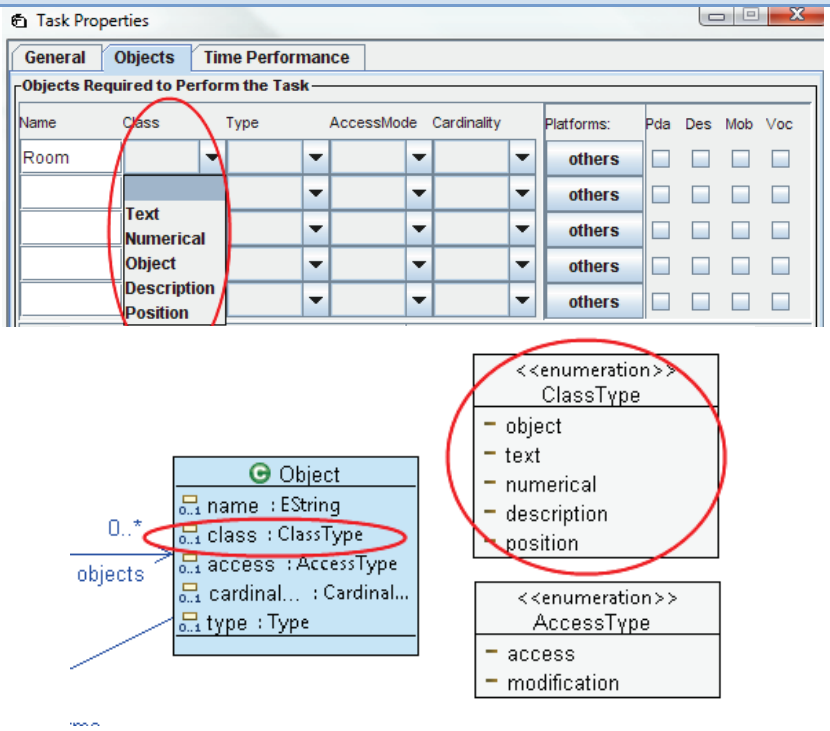

Figure 10 et 10bis Application des recommandations pour rétro-concevoir l'IHM de **CTTe**.

1.3 Discussion

L'extraction de modèle n'est pas une science exacte: nous posons des hypothèses qu'on doit revoir au fur et à mesure de l'utilisation. Il est aussi possible qu'on ne puisse pas déterminer certaines cardinalités dans un metamodèle (en particulier s'il y a des valeurs exclues). La retro-conception de

tels systèmes est un processus lent et itératif. Cependant il donne une appréhension et une réutilisation possible des systèmes propriétaires.

Le projet **Modisco**[Modisco] se focalise sur l'automatisation de la rétro-conception pour des systèmes propriétaires. Dans certains cas, au moins à titre de point de départ, il est possible d'automatiser (ou de semi-automatiser) par des règles de transformation, l'extraction de langage à partir des éditeurs ou à partir des formats d'enregistrement.

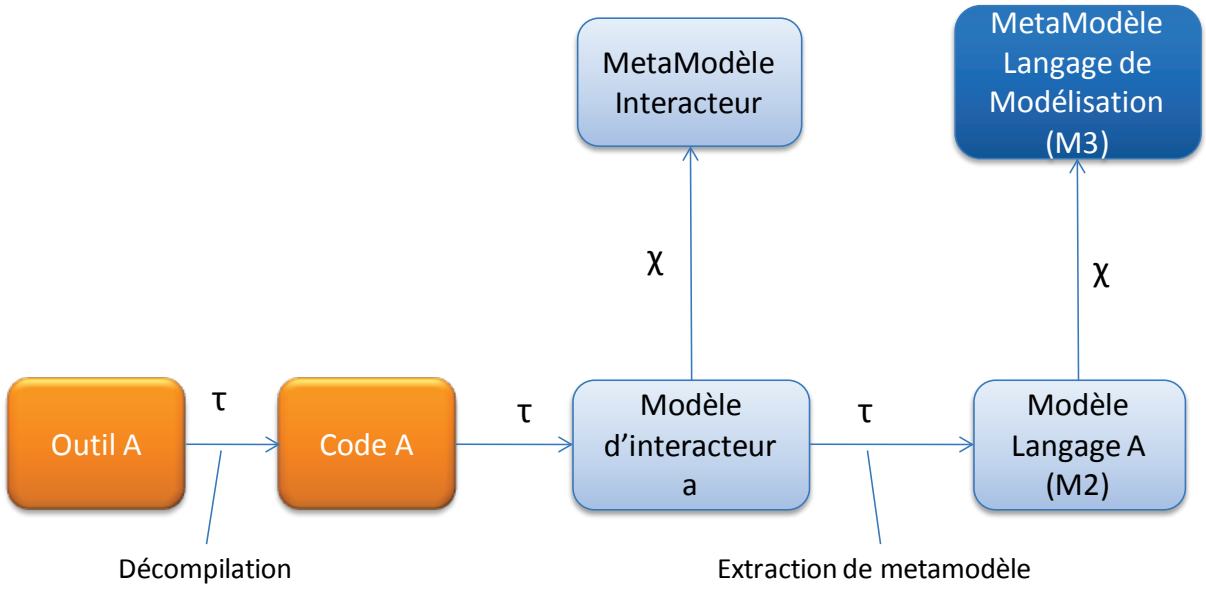

Figure 11. Schéma de principe d'extraction de metamodèle.

La figure 11 représente les différentes étapes de transformations (τ) à réaliser pour automatiser l'extraction de metamodèles. La première transformation est une décompilation permettant d'obtenir le code de l'IHM de l'Outil A. On peut également réaliser une analyse dynamique des éléments d'interaction. Ce code d'interface peut ensuite être injecté dans un modèle conforme à un metamodèle d'interacteur ou *IUC.* Ces transformations peuvent être automatisées par un ensemble d'outils : dans le projet *Modisco* il y a notamment un outil de J2SE (code) vers J2SE (modèle).

Lorsqu'on obtient ce modèle d'interacteur, on peut appliquer des transformations d'extraction : par exemple, celle qui est spécifiée sous forme de patron dans le point 1.2 précédent.

2 Comparaison des metamodèles (de tâches)

Nous proposons une comparaison entre des metamodèles de tâches. Il faut d'abord avoir formalisé les différents langages pour pouvoir les comparer, comme cela est proposé dans la section précédente.

Nous présentons ici, une première ébauche d'un travail de fond qui pourrait être réalisé par et pour la communauté de l'IHM. Nous étudions ici trois metamodèles de tâches correspondant aux outils **CTTe**, **IdealXML, K-MADe** .

2.1 Metamodèles de tâches extraits

Un bref historique des modèles de tâches est proposé dans la thèse de Van Wellie [(Welie, 2001)]. On y apprend que les modèles de tâches ont été introduits très tôt [(Annett, et al., 1967)]. Les systèmes étudiés dans ce travail sont basés sur ces concepts de tâche et de leur évolution.

Dans la suite de cette section, nous comparons les autres systèmes avec **CTTe**. Les parties entourées correspondent aux originalités des modèles proposés par rapport à **CTTe**.

2.1.1 Metamodèles de CTTe

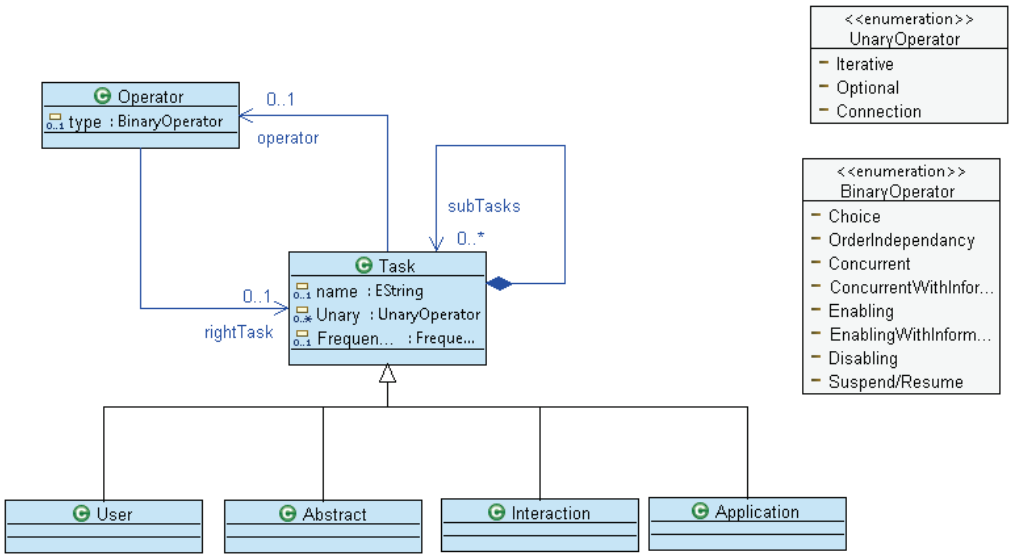

Figure 12. Metamodèle de tâche extrait du logiciel **CTTe**.

Ce metamodèle est celui que nous avons réalisé par extraction dans la partie 2 de ce chapitre. **CTTe** est le point de départ, ou du moins d'inspiration, de beaucoup de systèmes orientés tâche plus récents. Dans la figure 12 nous n'avons pas représenté les différentes sous-catégories de tâches. La précision de cette description qui peut descendre jusqu'au niveau « interaction » semble alors très proche des classes d'interacteurs. Par exemple, la catégorie de tâche « faire un choix parmi N possibilités » fait clairement référence à tous les interacteurs pouvant permettre la réalisation d'un tel choix (radio-boutons, liste déroulante, menus divers etc.). Derrière ces concepts, on sent la volonté des concepteurs de **CTTe** d'aller vers la génération automatique d'IHM. D'ailleurs **CTTe** est couplé avec un autre outil de génération de code : **Teresa**.

2.1.2 Metamodèle de tâche d'IdealXML

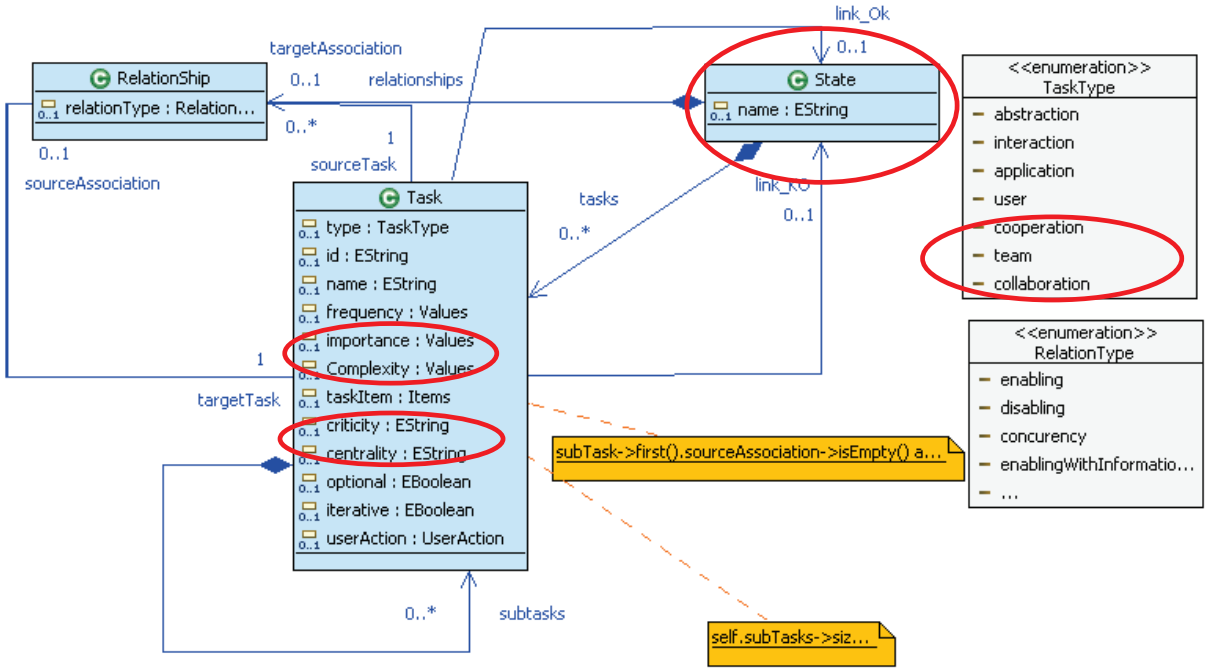

Figure 13. Metamodèle de tâches extrait du logiciel **idealXML.**

IdealXML propose entre autres un éditeur de tâche. L'édition de tâche dans **IdealXML** est inspirée de **CTTe**. **IdealXML** offre les mêmes opérateurs entre tâches, et presque le même type ou catégorie de tâches. Cependant il propose la notion d'état (classe *State*) qui permet de connaître le déroulement (flot) de l'application et conditionne les tâches ainsi que leur enchaînement.

On retrouve dans **IdealXML** les mêmes attributs d'une tâche que pour **CTTe** : les notions de fréquence, d'itération, d'optionalité et les types de tâches (abstrait, utilisateur, etc.). La particularité d'**IdealXML** est de donner des descriptions supplémentaires, comme la criticité d'une tâche (*criticity*), la complexité (*complexity*), l'importance d'une tâche (*importance*).

2.1.3 Metamodèles de tâches par K-MADe

K-MADe est basé sur les concepts introduits par Scapin dans le modèle MAD [Scapin, et al., 1990]. Contemporain de **CTTe**, **K-MADe** présente une approche semblable de description de la tâche (fréquence : attribut *frequency*), des types de tâche similaires (abstraite, utilisateur, interactive, système).

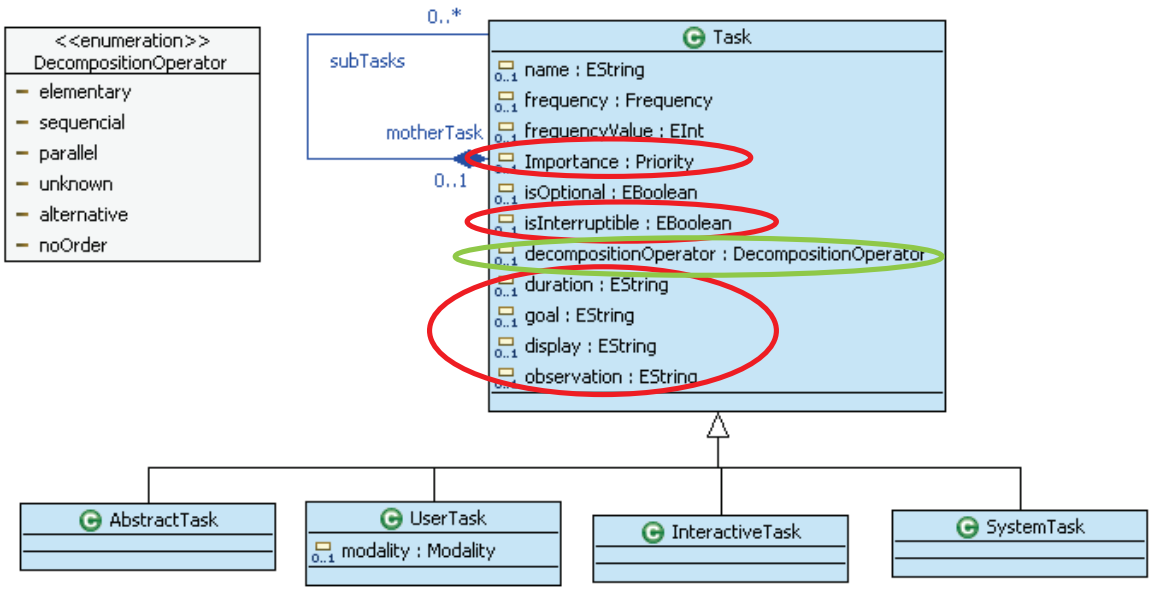

Figure 14. Metamodèle de tâche extrait du logiciel **K-MADe**.

K-MADe montre une décomposition de tâches différente : un seul opérateur de décomposition-attribut de la tâche (attribut *decompositionOperator* entouré). La structure est différente : c'est une véritable structure arborescente où les nœuds sont des tâches. Les tâches non-feuilles ont donc un opérateur de décomposition (ce qui pourrait être donné par une contrainte **OCL**). Ceci a aussi pour conséquence de changer la manière dont on construit l'arbre des tâches avec **K-MADe** par rapport à **CTTe** : il faut parfois ajouter des tâches abstraites pour compenser le fait que, sur un seul niveau de hiérarchie, on ne puisse avoir qu'un seul opérateur de décomposition (voir figure 15). Ceci ne pose pas vraiment de problème : c'est souvent ainsi qu'agit le concepteur, du fait de la méconnaissance des priorités entre opérateurs.

Figure.15 Schéma d'arbre des tâches à gauche, conforme la structure **CTTe ;** à droite conforme à la structure imposée par **K-MADe**. La tâche 6 est abstraite.

K-MADe s'oriente plus vers des aspects contemplatifs que génératifs : beaucoup d'informations concernent la mise en forme, comme la couleur (classe *Label,* attribut *color*) qui fait référence à des éléments destinés au concepteur. De manière générale il y a beaucoup d'éléments utiles au concepteur dans un cadre de documentation (attribut *observation*, donnée de durée *duration*). Les attributs entourés (foncé), sauf l'interruptibilité (attribut *interruptible*), sont des éléments contemplatifs.

K-MADe propose un langage de description de pré et post-conditions pour les tâches, langage que nous n'avons pas décrit ici. Ce dernier permet au logiciel de rendre exécutable une simulation du modèle de tâche construit avec **K-MADe**.

En annexe (annexe 2) nous proposons une comparaison plus détaillée qui reprend la proposition de metamodèle fournie dans ce travail par rapport à **K-MADe**

2.2 Metamodèle des outils : vision globale

Nous étendons la comparaison à la structure complète des metamodèles des outils ; c'est-à-dire aux préoccupations telles que les plates-formes, les concepts du domaine,… Dans cette partie nous mettons en avant les concepts pour chaque outil ainsi que leur localisation sur leurs metamodèles respectifs.

2.2.1 CTTe

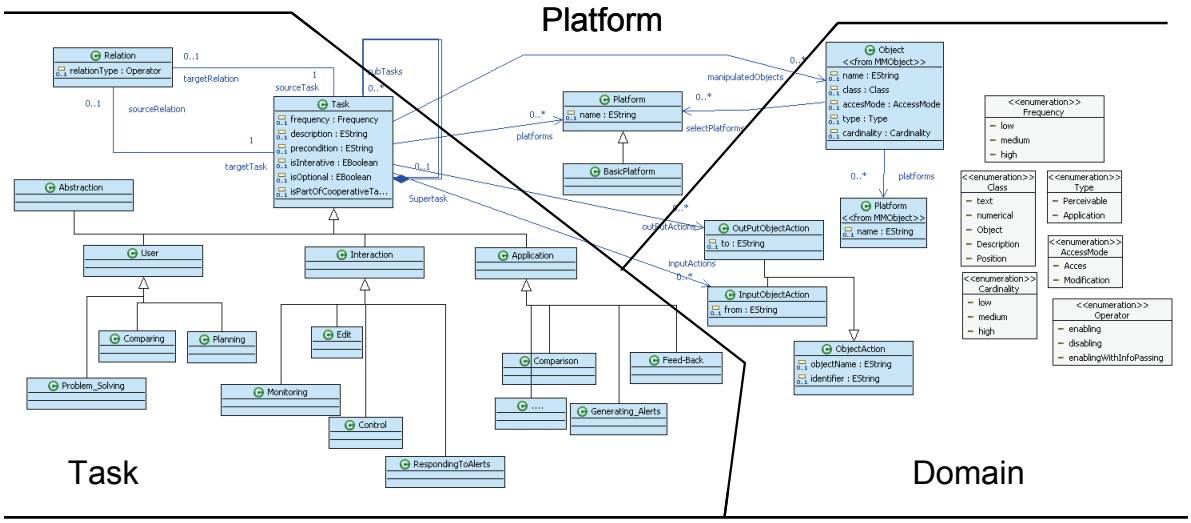

Figure 16. Metamodèle de l'outil **CTTe**.

CTTe est relativement riche en ce qui concerne les spécifications de tâches. Il y a de nombreuses catégories (ou types) et sous-catégories de tâches. En revanche, le modèle de *domaine* est peu

développé : on y retrouve des objets avec des types succincts (les types primitifs plus le type objet). Il est complètement basé sur les besoins en termes d'interaction.

2.2.2 IdealXML

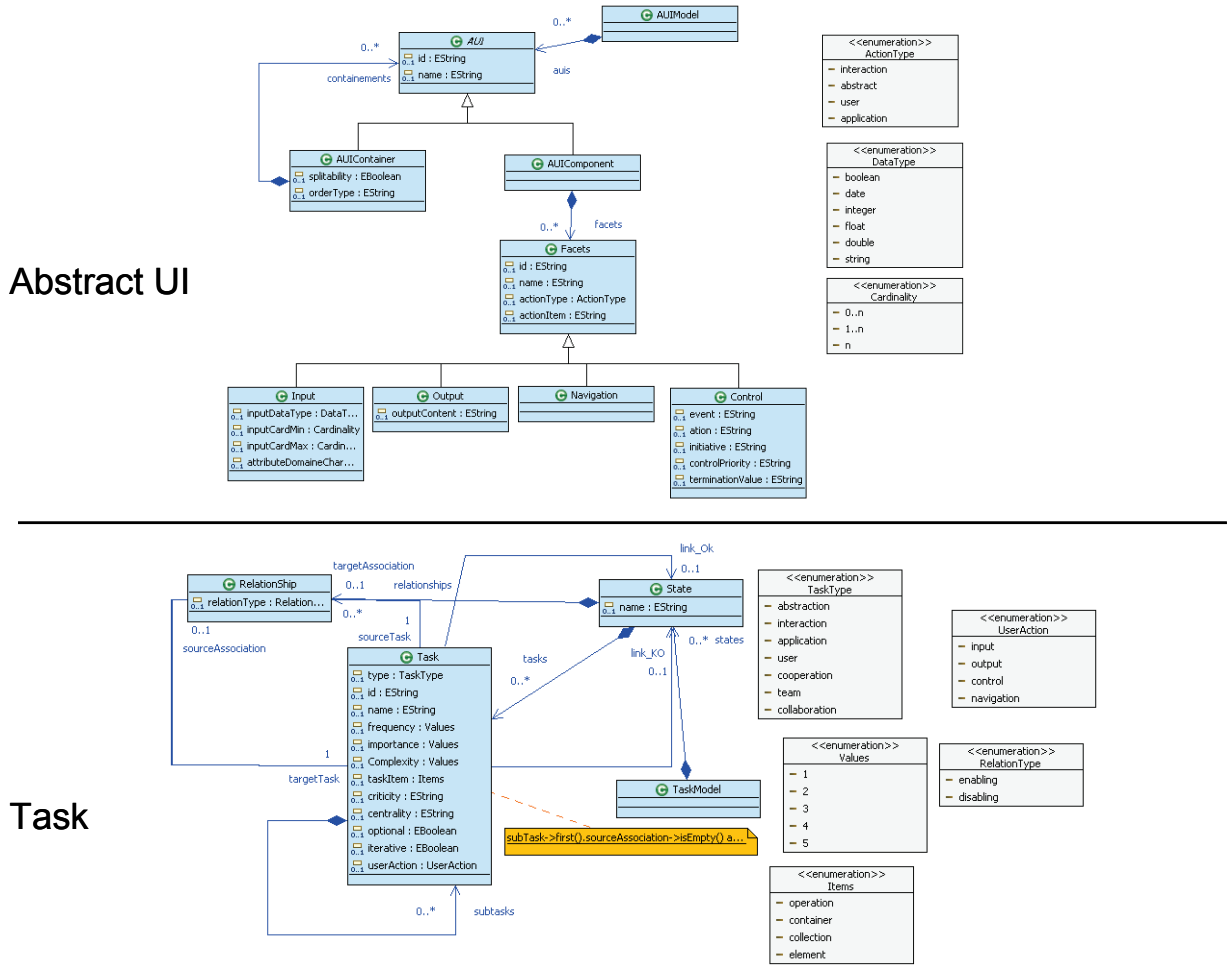

Figure 17. Metamodèle pour **IdealXML**.

IdealXML possède un module de transformation qui génère une IHM abstraite (qui par la suite reste éditable). Le modèle de tâche d'**IdealXML** est donc réalisé dans une optique de génération automatique d'IHM. **IdealXML**, dans une version plus récente que celle étudiée dans cet outil, couvre les autres modèles.

2.2.3 K-MADe

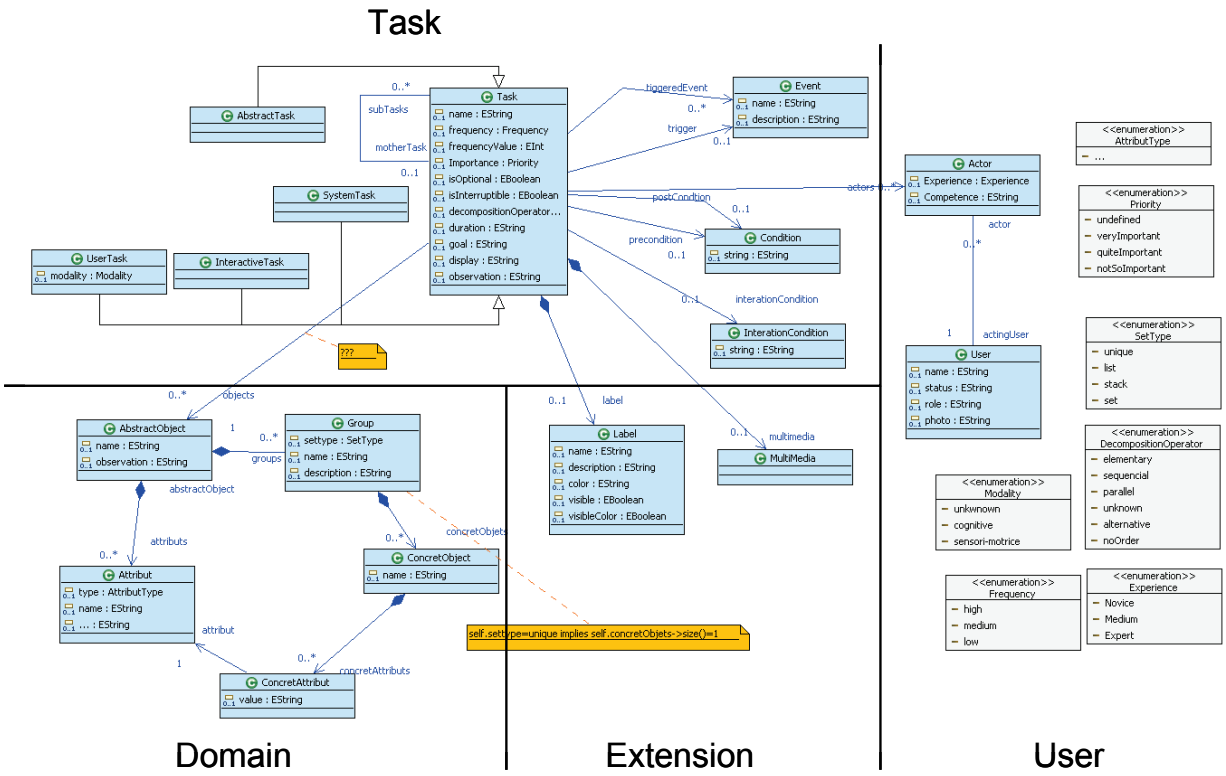

Figure 18. Metamodèle de l'outil **K-MADe**.

K-MADe offre un modèle de l'utilisateur, ce que ne permettent pas les autres systèmes. Historiquement le modèle **MAD** est axé sur l'ergonomie et donc l'utilisateur (partie *User* du metamodèle). Cette partie décrit l'expérience et les compétences de l'utilisateur, ainsi qu'un ensemble d'informations spécifiques (nom, photos, etc.).

En outre, un ensemble d'informations annexes peut être ajouté à une tâche : des libellés ou des éléments multimédias (partie *Extension* du metamodèle) photos, son, vidéo.

De par ces éléments d'extension ou de modélisation de l'utilisateur, **K-MADe** est résolument orienté vers la conception et la simulation pour l'ergonomie des IHM. En effet, les extensions réalisables sont composées d'annotations de couleur, de descriptions ou encore de fichiers vidéo pour aider à l'analyse du modèle de tâche. En revanche les autres parties du metamodèle restent relativement similaires (mise à part la structure) aux autres systèmes étudiés ici.

Enfin le metamodèle des *concepts du domaine* est complet, reposant sur une structure proche des diagrammes de classe et d'objets **UML**.

3 Extraction du savoir-faire : transformations

Jusqu'ici nous nous étions focalisés sur la partie « statique » des éditeurs, c'est-à-dire leur capacité à fournir les mots des langages qu'ils supportent. Dans cette partie, nous faisons cas des transformations qui capitalisent tout le savoir-faire d'un outil de génération.

La découverte des transformations peut s'accompagner d'extraction de metamodèle induite par cette transformation.

Prenons l'exemple **d'IdeaXML** ; ce logiciel propose la génération d'une IHM Abstraite à partir d'un arbre des tâches. Cette transformation est diffuse dans le code de l'éditeur **IdealXML**.

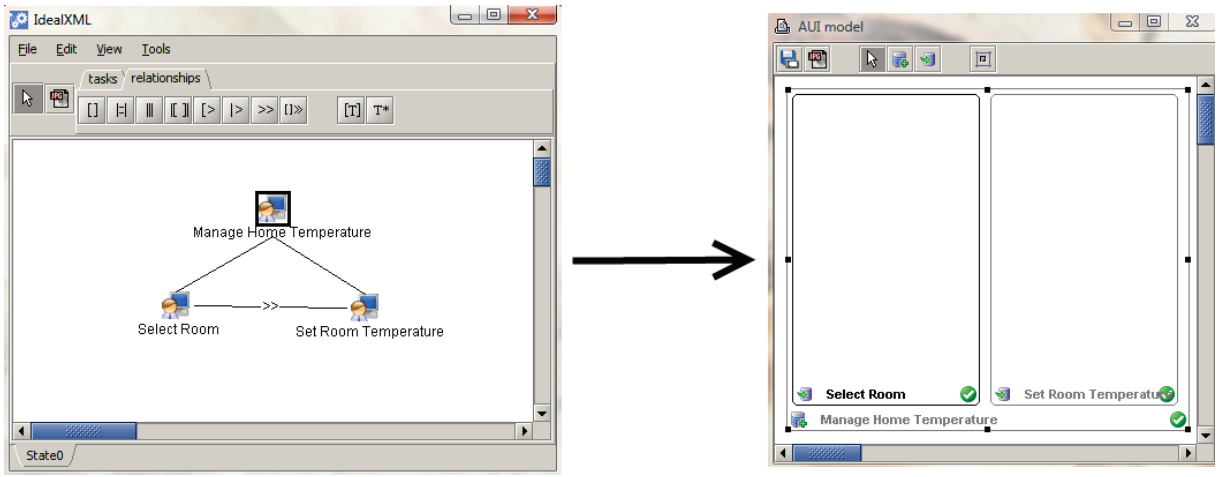

Figure 19. Génération automatique d'IHM abstraite (à droite) à partir d'un arbre des tâches (à gauche) avec **IdealXML.**

La figure 19 montre un exemple d'arbre des tâches (de l'application gestion de température à la maison) en cours d'édition dans **IdealXML**. A partir de ce modèle de tâche dans **IdealXML** on peut obtenir automatiquement le modèle d'IUA (figure 19 à droite).

En appliquant cette génération sur différents exemples, on extrait les règles qui donnent la structure du modèle d'IUA : certaines sont reportées dans le tableau 1. Un état (*State*) devient un modèle d'IUA : *règle TaskStateToAUIModel*. Les tâches feuilles de l'arbre (qui ne sont pas des tâches utilisateurs) deviennent des composants d'IUA : règle *leafTaskToAUIComponent*. Les tâches qui sont des nœuds deviennent des conteneurs d'IUA : règle *TaskToAUIContainer*.

```
module MMIdealXMLTaskToMMIdealXMLAUI ;

create OUT : AUIIdealXML  from   IN  : TaskIdealXML ;

-- Create one AUIModel for each Task decomposition State.

-- AUIModel contains all generated AUIs from the tasks

rule TaskStateToAUIModel {

  from state : TaskIdealXML!State

  to aui_mod : AUIIdealXML!AUIModel (

                auis <- state.tasks )}

-- Create a AUI Component for each leaf task that are not of

-- the following types: 'user'

rule leafTaskToAUIComponent {

  from leaf : TaskIdealXML!Task (

                leaf.subtasks->isEmpty()

                and not(leaf.type.toString() = 'user'))

  to compo  : AUIIdealXML!AUIComponent (

                name <- leaf.name )  }

-- Create a container for each non-leaf tasks which contains all

-- AUIs generated from it's sub-Tasks.

rule TaskToAUIContainer {

  from tsk : TaskIdealXML!Task(

               not(tsk.subtasks->isEmpty()))

  to cont : AUIIdealXML!AUIContainer (

               name <- tsk.name,

               containements <- tsk.subtasks

             ) }
```

Tableau 1. Partie d'une transformation décrite en **ATL** de la génération d'IUA d'**idealXML**. Cette transformation est appliquée sur les metamodèles précédemment extraits.

La retro-conception des transformations est un des points clés des passerelles / échanges possibles qu'on peut avoir entre les différents langages couvrant l'état de l'art.

4 Synthèse

Nous avons vu comment réaliser une extraction de metamodèle. Grâce à cette expérience, nous avons fait la promotion d'une extraction systématique, voire automatique à partir des IHM d'un outil.

La comparaison et la réutilisation sont deux points clés résultant de l'extraction de metamodèle. Nous en donnons ici une ébauche : certains éléments de « comparaison » sont au moins en partie automatisables. Il serait bon de poursuivre les recherches dans cette direction à l'aide d'outils appropriés.

L'extraction de transformations est importante pour capitaliser les savoir-faire de conception. Une fois modélisés (modèle de transformation), ces savoir-faire sont réutilisables : lors de l'évolution du logiciel par exemple. Dans le monde académique, c'est un moyen de re-exploiter les connaissances obtenues par expérimentation. On peut ainsi enrichir une approche de transformation sans être expert des heuristiques de conception qui ont mené à ces transformations. Par exemple, réutiliser des guides de conception ergonomiques pour la génération d'IHM sans être spécialiste des facteurs humains.

Dans le chapitre suivant, nous voyons comment reprendre les savoirs et savoir-faire extraits afin de créer une cartographie du domaine de l'IHM.

Chapitre V : Vers une cartographie des savoirs et savoir-faire pour l'IHM

Nous ouvrons ce chapitre avec, en préambule, des metamodèles d'état de l'art, afin de montrer notre vision et notre pari : les états de l'art dirigés par les modèles.

Nous reprenons ensuite quelques outils du **chapitre II** pour faire une comparaison plus précise entre leurs éléments de modélisation : des *metamodèles*, *transformations*, etc. C'est-à-dire que nous établissons une cartographie des outils et langages utilisés en conception d'IHM. Elle comprend également les relations entre ces artefacts. Cette cartographie est un *megamodèle* pour, et à terme, par la communauté IHM.

Enfin, plus techniquement, ce chapitre présente les passerelles entre langages de modélisation sur des exemples issus de l'IHM. C'est au travers de ces passerelles que nous pourrons donner toute la puissance des *megamodèles* de domaines. Ces passerelles serviront à aller d'une modélisation à une autre, d'une technologie à une autre et d'un domaine à un autre.

1 Préambule

Les états de l'art qu'on retrouve dans la littérature IHM couvrent des ensembles d'outils et proposent des comparaisons à l'aide de tableaux. Ce présent travail n'a pas dérogé à la règle, comme présenté dans le **chapitre II**.

1.1 Metamodèles des états de l'art

Un état de l'art permet d'identifier, de comparer des outils, des langages et, dans notre cas, des *metamodèles*. Les états de l'art sont alors eux-mêmes metamodélisables : ils contiennent par exemple une classe outils (*Tool*) (figure 1), proposent un ensemble d'artefacts (*Artifact*) qui peuvent être classifiés selon des types (langage, code). Par exemple, l'outil **CTTe** permet d'enregistrer un arbre des tâches CTT au format XML. Chaque outil provient d'un ou plusieurs auteurs (*Author*), qui ont réalisé une série de papiers à propos des outils et des artefacts.

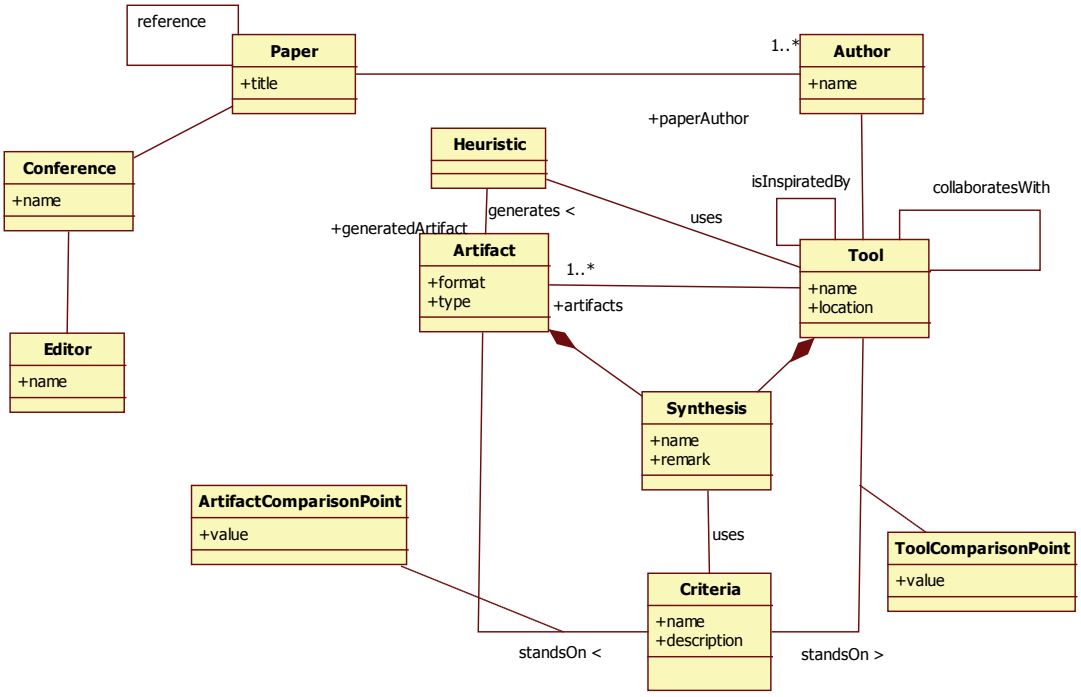

Figure 1. Metamodèle simplifié des éléments composant un état de l'art.

Outre le travail de dépouillement, l'état de l'art comprend la comparaison entre les différents outils et artefacts proposés : parmi ces artefacts, on compte des modèles, des cadres de travail, etc. Une synthèse (Synthesis) est alors donnée pour chaque outil (souvent sous forme tabulaire) et met en avant une explication pour chacun des outils en fonction de critères prédéterminés.

La plupart du temps ces comparaisons entre éléments sont faites par tableau que nous modélisons ici sous forme de classes associatives, entre Artefact (*ArtifactComparison*) et critères (*Criteria*) d'une part, entre Outils (*ToolComparison*) et critères (*Criteria*) d'autre part. C'est notamment le cas dans le **chapitre II** qui proposait un état de l'art en IHM et une comparaison entre les outils sous forme de tableaux.

Prenons par exemple une partie de l'état de l'art de David Thevenin. Dans le **chapitre V**, il y a une énumération des langages de modélisation de concept. Ces langages (*artefacts*) sont réunis dans un tableau puis comparés selon quatre critères avec trois valeurs possibles (cercle plein, demi-plein, vide).

	Fonction	Relation de composition	Relation d'héritage	Domaine de valeur
IDL	●	●	○	○
Quesnot	●	◐	◐	●
UML	●	●	●	●a
XML Schema	○	●	●	●
XTL	●	○	○	●
Adaptive form	○	●	○	●
Représentation de connaissances	○	●	●	●
Sage et autres	○	◐	○	●

Tableau 1. Tableau de comparaison extrait de la thèse de Thèvenin [Thevenin, 2001].

Cette table est modélisable, conformément au metamodèle de la figure1 : les langages proposés sont des *artefacts*, les différents critères des instances de la classe *Criteria,* les valeurs (cercles pleins à vides) des point de comparaison (classe *ArtifactComparisonPoint*, attribut *value*).

Le modèle de la figure 2 décrit, sous forme de diagramme d'objets, un modèle du tableau ci-dessus. Pour des raisons de lisibilité, nous n'avons pris que les trois premiers langages que nous avons croisé avec la colonne « relation d'héritage ». Seul **UML** possède une deuxième relation avec le critère fonction. Sur chacun des points de comparaison on a fait reposer une des trois valeurs : plein, vide, demi-vide (*full*, *empty*, *semi*).

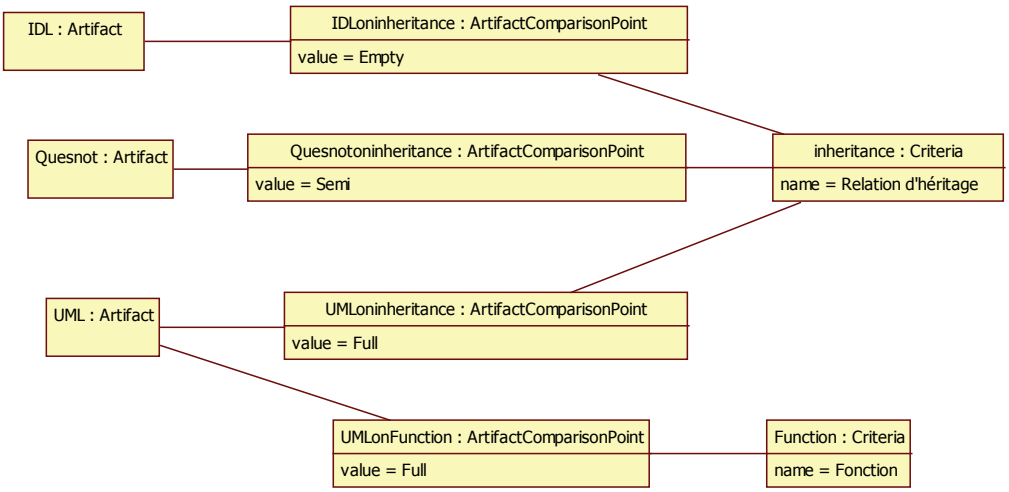

Figure 2. Modèle (conforme au metamodèle de la figure 1) décrivant le tableau proposé dans la thèse de Thevenin (tableau 1).

1.2 Etats de l'art spécialisés

Le metamodèle proposé en figure 1 est une première ébauche de ce qu'on trouve généralement dans les états de l'art. Cependant des états de l'art plus détaillés font référence à des concepts qui décomposent les artefacts ou les outils. Sans parler de metamodèles pour le moment, les états de l'art les plus avancés proposent une énumération de concepts pour un domaine donné.

Pour l'IHM, la proposition d'état de l'art de da Silva [Silva, 2000] est un exemple détaillé. Nous avons extrait le metamodèle de cet état de l'art pour mettre en avant les concepts (figure 3).

Cet état de l'art peut être vu comme une extension de celui décrit par la figure 1. Il présente un ensemble d'outils (*Tools*) possédant un ensemble d'informations sur les auteurs, les universités et leurs localisations. Ces outils sont en relation avec des artefacts (*Artifacts*), qui sont ici représentés sous l'appellation *ModelType*. La classe *ModelType* est spécialisée par héritage ; on retrouve ici la taxonomie des modèles en IHM : modèle du domaine de l'application (*Application Model*), modèle de tâche (*Task Model*), modèle d'interfaces abstraites et concrètes (*Abstract/Concrete Presentation Model*).

A la différence de l'état de l'art général proposé en figure 1, on a représenté ici dans le metamodèle de la figure 2 les éléments qui composent chacun des types de modèles. Ceci permet une vision plus précise du domaine et des points de comparaisons. Par exemple est-ce-que cet outil d'édition de tâche permet de réaliser des *pré-conditions* ?

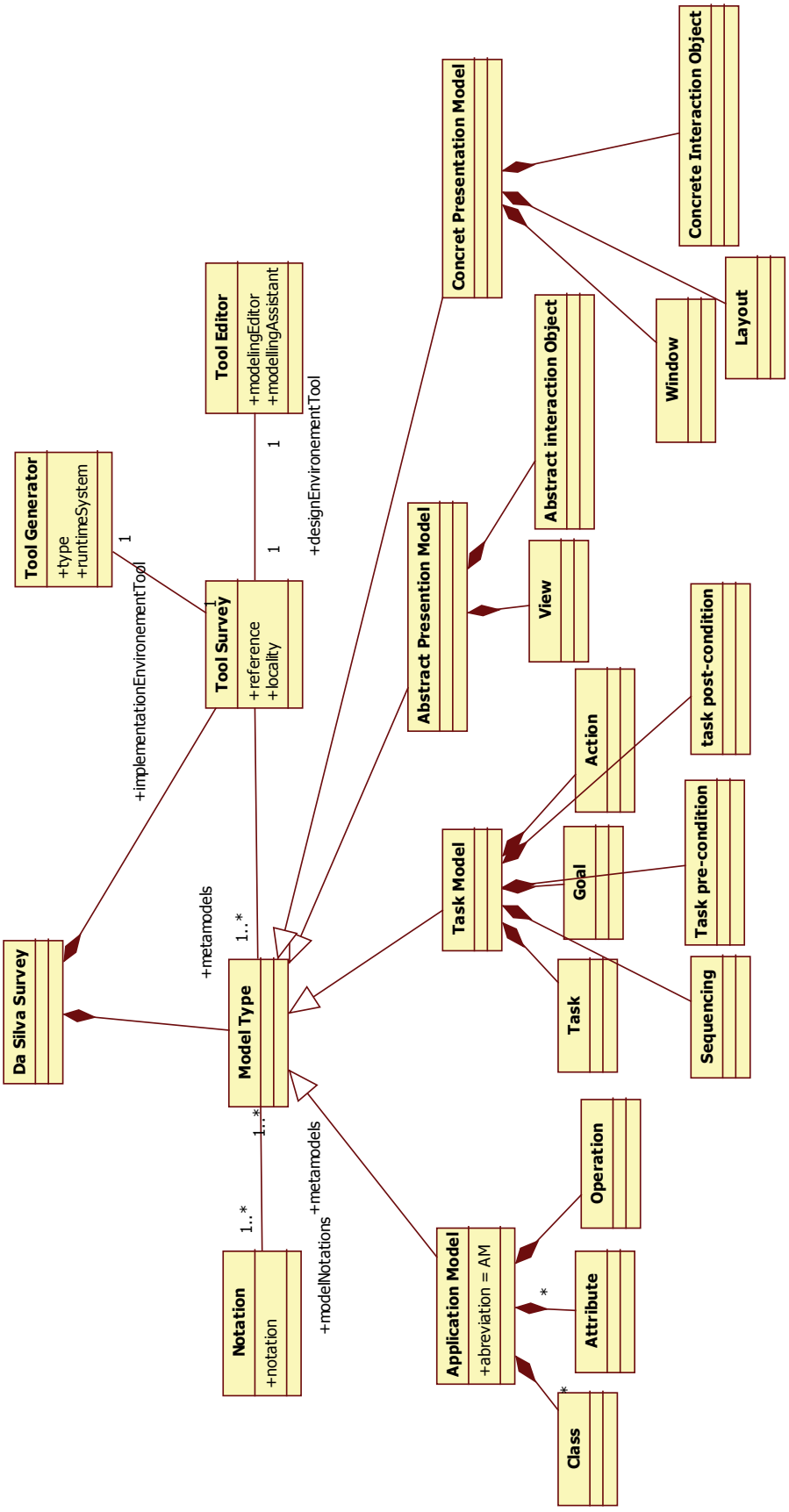

Figure 3. Metamodèle de l'état de l'art proposé par Silva et al.

Avec l'état de l'art de da Silva on s'oriente de plus en plus vers une vision dirigée par les modèles : on exprime sans le dire les éléments qui composent chacun des metamodèles du domaine de l'IHM.

1.3 Démarche : état de l'art orienté modèles

La réalisation d'un état de l'art orienté modèles passe comme tout état de l'art par l'exploration systématique des outils, langages, notations,… proposés tant dans le domaine académique que celui de l'industrie. Puis, pour chacun des outils, on identifie alors un ou des metamodèles correspondant aux différentes facettes que cet outil vehicule. La couverture de l'outil correspond bien souvent au métier pour lequel il est utilisé. Dans le cadre de la conception IHM on retrouve les différents modèles des cadres de travail IHM.

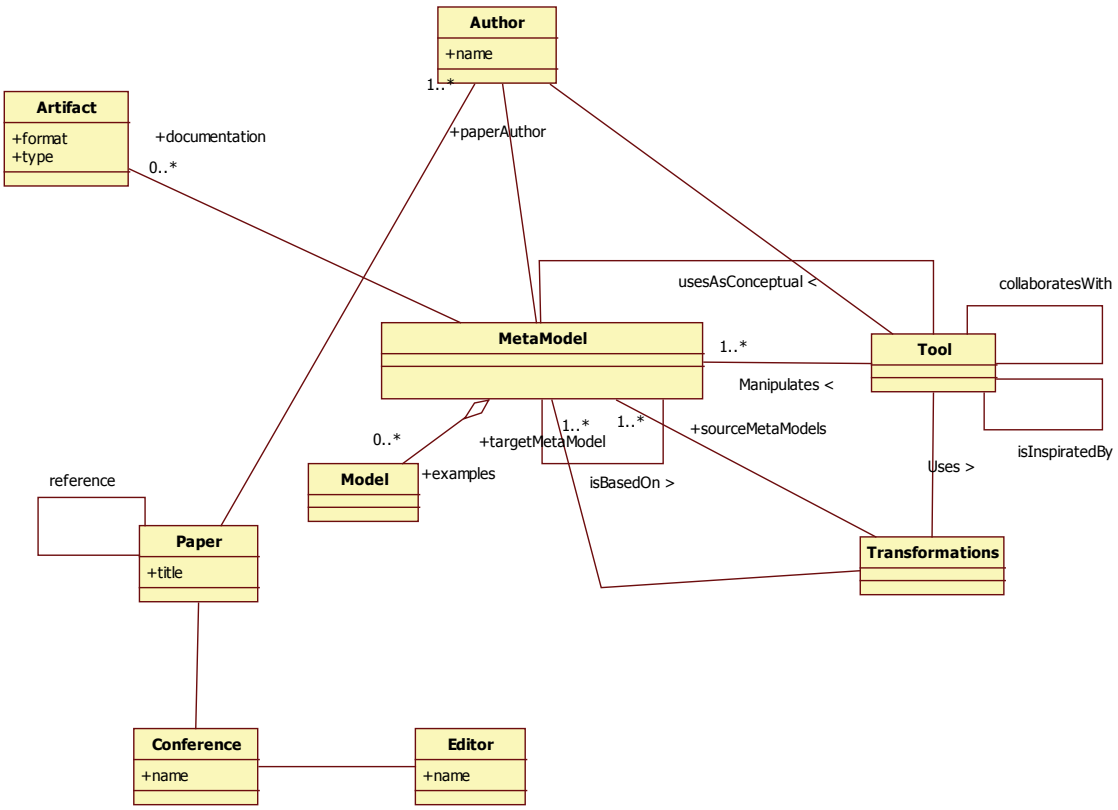

Figure 4. Metamodèle d'un état de l'art orienté modèle à gros grain. Sont représentés les notions de metamodèle, modèles et transformations.

Les critères de comparaison et de synthèse des outils ainsi que des langages qu'ils véhiculent sont ici des éléments des metamodèles : la classe metamodèle faisant référence à tout élément qui peut composer un metamodèle. On retrouve ce concept dans l'état de l'art de da Silva

Le modèle d'état de l'art proposé par da Silva peut lui aussi être revu dans une vision dirigée par les modèles. La classe *modelType* est un metamodèle (classe *MetaModel* de la figure 4). En effet, tout

comme l'est le metamodèle **d'UML**, le metamodèle de domaine pour da Silva est composé de *classes*, d'*attributs* et d'*opérations*.

2 Cartographie des outils, langages, transformations en IHM

Dans cette partie nous présentons une vision orientée modèle de l'état de l'art des outils, langages et transformations pour l'IHM. Cette cartographie représente aussi les relations unissant les outils des différentes organisations. Enfin elle fait la synthèse des points abordés dans ce chapitre. Parallèlement un travail de fond doit être réalisé au niveau de la comparaison et des passerelles entre metamodèles.

2.1 Metamodèle de cartographie

Une cartographie modélise un état de l'art. C'est-à-dire qu'elle propose une représentation des éléments qui composent le domaine étudié au moment où elle est établie. Le metamodèle que nous donnons pour cette cartographie est celui de l'état de l'art dirigé par les modèles de la figure 2. Comme nous l'avons vu, on peut décrire avec les concepts de *metamodèles*, *modèles* et *transformations* les artefacts et outils proposés par les acteurs du domaine. Une telle cartographie est un *megamodèle*.

Certains outils et leurs (méta) modèles associés sont « isolés » (îles) alors que d'autres sont inscrits dans des « continents » de metamodèles reposant sur des préoccupations similaires. Les relations entre outils et metamodèles sont importantes pour bien se rendre compte de ce qui est comparable et interopérable (partie 4 de ce chapitre).

De nombreux états de l'art ont déjà été réalisés, ils correspondent à autant de modèles différents de l'état de l'art. Il s'agit de relier les différentes cartographies entre elles par des éléments de *megamodélisation* communs : modèles, metamodèles ou transformations.

La cartographie que nous avons réalisé dans cette thèse est disponible en annexe 3 ou encore prochainement sur internet (http://megaplanet.org/jeansebastiensottet/).

2.2 Aspects Linguistiques

En étudiant les différents langages, on remarque des leur influences les uns sur les autres. Les évolutions des langages de description sont semblables à celles que subissent les langues naturelles. Cette présente cartographie permet aussi de visualiser les influences (inspiration, dérivation, amalgame, etc.) entre langages.

Dans l'exemple traité dans les parties précédentes, on convient que **CTTe** inspire le modèle de tâche **d'IdealXML** : ce dernier reprend alors la structure et les attributs qui composent **CTTe**. Mais une

partie de la communauté autour **d'UsiXML** ajoute de l'information spécifique à ce modèle. Le modèle de tâche **UsiXML** utilisé par **IdealXML** est en partie né pour se démarquer de **CTTe** mais aussi parce que celui-ci est un système fermé. L'utilisation de tel ou tel langage permet alors de différencier les communautés.

Au fur et à mesure du temps et de leur utilisation (de leurs utilisateurs) les langages de modélisations subissent une évolution. C'est notamment le cas du langage supporté par l'outil **DiaTask** [Reichart et Al.] qui est un bon exemple d'évolution de langages dans la communauté IHM. A l'origine **DiaTask** s'appuyait sur les graphes de dialogue imaginés par Schlungung et Ewert [(Schlungbaum, et al., 1996)] : ils donnent un metamodèle sous forme de spécification formelle de graphe. Ce metamodèle de graphe est intégré par la suite à **DiaTask** par Reichart [(Reichart, et al., 2004)]. Nous proposons dans la figure 5 le metamodèle extrait de l'outil proposé par Reichart : il définit un ensemble de vues (*View*) entre lesquelles on peut opérer des transitions. Certaines vues correspondent à des actions concrètes de l'utilisateur (*ViewActions*) ; ces dernières sont contenues dans des vues.

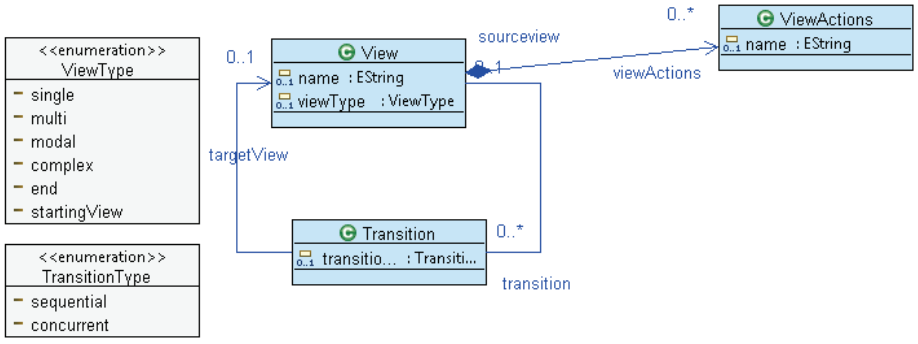

Figure 5. Metamodèle de graphe de dialogue dans **DiaTask.**

Sous l'influence de **CTTe** les modèles de tâches sont intégrés par Wolff. [(Wolff, et al., 2005)a] ; par la suite, des collaborations entre **CTTe** et **DiaTask** ont été envisagées. Le changement n'est pas fondamental en apparence (figure 6) : on remplace les actions par des tâches utilisateurs. Ces dernières font cependant référence aux tâches identifiées à l'aide de l'outil **CTTe** ! En outre les tâches utilisateurs ont un impact sur les transitions : l'exécution d'une tâche peut déclencher une transition. On rentre alors dans une description plus précise en termes d'interaction concrète (passage d'un état conditionné par une action utilisateur).

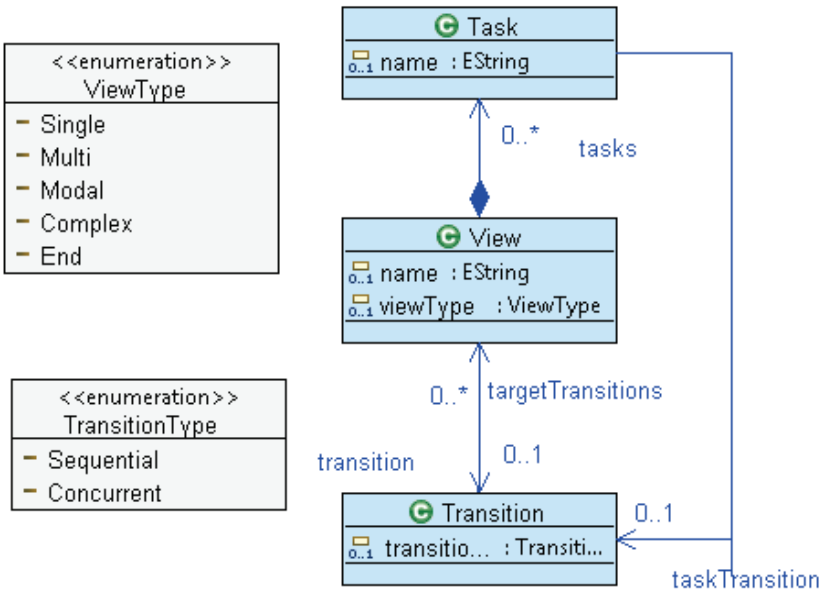

Figure 6. Metamodèle de graphe de dialogue revu avec des tâches *CTT* dans **DiaTask.**

Diatask voit son usage évoluer avec l'apport de génération de « prototype » *d'IU concrète.* Ce prototype est très simple à l'origine : bouton pour les transitions entres les vues. Une fois de plus, sous l'influence émergente de mozzila, le langage d'IHM graphique de **DiaTask** évolue pour intégrer des graphismes plus complexes (dessins SVG). Tout comme son langage vient d'être modifié, l'usage potentiel de **DiaTask** l'est également : la spécification de l'interaction est plus précise et peut cibler d'autres métiers (comme la gestion des réparations d'un garage) [(Wolff, et al., 2005)b].

La figure 7 est le sous-ensemble de la cartographie réalisée pour ce travail traitant de **DiaTask**. Cette cartographie reprend les évolutions et collaborations décrites dans le texte ci-dessus.

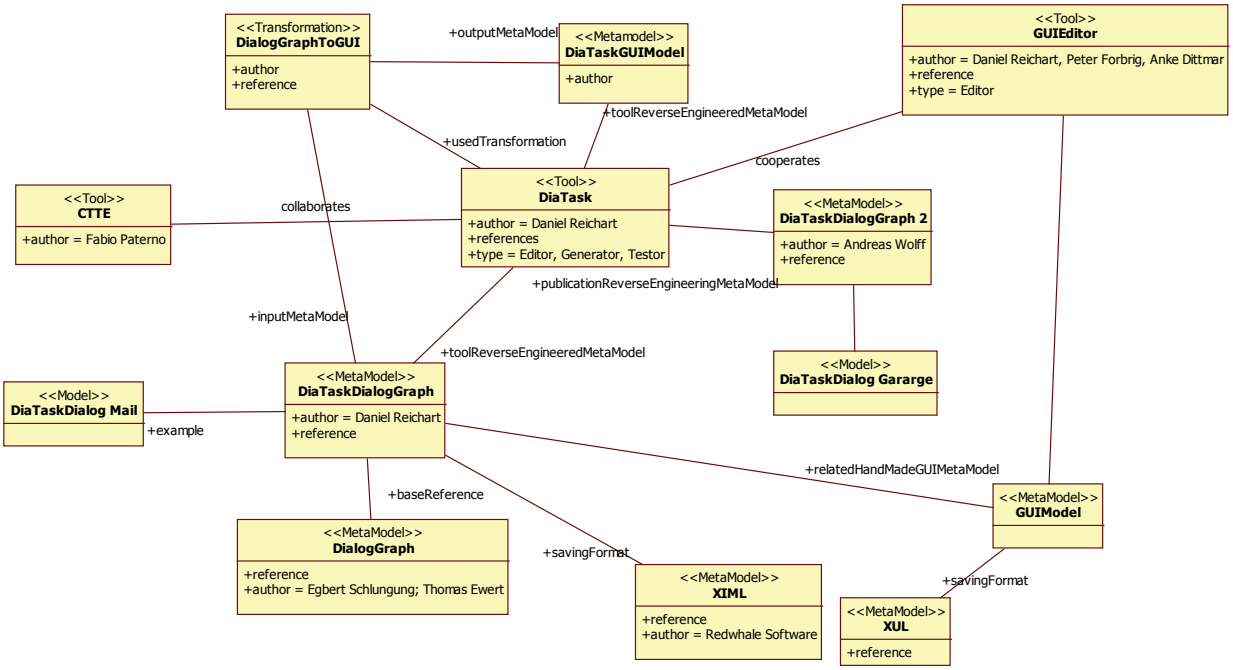

Figure 7. Extrait de la cartographie sur les langages et outils de conception en IHM.

2.3 Visualisation et utilisation

Une cartographie sur un état de l'art telle que proposée dans ce travail permet d'avoir une bonne vision d'ensemble des outils et langages proposés. Cependant, pour observer l'évolution d'un langage ou bien les points précis de comparaison entre langages, il faut passer par d'autres représentations. Ici nous utilisons une frise temporelle (figure 8) pour visualiser les influences subies par l'outil **DiaTask**.

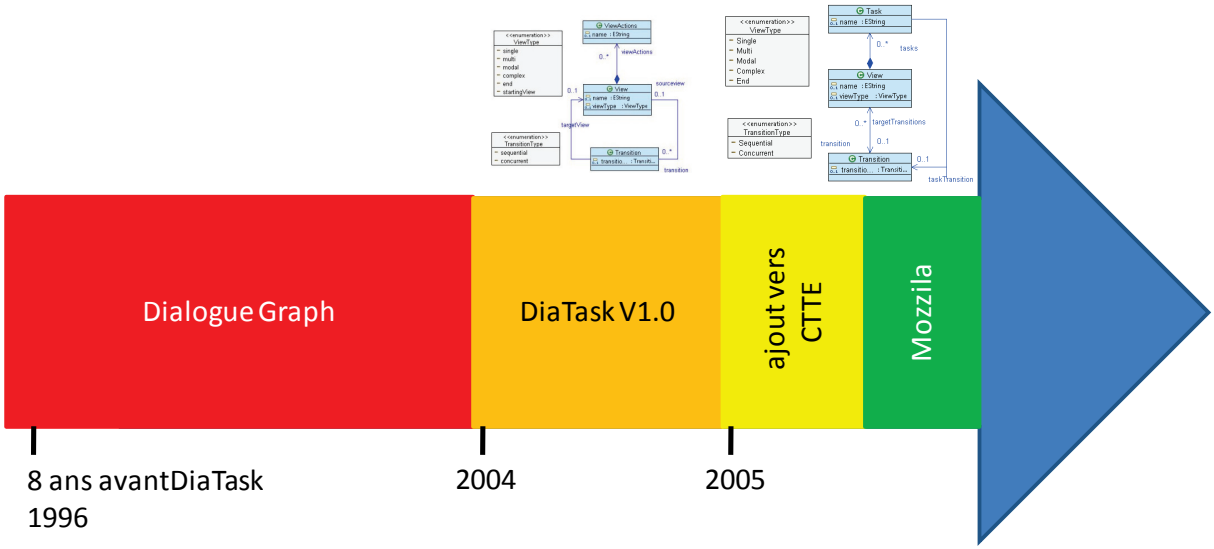

Figure 8. Visualisation sous forme de frise chronologique de l'évolution de **DiaTask**.

Une perspective pour ce *megamodèle* réside dans l'utilisation de techniques de visualisation pour accéder à l'information qu'elle contient. Selon la spécialité de l'utilisateur, il faut alors lui proposer un ensemble de visualisations approprié à son niveau de connaissance et son domaine d'étude. Par exemple un utilisateur du monde de l'*IDM* sera intéressé par les *metamodèles* décrit en **UML**. Alors qu'un ergonome ne peut vouloir que la liste des outils de validation ergonomique.

L'utilisation de ce *megamodèle* dans une perspective de Génie Logiciel est également un problème ouvert : il faut alors fournir les metamodèles et les transformations qui permettent de manipuler les langages définis ici. La problématique de la réutilisation dans un processus de conception des modèles et metamodèles est présentée dans la cartographie.

3 Passerelles entre langages (de tâche)

Un aspect de la malléabilité des IHM, en conception, concerne la flexibilité du système par rapport aux metamodèles et aux outils qui les véhiculent. Dans cette section nous nous intéressons aux moyens et aux motivations qu'on doit mettre en œuvre pour rendre les *megamodèles* (ou cartographies) « productifs » et non plus seulement « contemplatifs ».

3.1 Motivations

Il s'agit de permettre au concepteur d'utiliser le langage avec lequel il est le plus familier ou bien celui qui lui semble correspondre le mieux à ses besoins. Ici nous nous focalisons sur l'interopérabilité entre langages et non pas sur les mécanismes d'extension d'un langage existant. Cette interopérabilité n'est possible que si l'on a réalisé les étapes précédentes :

- Acquérir une bonne connaissance des outils et de leur langage (Extraction de metamodèle : **partie 2 chapitre IV**)
- Identifier les différences, points communs et particularités de chaque outil et de son langage (Comparaison de metamodèles **partie 3 chapitre IV**). Evidemment il faut que les outils présentent des buts communs pour qu'une traduction soit envisageable.

Nous donnons en annexe (annexe 1) un exemple de scénario que nous avons réalisé dans le cadre du réseau d'excellence Similar [(Similar)]. Dans ce projet, plusieurs partenaire travaillent sur des problématiques proches, avec des langages différents et donc des outils différents, dont **CTTe** et **IdeaXML**.

3.2 Vers des passerelles automatiques

Que ce soit dans le cadre d'une collaboration entre partenaires utilisant différents outils ou bien dans l'évolution d'un logiciel au sein d'une entreprise ou d'une institution, les passerelles entre langages sont nécessaires.

Pour les systèmes propriétaires (*legacy*) ou ne présentant pas explicitement de metamodèle, on passe par une phase d'extraction de metamodèle (partie 2 chapitre IV). Cette phase est représentée sur la figure 9 par la flèche extraction de metamodèle partant des outils A et B vers respectivement les metamodèles A et B. Cet extraction peut être suivie de l'élaboration de transformation permettant de passer du langage manipulé par l'outil A au langage de modélisation : par exemple du langage défini par le schéma **d'idealXML** vers le metamodèles en **EMF** que nous avons établi pour cet outil. Il faut alors définir des fonctions pour lire et écrire les langages spécifiques des outils A et B : ils peuvent être réalisés par des *extractors* et *injectors* de **TCS** [Jouault et al., 2006].

Cette phase d'extraction est suivie d'une phase de comparaison (partie 3 de ce chapitre). Les comparaisons nous donnent les points de divergence et de convergence entre les metamodèles. Il est important à ce niveau d'avoir des outils et des langages proches au niveau des préoccupations qu'ils véhiculent. Par exemple il ne s'agit pas d'établir une passerelle entre un éditeur Wysiwyg Swing et un éditeur de tâches.

L'écriture d'une transformation permettant le passage d'un système à l'autre est alors un processus relativement incomplet : on peut perdre de l'information en passant d'une modélisation à une autre, donner un modèle résultant à compléter. C'est une transformation de modèle à modèle qui peut être accompagnée d'un listing des éléments non complets à titre de documentation (figure 9 Transformation du modèle A source vers une documentation type).

Enfin il faut réaliser une transformation du modèle vers le texte ou code spécifique utilisé par le système « cible ». Cette transformation implique aussi l'étude du langage utilisé par l'outil B comme format d'enregistrement.

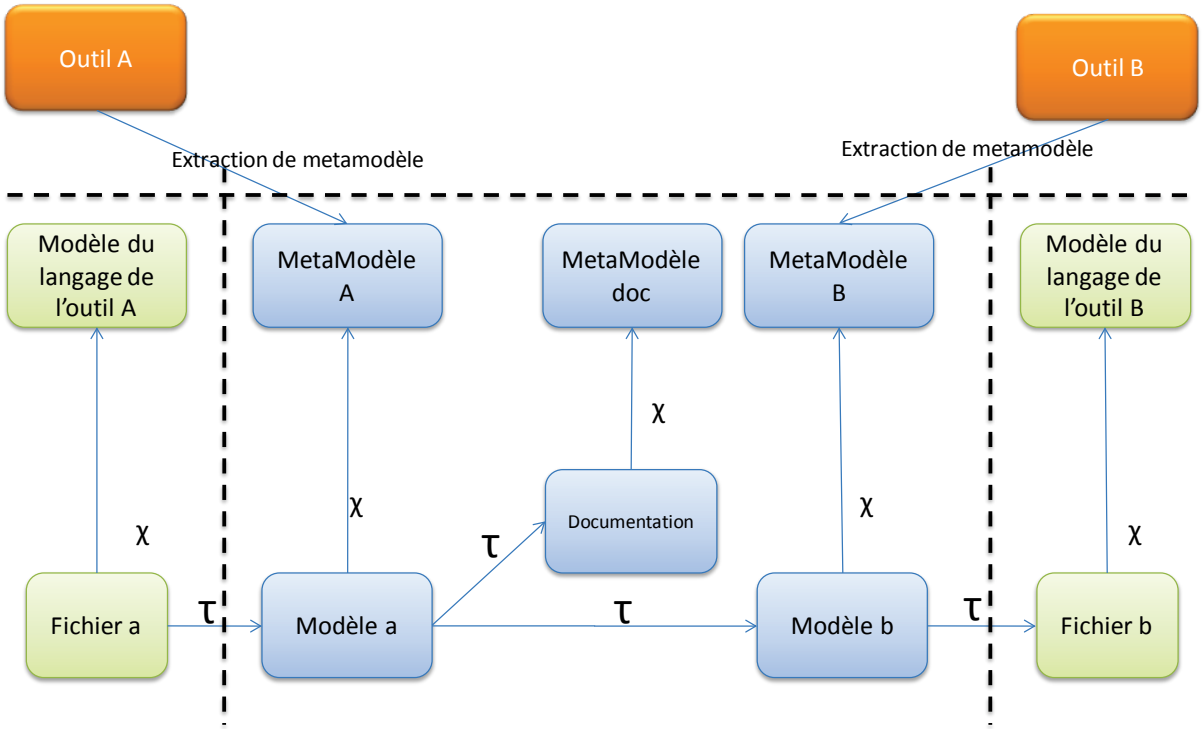

Figure 9. Résumé des opérations (transformation et extraction) à suivre pour l'établissement de passerelles. (Manque la partie comparaison de metamodèles).

4 Synthèse

Ce chapitre présente une initiative de cartographie dirigée par les modèles pour les IHM. Cette cartographie est en fait un *megamodèle* qui s'inscrit, entre autres, dans les initiatives du Zoo de metamodèles (décrit dans le **chapitre III**). C'est en tout cas une autre manière originale et constructive de présenter les états de l'art. En outre, cette modélisation permet de les comparer tant sur les critères choisis que sur le contenu (voir le **chapitre IV** sur la comparaison).

C'est à chaque communauté de réaliser les metamodèles, modèles et transformations de son domaine. La communauté IHM a cependant une tâche importante à accomplir en termes de visualisation d'information. En effet le zoo de metamodèles peut être envisagé selon de multiples perspectives : voir frise figure 8, visualisation en œil de poisson, arbres hyperboliques, etc. pour simplifier l'étude des zoos.

Chapitre VI. Mega-IHM : Malléabilité par l'utilisateur des savoirs et savoir-faire

Au chapitre précédent, nous avons vu l'importance de la *megamodélisation*, un concept émanant directement de l'IDM. Plus particulièrement, nous avons montré comment ce concept permet de cartographier des connaissances en IHM. Dans ce chapitre, nous allons découvrir comment rendre réciproques les contributions IDM et IHM, c'est-à-dire comment le domaine des IHM peut aider l'IDM, et plus généralement comment ces deux domaines peuvent s'intégrer pour donner lieu à un nouveau concept : le concept de mega-IHM, contraction de mégamodèle et d'IHM.

Pour bien comprendre la portée et l'importance de ce concept émergent, il nous faut revenir à des notions élémentaires mais fondamentales. Nous allons voir que les concepts de vues, de langages et d'acteurs, bien que communs à l'IDM et l'IHM, ont été désignés par des terminologies différentes et considérés selon des perspectives distinctes. Bien souvent les critères d'évaluation diffèrent selon le domaine considéré, ce qui est gênant en pratique car cela implique nombres d'incompréhensions entre communautés. Ainsi un travail qui serait jugé comme pertinent dans une communauté, peut être décrié dans une autre, tout simplement parce que les critères d'évaluation ne sont pas les mêmes.

C'est cette incompréhension que nous allons essayer de lever, en montrant que les communautés de l'IDM et de l'IHM pourraient utilement trouver un terrain d'entente autour de la notion de mega-IHM ; notion qui, si on doit la définir informellement mais succinctement, vise à fournir **systématiquement** des IHM pour les modèles et des modèles pour les IHM, ainsi que de gérer les relations entre ces différentes entités.

A priori, l'idée peut sembler triviale, si ce n'est qu'il faut tout de même y intégrer la notion d'acteur et de langage. Il s'agit en effet de déterminer à qui sont destinés ces IHM et ces modèles et, en fonction de cela, proposer des langages appropriés. Les systèmes auxquels on s'intéresse étant de plus en plus complexes, la structure des acteurs et des langages augmente aussi en complexité. C'est grâce à la notion de mega-IHM que nous comptons gérer cette complexité.

Ce chapitre est structuré comme suit. Tout d'abord dans la partie 1 nous allons brièvement décrire les principaux écueils sur lesquels ce concept de mega-IHM pourrait s'échouer. A la frontière entre deux domaines, il risque de ne satisfaire entièrement aucune communauté. C'est un risque que nous prenons dans notre démarche scientifique.

Dans la section 2, nous présentons les concepts fondamentaux qui font que l'IHM et l'IDM traitent de problèmes similaires mais selon différentes perspectives : quelle est la différence entre une IHM et

un modèle ? Les deux représentent une vue partielle sur un système… Certains diront qu'une IHM est destinée à un utilisateur alors qu'un modèle est destiné à un développeur, mais dans le fond ce sont simplement deux types d'acteurs. De la même manière, un langage de programmation peut être vu comme l'IHM du programmeur. Bref, il nous faut clarifier cette situation et préciser ce qui peut faire la différence entre modèles et IHM.

Après avoir défini les concepts fondamentaux permettant de lier IHM et IDM, nous allons développer dans la section 3 la notion de mega-IHM. En fait, nous l'utilisons ici comme terme générique pour regrouper différents concepts plus précis : la notion d'extra-IHM (ou μ-IHM, correspondant à la relation μ), la notion de meta-IHM (ou χ-IHM), la notion de trans-IHM (ou τ-IHM) et finalement la notion de mega-IHM qui donne une vision globale. Ainsi la mega-IHM est la transposition du concept de *megamodèle* dans le monde des IHM.

Après avoir énoncé ces différents concepts, nous voyons dans la section 4 que ces derniers correspondent à une réalité, car il existe déjà des incarnations partielles de ceux-ci, par exemple dans le cadre du développement de jeux vidéo. Ces réalisations, provenant de l'expérience dans des domaines particuliers et découlant de besoins réels découverts « sur le tas », nous permettront de valider l'intérêt du concept, même si, à notre connaissance, celui-ci n'a jamais été énoncé explicitement, ni *a fortiori* exploré de manière systématique.

1 Le contexte : points de vue divergents et communautés séparées

Les communautés de l'IDM et de l'IHM ne se concentrent pas *a priori* sur les mêmes objets informatiques et d'une certaine manière ne partagent pas les mêmes valeurs.

1.1 Au centre de l'IHM : l'homme ; au centre de l'IDM : la machine

L'utilisateur est au cœur de toute la problématique des IHM. Tout gravite autour de lui et c'est donc essentiellement la partie visible et perceptible de l'IHM qui sera jugée et évaluée. Une IHM se doit non seulement d'être fonctionnelle, mais également « esthétique ». Une IHM traditionnelle de type « formulaire », bien qu'utile, risque d'être décriée dans le monde de la recherche en IHM, car elle ne correspond pas aux interactions les plus innovantes et ne fournit pas à l'utilisateur une « expérience » sensorielle nouvelle.

Dans le contexte de l'IDM, le cœur du métier est le modèle, mais aussi et surtout, la capacité qu'il a d'être exécutable, c'est-à-dire interprétable par une machine. Contrairement aux approches de modélisation des années 80 et 90 (tels que Merise ou OMT), approches donnant lieu à des « modèles contemplatifs », dans la communauté IDM l'accent est mis très fortement sur l'aspect « productif ». Autrement dit, les modèles sont créés de sorte à être interprétables par la machine.

C'est l'humain qui est au cœur des préoccupations de la communauté IHM alors que c'est la machine qui est au cœur de l'IDM[3]. Il s'ensuit parfois des incompréhensions, mais l'informatique est l'utilisation des machines par les hommes et nous nous devons de combiner les différents points de vue et approches.

Une IHM est jugée satisfaisante si l'utilisateur peut en interpréter correctement les différents éléments. Un modèle est jugé satisfaisant si la machine peut l'interpréter correctement. A communauté différente, critères d'évaluation différents.

Dans le cadre de cette thèse, nous nous intéressons à l'intersection entre IDM et IHM. Il est très important de bien clarifier quelle contribution nous cherchons à faire. Nous nous plaçons en premier lieu dans le cadre de l'IIHM, et notre préoccupation est d'appliquer des techniques de Génie Logiciel à la production d'IHM. Comme nous allons le voir, la génération d'IHM est souvent décriée dans le monde de l'IHM. Inversement, bon nombre de chercheurs en IDM se concentrent sur les aspects abstraits des modèles au détriment de ce qu'ils qualifient de « sucre syntaxique ».

1.2 IHM : les techniques génératives décriées

Dans cette thèse, nous nous concentrons, non pas sur la partie visible des IHM, mais sur sa partie cachée, c'est-à-dire sur le logiciel permettant de réaliser l'IHM. Ainsi plutôt que de proposer des techniques innovantes d'interactions qui « se voient » et font « impression », nous nous concentrons sur des aspects de génie logiciel visant à diminuer les coûts de production d'IHM « classiques », tout en en augmentant leur malléabilité.

La communauté de recherche travaillant sur les IHM innovantes considère souvent avec méfiance la génération d'IHM dites « classiques » et ne retient de cette expérience qu'un échec.

C'est pourtant celle que nous suivons ici. Il est donc indispensable d'étudier les raisons de ce qui est qualifié comme « échec » et de prendre en compte les arguments des détracteurs de l'approche. L'argument principal est que les IHM générées sont de moindre qualité que celles qui auraient été produites « à la main » : celles-ci manquent d'originalité et de la patte de l'artiste (graphiste). Autrement dit, ergonomes, concepteurs d'IHM, graphistes et autres spécialistes des IHM, voient mal leur expertise être remplacée par un « simple » générateur.

En fait, dans cette discussion, une dimension est manquante : c'est le coût de production des IHM. En Génie Logiciel, l'originalité n'est que rarement une valeur : le suivi de normes et l'aspect

[3] ici le terme machine fait référence à une machine abstraite ou concrète et il peut s'agir d'un interpréteur, d'un transformateur ou de tout autre « outils » IDM ayant accès au modèle.

systématique fait partie de la culture GL. Par contre le coût de développement est un critère important ; généralement implicite d'ailleurs.

Nous allons donc distinguer les « IHM originales » des « IHM systématiques ». De plus, entre ces deux extrêmes, nous allons proposer un intermédiaire, les « IHM malléables ».

- Les **IHM « originales »** (ou « sur mesure »). Une grande partie des chercheurs en IHM valorisent ce type d'IHM, car ils cherchent avant tout l'innovation. C'est sur la partie visible de l'IHM que l'accent est mis, et le coût de production n'est pas nécessairement considéré comme un problème. Créativité et originalité sont perçues comme des valeurs positives chez les chercheurs. Dans un contexte industriel, la situation est quelque peu différente. L'aspect « original » n'est pas nécessairement perçu comme un avantage, sauf bien sûr si le système pour laquelle l'IHM est construite est lui-même original et requiert de sortir des sentiers battus. Considérons par exemple l'IHM devant apparaître dans le cockpit d'un nouvel avion de chasse ou dans un sous-marin. Le coût de production de telles IHM est sans commune mesure avec le coût de l'avion. Evidemment il est tout à fait pertinent d'affecter les meilleurs experts à ce projet. La conception, production et validation vont donc être naturellement des activités humaines intensives dans lesquelles ergonomes, graphistes et autres spécialistes d'IHM pourront exprimer leur talent. De même si l'on considère le tableau de bord d'une voiture de luxe, esthétisme, créativité, originalité sont des valeurs positives. Ainsi, les IHM qu'on qualifie ici d'originales sont créés par l'homme et pour l'homme dans un ensemble de contextes restreints et originaux. Il s'agit plus d'un art que d'une discipline d'ingénierie. Certains diront que c'est de la grande cuisine.

- Les **IHM « systématiques »**. Dans le monde du Génie Logiciel, originalité et créativité sont des valeurs souvent perçues (implicitement) comme négatives. En effet, le Génie Logiciel est le règne des normes, des standards et de tout ce qui peut être systématique. L'originalité et la créativité ont un coût et réduire les coûts et les ressources nécessaires au développement et à l'évolution d'un produit est justement l'un des objectifs du Génie Logiciel. Contrairement aux IHM originales dont la production implique un effort important de multiples intervenants, les IHM systématiques sont définies par l'application de règles de transformation plus ou moins systématiques, ce qui permet une certaine automatisation. Bien évidemment, plus les modèles en entrées sont riches et plus l'ensemble de règles est varié, meilleurs sont les résultats obtenus. Inversement, des modèles pauvres et des règles figées donnent lieu à des IHM certes systématiques, mais sans doute de piètre qualité du point de vue utilisabilité. En tout cas si le domaine d'application n'est pas spécialisé.

- Les **IHM « malléables »** constituent un compromis entre les IHM originales et les IHM systématiques. Elles offrent *via* les points de malléabilité la souplesse qui manque aux IHM systématiques. En fait ce qui est en cause dans ces dernières est leur rigidité, le fait qu'elles soient basées sur des règles trop générales et systématiques plutôt que sur le savoir-faire d'un concepteur d'IHM. L'idée est donc simple : il faut rendre explicites savoirs et savoir-faire et fournir au concepteur (ou à l'utilisateur le cas échéant) une IHM pour mettre son « grain de sel » qui rend l'IHM plus facilement utilisable.

De cette discussion il ressort que, bien que les techniques génératives soient souvent décriées dans la communauté des IHM. D'une part il faut déterminer quels sont les objectifs poursuivis (ce peut être de diminuer le coût de production des IHM) ; d'autre part il faut prendre en compte le fait que la génération est une solution potentiellement viable dans des domaines d'applications spécifiques ; finalement si les IHM systématiques ne conviennent pas, il semble possible de remplacer l'approche générative « boîte noire », par une approche plus ouverte basée sur l'explicitation des savoirs et savoir-faire.

1.3 IDM : le sucre syntaxique en décrié

En IDM, l'accent est mis du côté de la machine, ou plus précisément des modèles opérationnels, c'est-à-dire pouvant être interprétés ou transformés par une machine. Dans un tel contexte, il n'est pas surprenant de constater que du point de vue langage, l'accent est fortement mis sur la syntaxe abstraite des modèles plutôt que sur la syntaxe concrète.

La *syntaxe abstraite* correspond à la structure minimale qu'il est nécessaire de décrire pour que la machine puisse interpréter les informations contenues dans un modèle. Les metamodèles, concept fort s'il en est en IDM, représentent généralement la syntaxe abstraite du langage qu'ils modélisent. Par exemple, on parle du metamodèle d'UML comme étant la définition de ce langage. Ce faisant, on met l'accent sur le fait que, du point de vue de la machine (ou plus précisément des environnements de modélisation, des moteurs de transformations, etc.), ce qui est important n'est pas la représentation graphique des modèles, mais au contraire les abstractions qu'ils représentent. En fait, l'un des grands pas en informatique a été franchi dans la communauté des langages formels lorsqu'on s'est aperçu qu'on pouvait raisonner sur la syntaxe abstraite, sans prendre en compte ce qui est appelé le « sucre syntaxique ». Par exemple, les travaux concernant la sémantique des langages sont tous basés sur la syntaxe abstraite, et non sur la syntaxe concrète. Comme bien souvent en tel cas, le manque de discernement après de telles découvertes fait que les aspects qui étaient considérés comme importants se trouvent alors relégués à un rang inférieur. Le terme « sucre syntaxique » décrit bien cette vision : la syntaxe concrète est une affaire annexe dont on

devrait s'abstraire. Le sucre syntaxique n'est là que pour rendre les choses plus « agréables » : sous entendant par là que ce n'est pas une affaire scientifique.

Cette vision, bien que largement répandue dans la communauté des langages formels, mais également dans le monde de l'IDM, correspond à une perspective uniquement centrée machine qu'il est important de remettre en cause. En effet, les modèles constituent le lien entre les ingénieurs et les outils qui les manipulent. On pourra réaliser tous les outils qu'on voudra. Si ce qui est perçu par les ingénieurs n'est pas en accord avec ce qu'ils « veulent dire » ou « comprennent » du système, tout l'édifice de l'IDM s'écroulera.

Dans le cadre de cette thèse, nous définissons le concept de vue comme étant la partie perceptible d'un modèle. Les ingénieurs ne considèrent en pratique que ces vues, et n'ont pas accès aux modèles abstraits que manipulent les machines. C'est donc un point extrêmement important, et si les modèles doivent comporter les bons grains de sel pour les machines, il ne faudrait surtout pas déconsidérer le sucre syntaxique. En fait, cet aspect est la raison d'être des IHM. Il s'agit si l'on y pense de rendre perceptibles les systèmes ; le système de perception de l'humain est alors au cœur du problème. Syntaxe abstraite et syntaxe concrète sont tout aussi importantes les unes que les autres, mais pour des raisons différentes. Alors que la syntaxe abstraite est la partie dédiée à la machine, la syntaxe concrète est dédiée aux ingénieurs. Une fois de plus, nous allons devoir unir ces deux facettes au-delà des antagonismes qui existent parfois.

2 Concepts fondamentaux : modèles, vues, acteurs, langages

Traditionnellement les langages sont des entités qu'on manipule sans en connaître explicitement et à chaque instant la grammaire.

2.1 La pyramide des acteurs : relations horizontales χ de conformités

Comme on l'a vu, on peut représenter les différents niveaux de modélisation de l'IDM sur une pyramide. Ceci donne une vision à gros grain des concepts à traiter par la suite. Dans cette partie, nous considérons aussi les différents acteurs liés à des niveaux de modélisation : ces niveaux correspondent à autant de métiers.

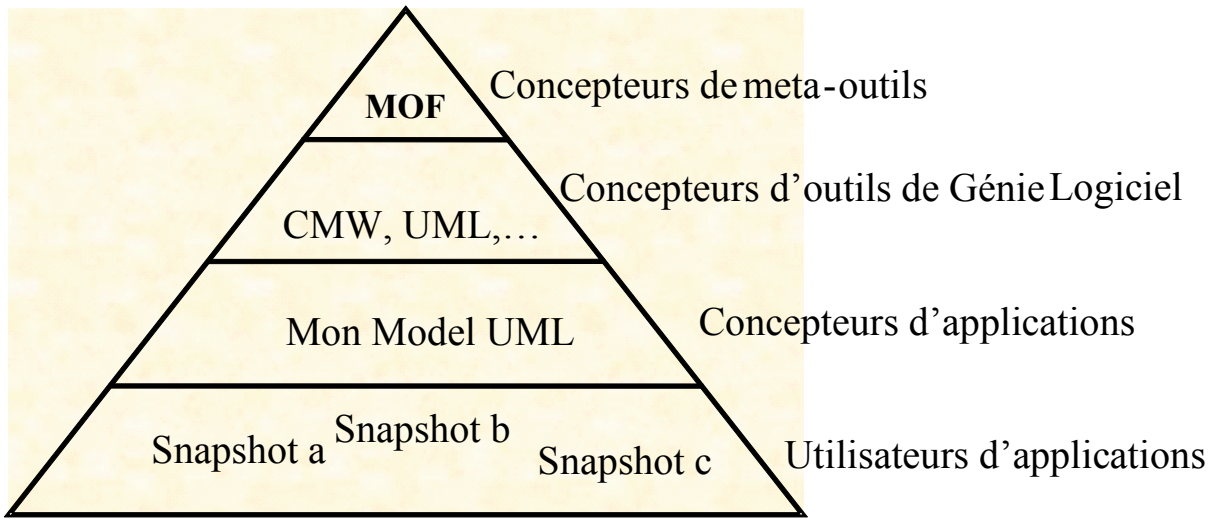

Figure 1. Acteur *versus* niveau de modélisation (exemple avec UML dans la pyramide).

Au niveau du *metametamodèle*, on retrouve communément la **MOF**. Ce niveau concerne un faible nombre de personnes travaillant à l'élaboration de *meta-outils*, c'est-à-dire des outils réalisant des metamodèles, transformations, etc. Par exemple, les ingénieurs qui travaillent sur **EMF** s'appelleraient des concepteurs de *meta-outils*. Ils peuvent travailler avec des langages de plus bas niveau (programmation) pour réaliser les infrastructures subséquentes aux langages de *metameta-modélisation*.

Les concepteurs de meta-outils réalisent des langages de modélisation qui sont utilisés par les **concepteurs d'outils de Génie Logiciel**. Ces développeurs (ensemble encore réduit) produisent des langages de modélisation comme **UML**, des metamodèles de tâches pour les IHM, etc.

Les **concepteurs d'applications** utilisent des langages de modélisation pour réaliser une application (ou le modèle d'une application). La décomposition à ce niveau est plus ou moins complexe selon que l'on considère un seul langage (par exemple **C** pour le code) ou bien tout le cycle de développement (langages d'analyses, de spécification, de conception).

L'utilisateur manipule le langage défini dans l'application par les développeurs. Dans le cadre des IHM, ce langage peut être déterminé par les *widgets* provenant des boîtes à outils.

Si les métiers de l'IDM sont séparés sur la fisgure 1, il reste en revanche à parler de la forme que prennent ces langages pour une séparation sur le « même niveau de modélisation » des métiers : quelles représentations pour créer, manipuler et utiliser des *DSL* ?

Acteur	Langage
Concepteurs de meta-outil	*Metametamodélisation* MOF
Concepteurs d'outil de Génie Logiciel	*Metamodélisation* UML, EMF, MDR, …
Concepteurs d'applications	*Modélisation/programmation* C, C++, Java, etc.
Utilisateurs d'applications	*Interaction* : Bouton, Click, Clavier, etc.

Tableau 1. Associations entre les acteurs et des exemples de langages.

2.2 Vues *versus* modèles : notions de représentation et concrétisation

Traditionnellement en **UML**, une vue est un diagramme : c'est un artefact perceptible et manipulable par un être humain. Par exemple dans un diagramme les couleurs peuvent être importantes, la disposition des « boîtes » aussi. Le modèle, en **UML**, n'est pas une entité manipulable directement par l'humain ; la disposition des objets n'a aucune importante.

Il est important de considérer deux points : la vue ou le modèle. La vue rend les choses perceptibles pour l'humain. C'est un artefact concret et manipulable. Le modèle (on parlera de modèle au sens large) est un système abstrait mais cependant utile pour les raisonnements au niveau de la machine. Le modèle a une structure qui est précisée dans un metamodèle.

Le standard d'architecture IEEE 1471 [IEEE,2000] définit la notion de « vue » comme le modèle et la notion de « point de vue » comme le metamodèle. En outre le concept clé de ce standard passe par les préoccupations des différentes parties ou utilisateurs (*stakeholders*) qui nécessitent alors plusieurs vues différentes. Cette vision s'ancre très bien dans l'IDM mais aussi dans la notion de *mega-IHM* du fait de la prise en compte d'un point de vue « acteur ». Le tableau 2 fait le lien entre le standard IEEE 1471, les notions IDM et l'IHM.

IHM	Standard IEEE 1471	IDM
IHM	View	Modèle
Meta-IHM	View point	Metamodèle
Mega-IHM	..	Megamodèle

Tableau 2. Standard IEEE 1471 versus IDM&IHM.

La séparation de préoccupations conduit à des metamodèles différents, donc des langages différents. Il est important de prendre la mesure des métiers des utilisateurs des langages. En IHM, la conception centrée utilisateur se focalise sur la manière dont l'acteur (utilisateur) va avoir à manipuler l'information (sur l'IHM) qui lui est donnée pour réaliser ses buts.

Dans le cadre de l'IDM d'un point de vue Génie Logiciel, on est confronté à ce même problème : donner les bons éléments et les bonnes représentations des langages à l'utilisateur. On doit donner une « vision » selon le rôle de l'utilisateur du système modélisé. Classiquement on donne l'ensemble des plans pour une maison : plan pour l'électricien, le plombier, le carreleur, le maçon etc. Chacun de ces plans diffère par le fond et la forme, pour être le plus approprié au métier. Cependant il faut aussi avoir une vision globale pour ne pas mettre les arrivées d'eaux à coté des câblages électriques.

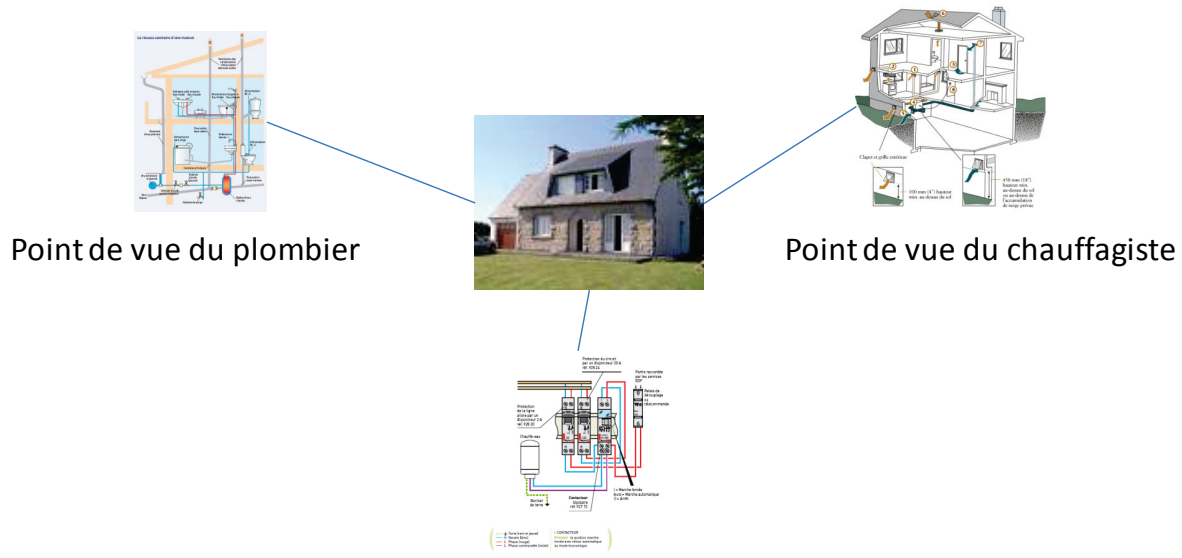

Figure 2. Extraits de quelques modèles (plans) qu'on retrouve autour de la maison

Autour d'un même système on a donc un ensemble de modèles, chacun composant une partie de la modélisation « globale » (relation δ). Pour tous les acteurs on a alors un modèle de vue spécifique. Cependant, pour le néophyte en matière de régulation thermique, typiquement l'habitant de la maison, on peut proposer une autre vue : par exemple la déperdition d'énergie (vue économique).

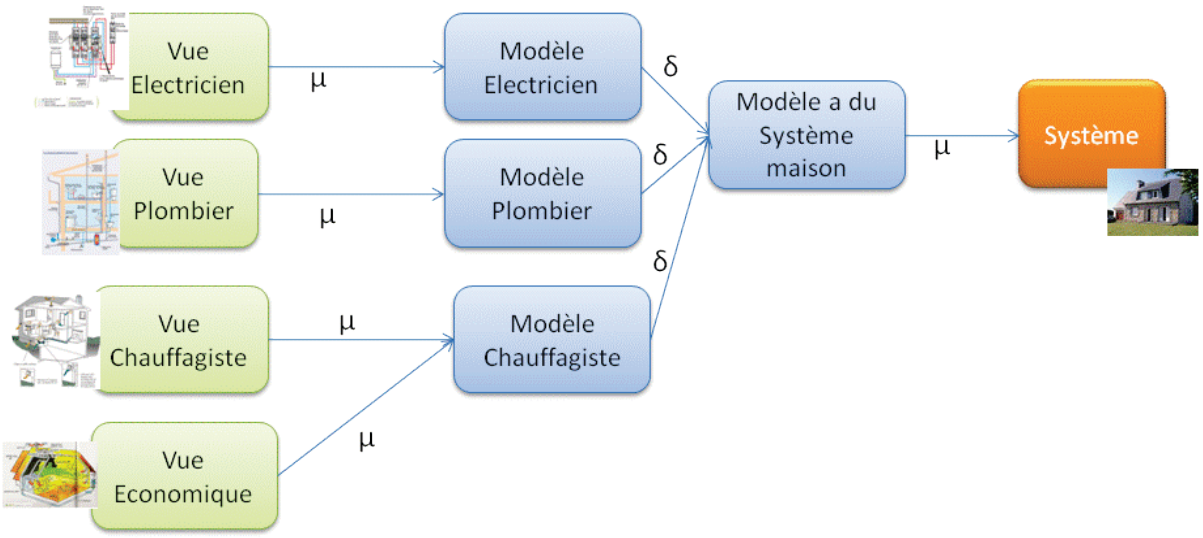

Figure 3. Modèles métiers et vues.

Chacun de ces modèles a son propre metamodèle. Le langage défini par les metamodèles est à associer aux vues. La vue est la forme du metamodèle aussi appelée syntaxe concrète. Le metamodèle de la vue contient, entre autres, l'ensemble des éléments graphiques la composant à l'instar de l'alphabet. La figure 4 est une illustration de ce que pourrait être un metamodèle de vue pour une langue indo-européenne basée sur l'alphabet. Ici la forme graphique (style) est le gothique.

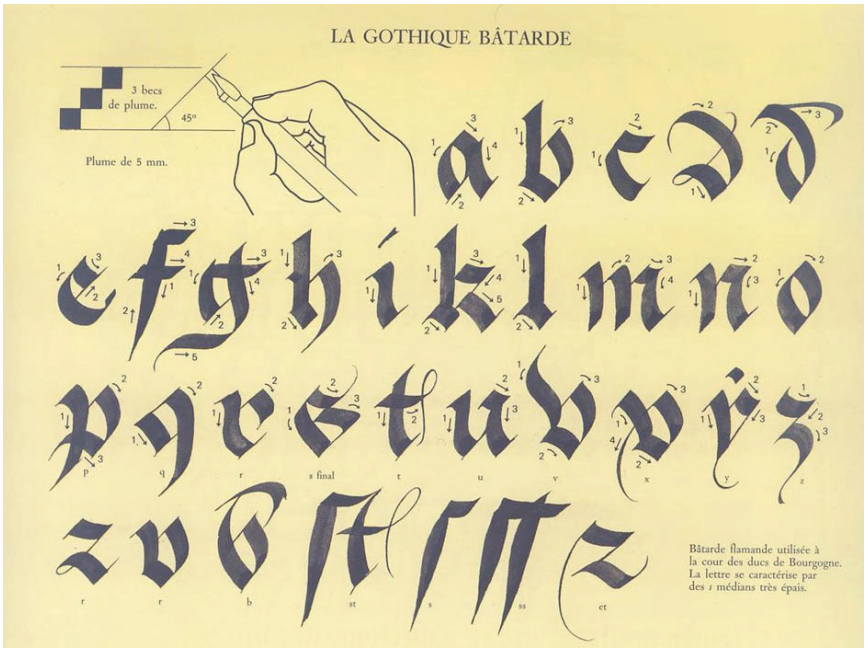

Figure 4. Metamodèle de la vue pour l'écriture. (Extrait de http://aufildugn.org/).

2.3 Application à l'IHM

Dans le domaine des IHM, nous sommes amenés à modéliser les vues pour un utilisateur : une IHM regroupe tout système perceptible (par tous ou une partie de nos cinq sens) et manipulable par un humain. Les metamodèles que nous avons élaborés à partir de ceux présentés dans l'état de l'art (**chapitre II**) décrivent ici le metamodèle de la « forme » ou d'interaction. En d'autres termes ces metamodèles IHM regroupent l'ensemble des informations de l'éditeur du modèle métier.

En IDM, on parle habituellement de syntaxe abstraite pour le « fond » du modèle et de syntaxe concrète pour la « forme ». Dans ce travail, nous allons plus loin en considérant la syntaxe concrète comme une IHM à part entière : avec ses règles de construction, d'interaction, etc. L'IHM est un système concret. La relation entre le modèle métier et la vue notée µi (figure 5) est la relation de concrétisation d'un modèle par un système « IHM » : comment rendre le modèle manipulable par un humain.

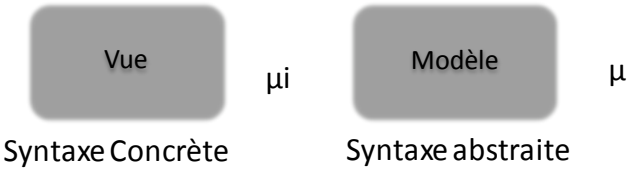

Figure 5. Schéma des différentes relations entre le modèle, la forme du modèle et le système qu'il modélise.

Dans le cadre du développement d'IHM, le système étudié est l'interaction avec l'utilisateur. Pour le métier de l'IHM, la figure 5 est réflexive. Une IHM classique, c'est-à-dire non orientée modèle, propose une relation directe de représentation d'un système abstrait. Par système abstrait, on entend tout système non manipulable par l'utilisateur de manière directe (typiquement une base de données, un composant de sécurité, etc. sans IHM).

On élargit alors la problématique de modélisation et de *megamodélisation* autour du choix à la fois du fond (modèles abstraits) et de leur forme (IHM ou modèles concrets). Cette problématique donne lieu à la notion de *mega-IHM* : une IHM pour contrôler tous les points de vue « modèles » du système. Pour ce faire nous introduisons une taxonomie des IHM (modèles concrets) selon le niveau de modélisation. C'est un problème réunissant IDM et IHM dans une collaboration mutuelle.

3 Extra-IHM, Meta-IHM, Trans-IHM et Mega-IHM : une IHM pour chaque modèle et un modèle pour chaque IHM

Dans cette section nous présentons les concepts et la mise en œuvre de la *mega-IHM*.

3.1Meta-IHM, extra-IHM, IHM, définitions

Chacun des éléments du *megamodèle* (par exemple *metamodèle* de tâche, modèle de tâche,…) est placé sous le contrôle d'un acteur (linguiste, ingénieur, utilisateur).

3.1.1 IHM et utilisateur

Comme définie dans la partie précédente, une IHM rend un système perceptible et manipulable. Dans le cadre d'une IHM classique on a le patron suivant (figure 6). Les IHM orientées modèles en revanche utilisent un modèle intermédiaire comme dans la figure 5.

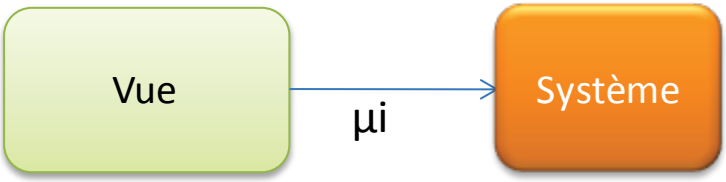

Figure 6. Modèle d'une IHM classique.

Dans le cadre d'une application domotique (exemple de l'introduction), l'*utilisateur* va se servir de l'IHM de gestion qui lui est fournie (figure 7). Cette IHM lui permet de sélectionner une pièce et pour chacune des pièces d'accéder aux différentes options d'interaction qui lui sont proposées : par exemple entrer la température désirée de la pièce considérée. Cette IHM est la représentation du modèle d'interacteur et le rend ainsi perceptible et manipulable.

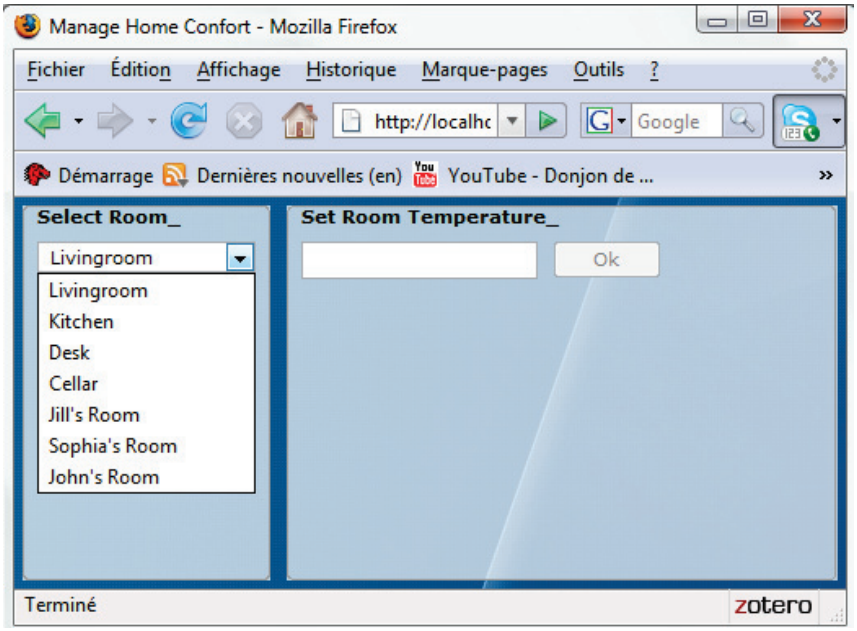

Figure 7. IHM pour l'utilisateur final.

3.1.2 Extra-IHM ou (µ-IHM)

Une extra-IHM ou µ-IHM est une IHM qui représente et donne le contrôle sur ou par l'intermédiaire d'un modèle à un système concret (c'est-à-dire une IHM). C'est en quelque sorte l'IHM de configuration d'une IHM. Le patron type d'une *extra-IHM* est donné dans la figure 9 ; l'*extra-IHM* donne accès au modèle abstrait de l'IHM.

Une *extra-IHM* est également une IHM (orientée modèle). C'est sa position relative en tant qu'IHM d'un modèle d'IHM qui en fait une *extra-IHM*. La chaîne de modèles abstraits pour décrire l'IHM concrète n'est pas forcement limitée à un seul modèle.

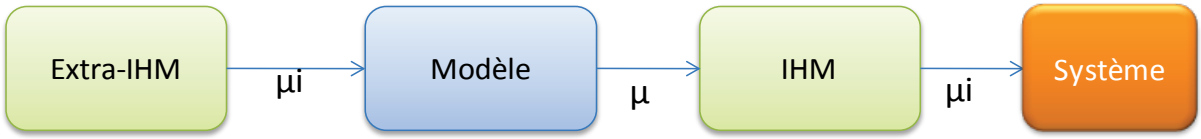

Figure 8. Patron d'extra-IHM.

L'IHM pour l'utilisateur (figure 7) est obtenue à partir d'un ensemble de modèles réalisés par *l'ingénieur concepteur de l'application*. Ces modèles font l'objet d'une réalisation dans un éditeur de modèle adapté (figure 9). L'éditeur de modèle de tâche ici présenté s'appuie sur le metamodèle de tâche : c'est une *extra-IHM*. En outre nous exposons la relation entre une tâche et une machine, ce qui permet au concepteur d'agir sur les tâches qui seront visibles ou non par l'utilisateur.

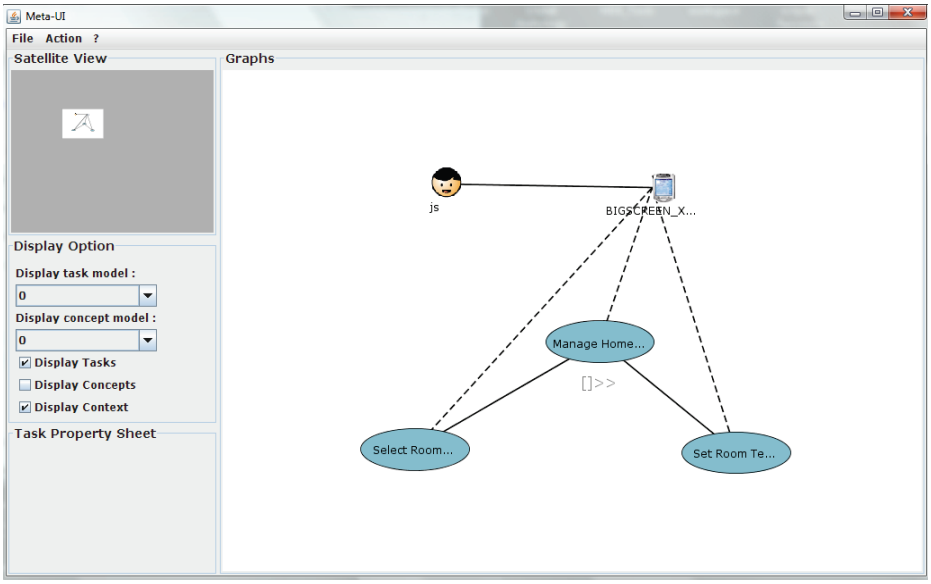

Figure 9. *extra-IHM* : IHM de malléabilité représentant et manipulant le modèle de tâches.

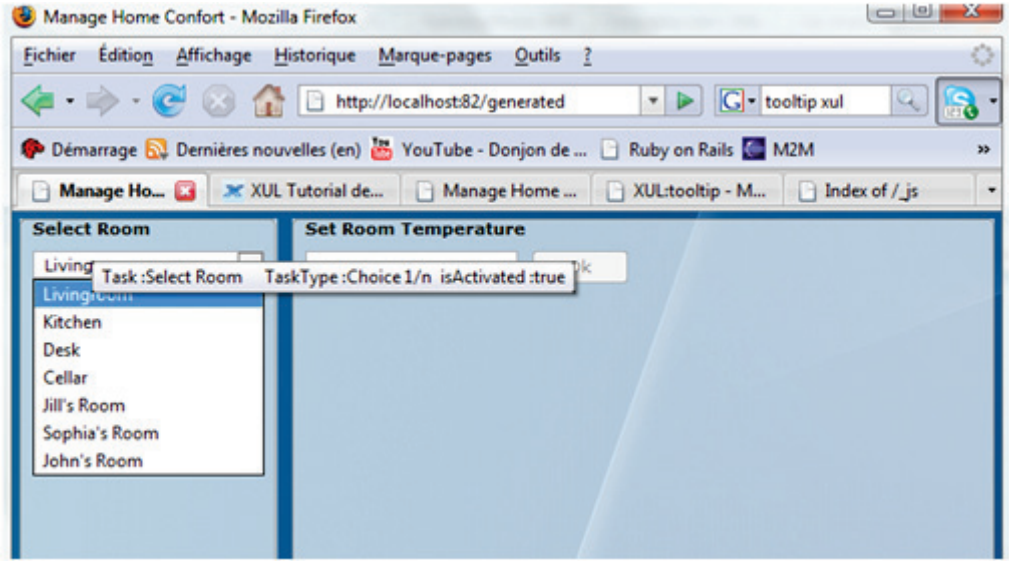

Figure 10. Extra-IHM permettant le débogage en direct de l'IHM.

D'autres extra-IHM peuvent être proposées au concepteur. Dans la figure 10 nous voyons une extra-IHM de débogage : la partie extra-IHM ne concerne que le « *tooltip* » (bandeau affichant de l'information contextuelle par-dessus un élément de l'IHM) indiquant ici le nom de la tâche, le type de la tâche et si cette dernière est activée.

Le *langage* véhiculé par ces extra-IHM est destiné au concepteur qui a une connaissance des arbres de tâches. D'autres versions, adaptées à l'utilisateur final, peuvent être proposées pour qu'il puisse reconfigurer son espace d'interaction. Dans la figure 11, nous présentons une extra-IHM qui permet à un utilisateur de migrer des parties d'IHM de son PC vers son PDA. Cette extra-IHM représente le

lien entre une tâche et une plate-forme. Ceci revient à agir sur la relation entre une tâche et une plate-forme (voir figure 9) mais cette IHM-ci est plus didactique pour un utilisateur.

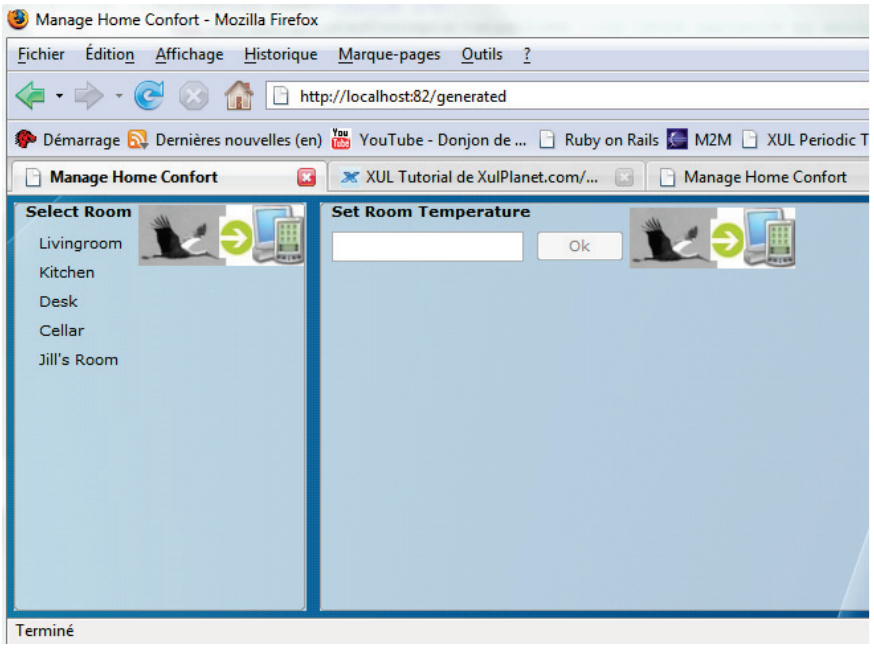

Figure 11. Extra-IHM pour l'utilisateur : contrôle de la « migration » d'un élément d'IHM sur ses appareils.

3.1.3 Meta-IHM (ou χ-IHM)

Une *meta-IHM* ou *χ-IHM* est une IHM donnant accès (en visualisation et/ou en manipulation) au *metamodèle* du *modèle* abstrait représentant l'IHM. Dans la figure 12, il n'y a pas de relation χ entre la *meta-IHM* et *l'extra-IHM*. Mais d'un point de vue pratique, les actions de la *meta-IHM* ont un impact sur *l'extra-IHM* par l'intermédiaire du modèle abstrait. D'un point de vue conceptuel et de la même manière que l'on a un μi, on peut avoir un χi : c'est la conformité en termes de représentation IHM.

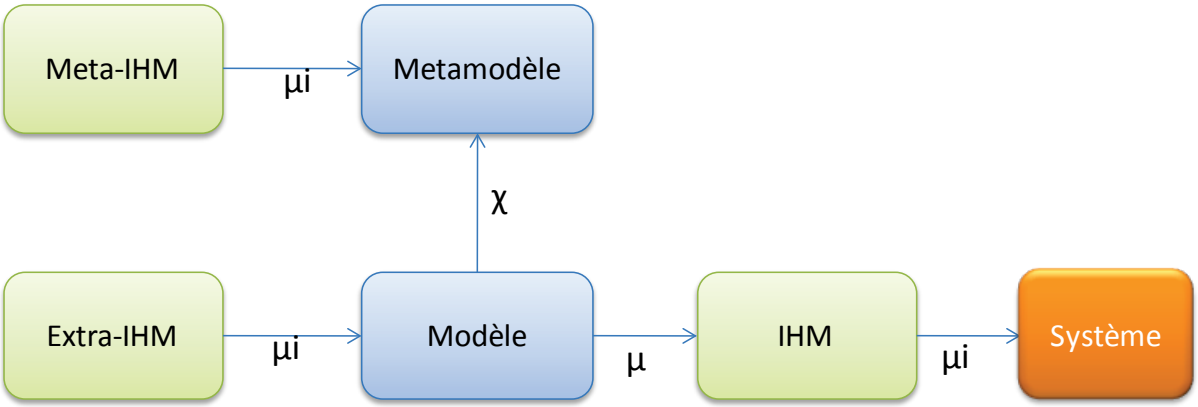

Figure 12. Patron type de *meta-IHM*.

Les légendes de cartes (routières, transports en commun, etc.) sont les meilleurs exemples de *meta-IHM* qu'on a dans la vie quotidienne. Elles véhiculent une information complète et concise des symboles indiqués sur la carte : elles offrent de l'information à l'utilisateur de la carte.

La meta-IHM de la figure 13 offre à un utilisateur débutant ou un étudiant débutant en IHM, une idée des objets qui composent son IHM et leur objectif. On fait ici le parallèle avec la légende grâce aux « tooltips ». Ces derniers, apparaissant sur les objets de l'IHM, donnent le type et la pertinence d'utilisation. Ici le « tooltip » indique à partir du bouton « Ok » la nature de cet interacteur (*button*) ainsi que l'utilisation qu'on peut en faire : lancer une action, confirmer l'action réalisée par un autre interacteur,...

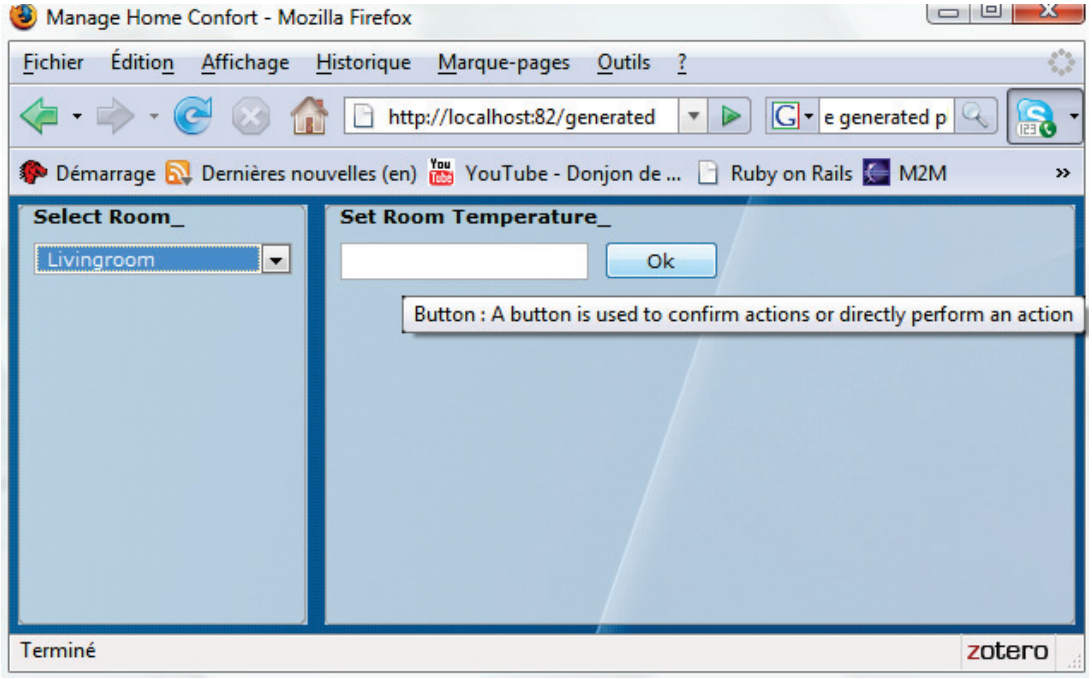

Figure 13. Meta-IHM d'apprentissage en lecture seule.

Le *metamodèle* lui-même a été réalisé par l'intermédiaire d'un éditeur **UML** sous forme de diagramme de classes. Ce *metamodèle* de tâche est réalisé par un expert du *langage* maîtrisant lui-même le *langage* de modélisation **UML**. La figure 14 présente l'IHM de conception d'un *langage* de tâche dans **Topcased** [Topcased] : cette IHM est appelée une *meta-IHM*.

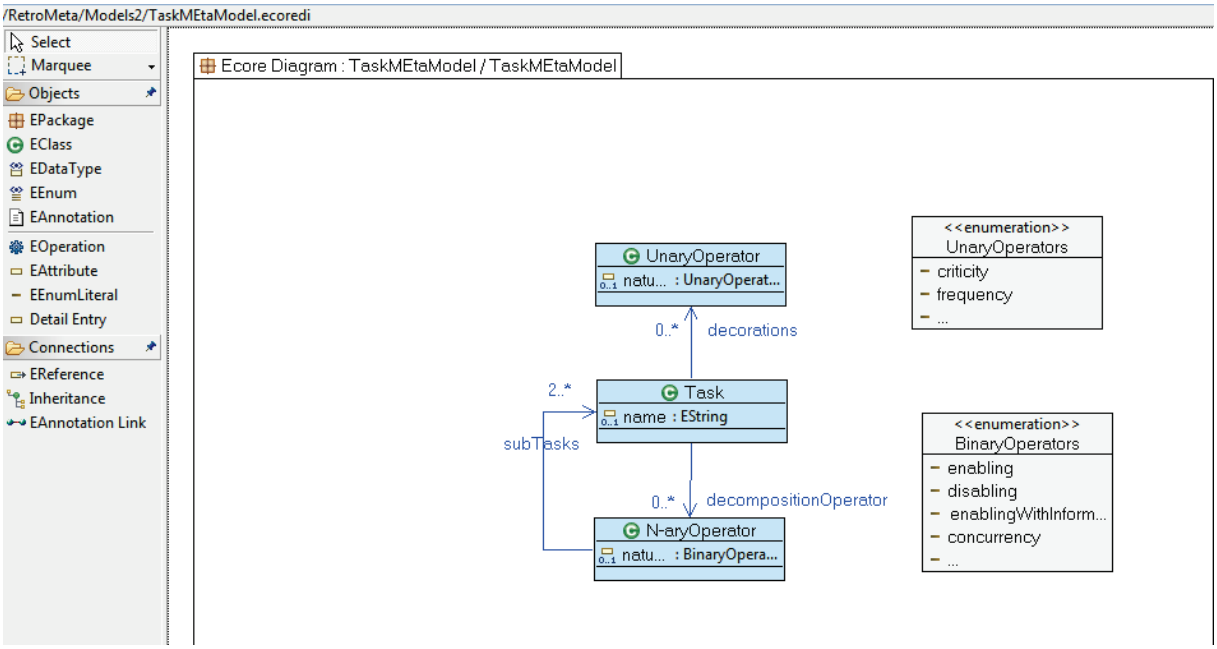

Figure 14. IHM d'édition du *metamodèle* de tâche (**opcased**).

3.1.4 *Trans-IHM ou τ-IHM*

Une *trans-IHM* ou *τ-IHM* est l'IHM du modèle de la transformation. On a plusieurs *trans-IHM* selon que ces dernières modélisent les règles de transformation, la trace de la transformation, etc. Dans la figure 15 on représente le patron type d'une *trans-IHM* : l'IHM du modèle abstrait de transformation.

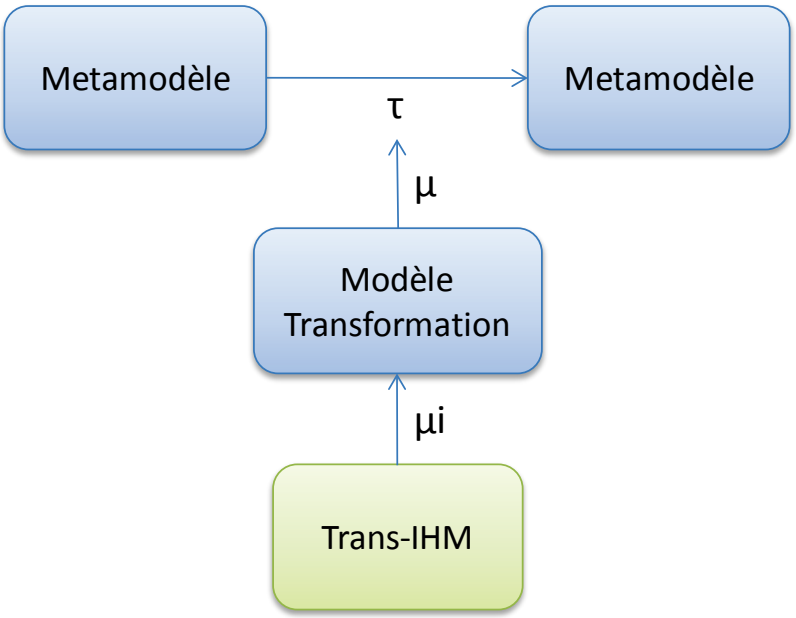

Figure 15. Patron type d'une trans-IHM.

De la même, les *meta et extra-IHM,* les *trans-IHM* peuvent être spécifiques à un utilisateur ou une approche. A l'instar des *DSL*, on parle de « langage de transformation spécifique à un domaine » (*DSTL* : Domain Specific Transformation Languages). Les feuilles de style CSS sont un langage de transformation spécifique au domaine des IHM ciblant l'aspect final et le rendu. Le langage ATL que nous utilisons est, en revanche, un langage généraliste de transformation.

Dans le présent travail, nous élaborons une *trans-IHM* spécifique à la génération d'IHM. Cette *trans-IHM* expose les concepts manipulés : des types de tâches vers des interacteurs. La *trans-IHM* de la figure 16 permet de sélectionner quel type d'interacteur on désire manipuler en fonction d'un type de tâche. L'utilisateur de cette *trans-IHM* se voit guidé dans son choix d'interacteur (liste centrale de la figure 16) : il est réduit en fonction du type de tâche qu'il sélectionne (liste de gauche figure 16). La troisième liste (à droite) permet d'ajouter un paramètre à la transformation, ici la couleur de l'interacteur. L'IHM proposée n'est qu'une maquette de ce que nous sommes en train de réaliser dans le cadre de cette thèse.

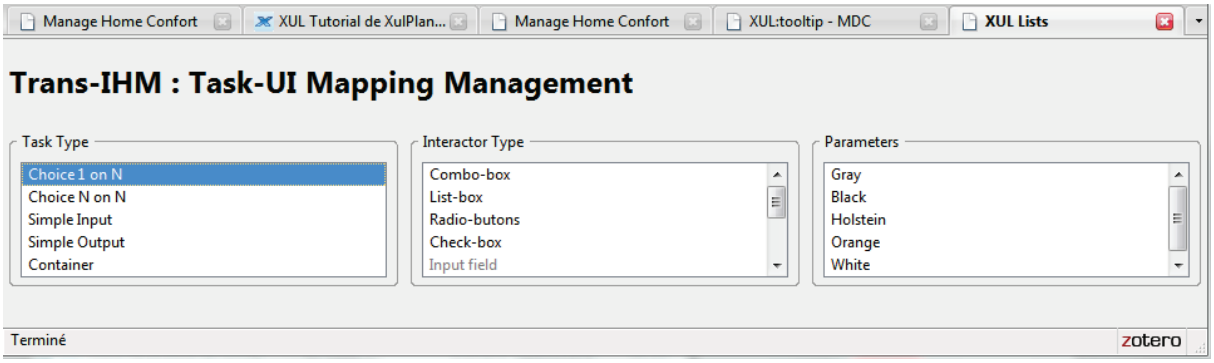

Figure 16. *Trans-IHM* représentant une DSTL pour la génération d'IHM à partir de type de tâches.

La portée de cette *trans-IHM* reste du domaine d'un concepteur d'IHM. Cependant ce type d'utilisateur n'a pas spécifiquement besoin d'être expert d'un nouveau langage de transformation à partir du moment où il sait concevoir les modèles de l'IHM.

L'éditeur de transformation ATL qui permet d'écrire des règles de transformation est aussi une *trans-IHM* (textuelle). Cette IHM (figure 17), éditeur d'ATL, représente un modèle de transformation.

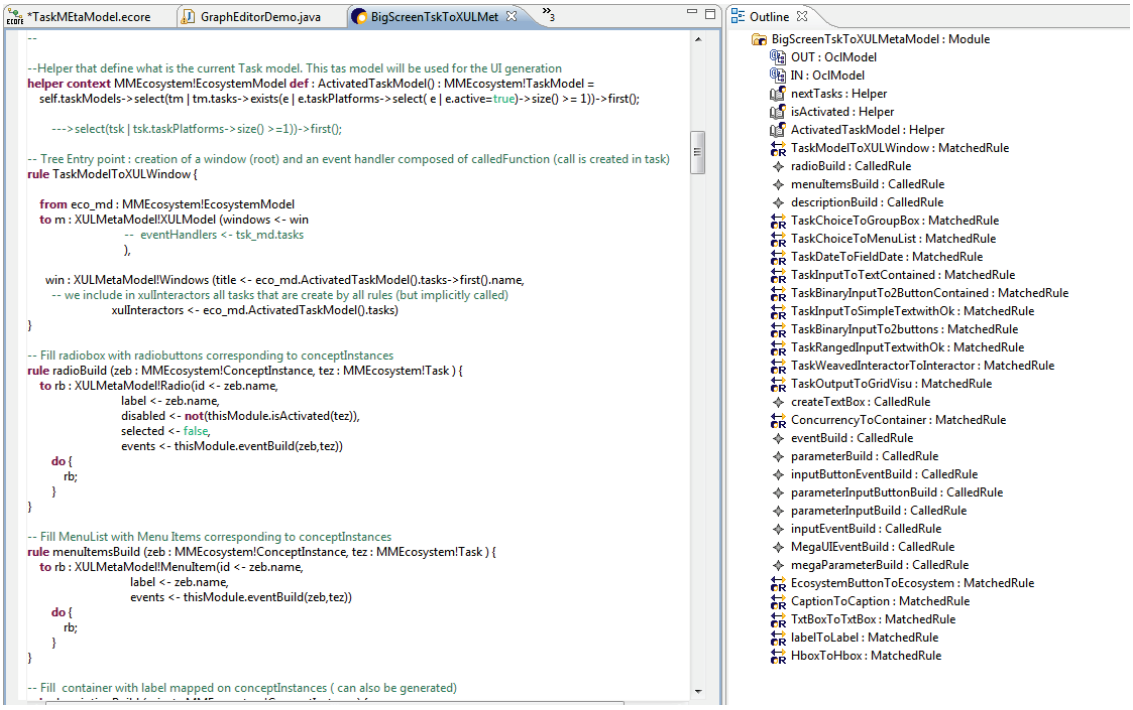

Figure 17. Editeur de transformation (**Eclipse** + **ATL**) : *trans-IHM* pour le développeur de *transformations.*

Un éditeur de règles de transformation ATL est utile pour l'expert en transformation de modèle mais il ne convient pas forcément aux concepteurs de tout niveau, tout comme aux utilisateurs finaux. Il faut réaliser d'autres *trans-IHM* avec des syntaxes concrètes différentes. Pour notre application domotique il serait intéressant de proposer à l'utilisateur un moyen de paramétrer les transformations de modèles par une *trans-IHM* adaptée à ses connaissances.

3.2 Une mega-IHM

3.2.1 Megamodèle & Mega-IHM : les relations

Le concept de mega-IHM, tout comme celui de *megamodèle,* vise à unifier les représentations des modèles (vue IHM) autour des différentes relations de l'IDM (χ, μ, ε, τ) et les relations que nous avons introduites (χi, μi). La figure 15 est une vue globale des patrons présentés dans la section précédente, mettant en relation des IHM et les modèles abstraits qu'elles représentent. On a précisé à quel acteur se dédie l'IHM considérée.

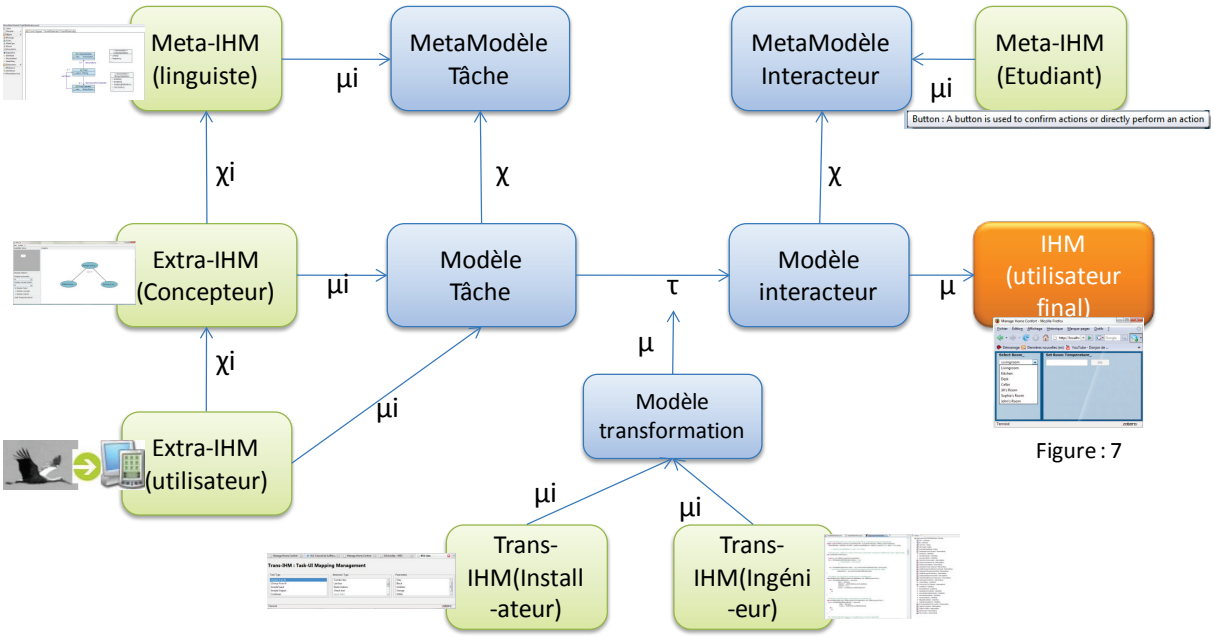

Figure : 7

Figure 18. *Megamodèle* pour les IHM et les IHM associées aux différents utilisateurs. Toutes les relations de la mega-IHM n'ont pas été représentées.

3.2.2 Une relation réflexive pour l'IHM

Toutes les IHM présentées ici ont, elles aussi, leur propre *modèle* de tâches : l'extra-IHM (figure 9) peut alors se décrire elle-même (réflexivité). Dans tous les cas, les IHM résumées dans la figure 18 sont elles-mêmes décrites des modèles d'IHM (Modèle de tâche, d'interacteur par exemple) à l'instar de la figure 7. On pourrait donc remplacer dans la figure 18, l'IHM pour un utilisateur final par n'importe quelle *meta, extra* ou *trans*-IHM.

La figure 19 présente la méga-IHM mettant ensemble les mega, extra, trans-IHM unifiées par les relations de représentation χi, μi. Le rôle de ces IHM est défini de manière relative par leurs relations. L'IHM au milieu à gauche, qui possède une relation μ réflexive, est une extra-IHM générique contenant la modélisation d'extra-IHM : comme on modélise des IHM, toutes les extra-IHM sont conformes à la même meta-IHM (figure 19 en haut à gauche).

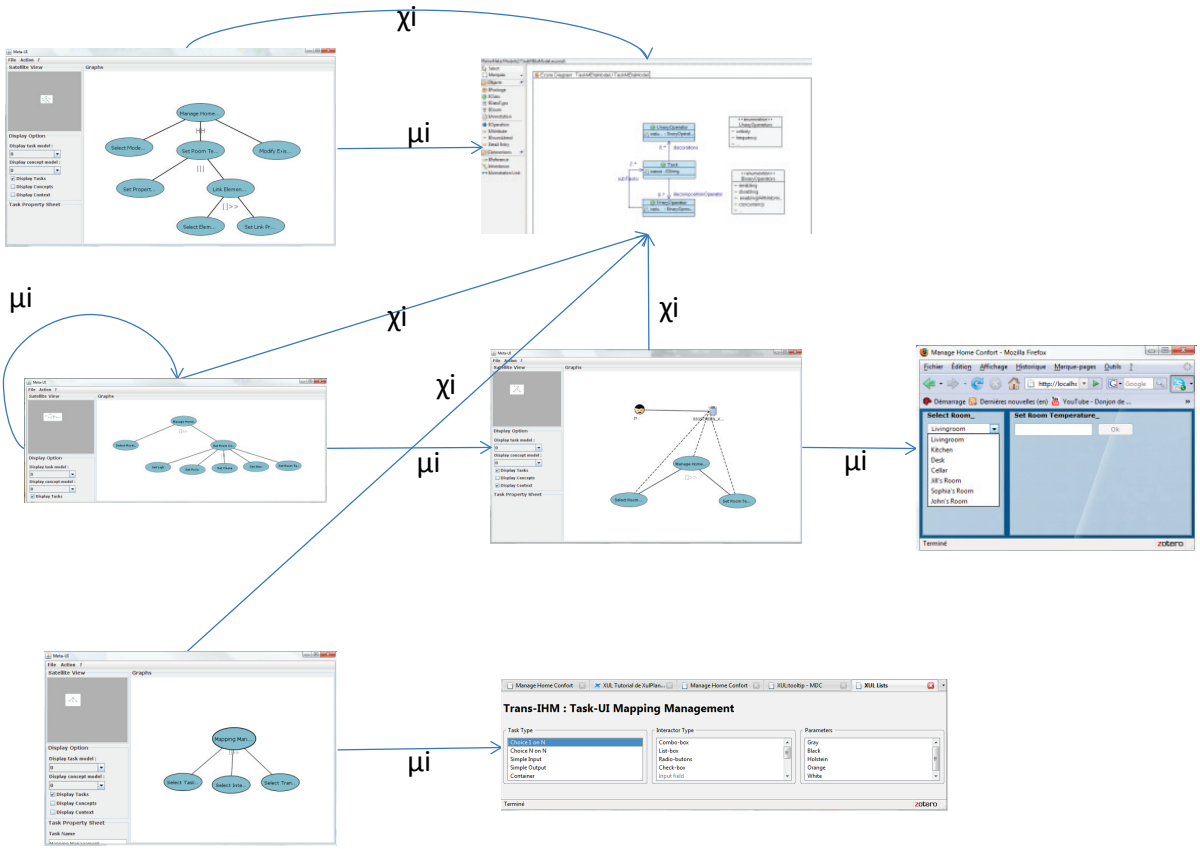

Figure. 19 Megamodèle mettant en relation les IHM de malléabilité

Une IHM regroupant tous ces points vus (ainsi que leurs relations) est une mega-IHM. Elle représente alors tout le panel des points de malléabilité sur lesquels on peut agir pour rendre l'IHM malléable.

4 Approches connexes et conséquences

Un système de modélisation/génération manipulant des langages permet de réduire le coût de réalisation d'outils de conception. De manière générale, donner accès au niveau de modélisation supérieur réduit les temps de modification, débogage, etc. : c'est la volonté de tout spécialiste de l'ingénierie logicielle.

En conséquence, ce concept de vue ou IHM spécifique n'est pas nouveau. Nous avons ici un système montrant une approche similaire : cependant tous les aspects que nous avons exposés dans la mega-IHM ne sont pas présents ici. Nous verrons aussi conceptuellement les impacts potentiels d'une réalisation complète de la mega-IHM pour ce système.

4.1 Concept similaire existant, limitations et avenir

Les langages généralistes, à l'instar des lignes de commandes, possèdent un ensemble complet d'instructions. Cependant, il faut parcourir une importante documentation et faire des formations avant de pouvoir se dire « expert » dans un langage. Bien souvent la forme (IHM) de ces langages est

abstraite par rapport au problème considéré. Aussi pour les spécialistes des domaines, non informaticiens, utiliser ce type « d'IHM » n'est pas raisonnable.

Un langage spécifique à un domaine va être en revanche plus directif, avoir une syntaxe plus représentative de ce qui est réalisé par la machine. Souvent pour des raisons de temps d'apprentissage, les commandes réalisables sont limitées au domaine d'application. Dans ce type de langage on fera aussi entrer les API, qui sont des spécialisations d'un langage généraliste pour un domaine donné avec un vocabulaire précis.

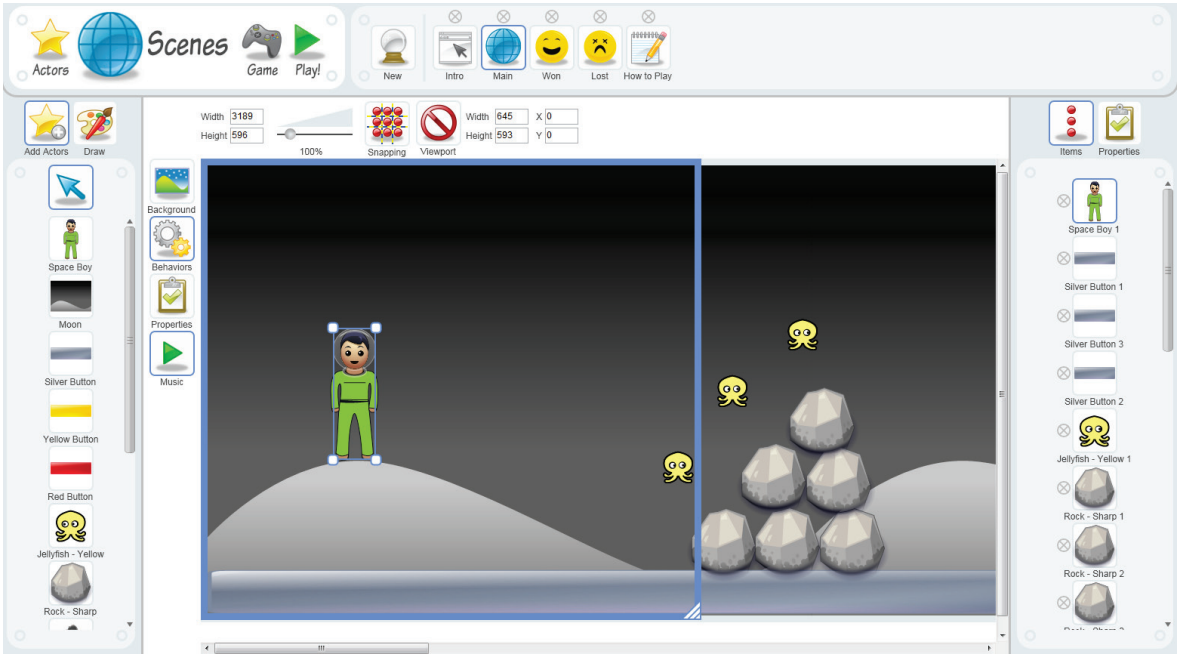

Figure 20. Langage graphique spécifique au domaine des jeux vidéo dans Microsoft Popfly [Popfly].

Le langage de description de l'IHM du jeu (figure 20) propose une composition personnelle de la scène d'interaction du jeu par des non spécialistes en informatique. Le comportement de certains objets est fixé ou éditable dans certaines bornes. Les mécanismes d'extension sont limités mais guident rapidement l'utilisateur dans ses choix. Le système Popfly possède aussi une IHM programmatique (pour le débogage) et pour les utilisateurs avancés et spécialistes de l'informatique (figure 21).

GetActors

Parameters:

Property	Type	Description
Actors	Array of String	List of actor names to retrieve

Returns:

Type	Description
List of Actors	List of retrieved actors based on the actor names passed in

Retrieves a list of actor instances based on the list of actor names passed in.

JavaScript:

```
var curScene = Game.CurrentScene;
var actors = curScene.GetActors(["Spaceship 1", "Spaceship 2"]);
```

Figure 21. Langage textuel (C#) spécifique au domaine des jeux vidéo dans Microsoft Popfly.

Ce système convient bien aux utilisateurs non experts (figure 20) tout comme aux experts (figure 21). Cet éditeur de jeux est une extra-IHM donnant le contrôle sur l'interaction du jeu vidéo. La figure 22 montre la couverture de Popfly par rapport aux patrons *mega-IHM*.

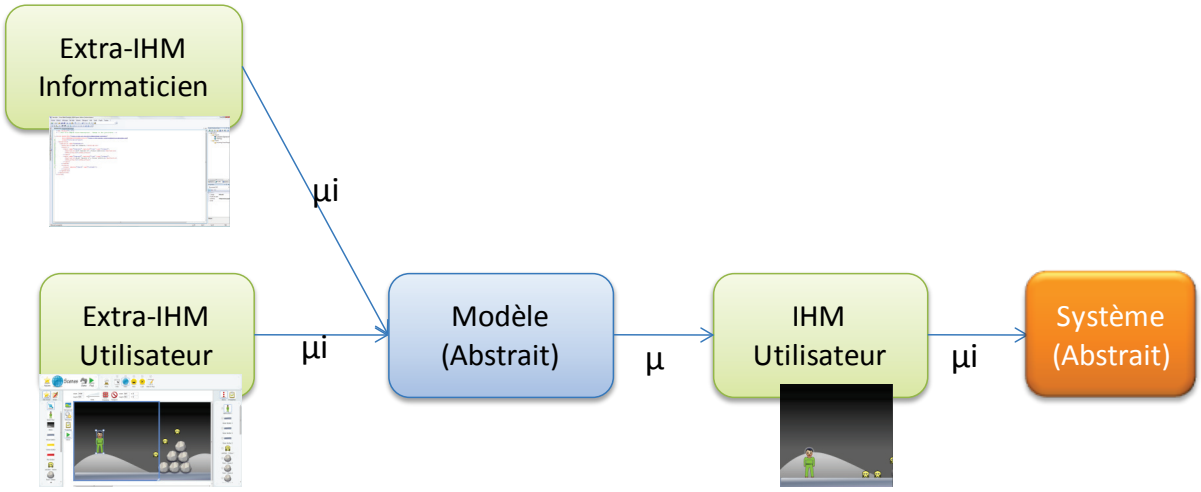

Figure 22. Couverture de Popfly par rapport à la *mega-IHM.*

Bien souvent les systèmes dédiés, aussi puissants soient ils, voient leur usage freiné par le manque de possibilités d'extension. Bien souvent l'utilisateur veut faire une chose proche de ce qui est proposé par l'outil sans que cette fonctionnalité soit disponible. Parfois ce sont les systèmes d'extension qui ne sont pas clairs ou trop complexes à mettre en œuvre. Dans le cas d'API cela peut prendre du temps à une communauté d'ajouter des fonctionnalités qui ne sont pas forcement mises en œuvre par l'utilisateur direct de l'API mais par quelques informaticiens décidés.

Pour l'application Popfly, selon la figure 20, il manque une IHM pour un utilisateur avancé afin de comprendre et modifier/étendre les possibilités offertes par l'éditeur de jeux (*extra-IHM*). Ce type d'IHM est typiquement une *meta-IHM* décrivant (χi) le langage spécifique (metamodèle) de jeux vidéo de Popfly.

Les IHM que nous proposons (*extra, meta, trans*) apportent les moyens de mettre en œuvre, de manière souple et adaptée à l'utilisateur, la malléabilité (chapitre I). Dans le cadre de systèmes généralistes, la *mega-IHM* peut aussi servir à une meilleure documentation/compréhension de la part de l'utilisateur. Ici nous ne voyons pas les outils pour mettre en œuvre cette souplesse mais nous considérons l'utilisateur comme entité première.

Il ne s'agit pas de fournir une API basée sur un langage de programmation (**C**, **JAVA**, etc.) à un spécialiste du design (automobile, site internet, etc.) s'il veut ajouter, modifier quelques fonctionnalités sous-jacentes à son système de conception. En revanche il faut lui donner une IHM (graphique dans cet exemple) lui permettant de tels ajouts.

Une première conséquence intéressante, d'un point de vue Génie Logiciel, réside dans la réduction du temps de développement des langages. En effet, on permet à l'utilisateur d'accéder aux éléments du langage qu'il utilise à un plus bas niveau (figure 23).

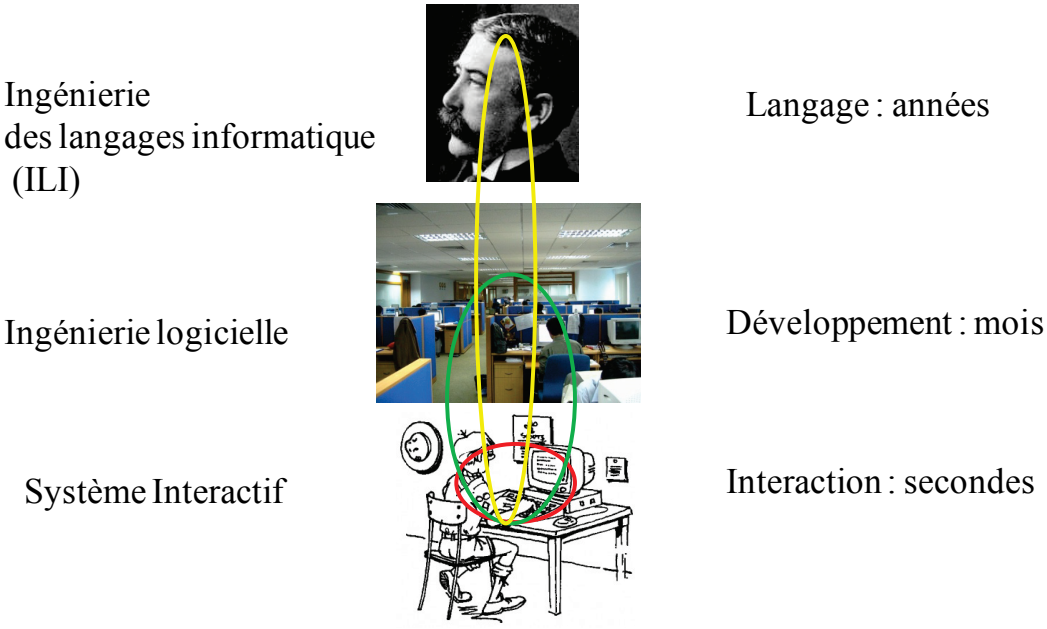

Figure 23. Cycle de vie : des langages informatiques à l'interaction.

Les cycles de réalisation (figure 23) représentent le temps que prend chacune des phases en fonction des acteurs (utilisateur final, développeur, ou linguiste). La *mega-IHM,* en donnant accès à un acteur d'un niveau inférieur à un niveau supérieur de modélisation permet, dans l'idéal, de réduire

l'évolution d'un langage (en termes d'années) à quelques secondes (en termes d'interaction dans une IHM). Les langages de programmation sont enfin reconnus comme ce qu'ils ont toujours été : des IHM : IHM Homme-Machine puis IHM Homme-Modèle.

En restant basé sur la conception centrée utilisateur, de tels systèmes malléables peuvent avoir le même succès qu'ont eu les IHM pour simplifier l'accès de l'informatique à un plus grand public.

4.2 Une « ouverture » des savoirs et savoir-faire

En IHM, le développement par l'utilisateur final est un mouvement émergent et un véritable défi pour la recherche. La *mega-IHM* est un premier pas dans cette direction : elle offre la possibilité pour l'utilisateur d'obtenir les moyens de reconfigurer son espace d'interaction.

Un important travail est encore à réaliser en termes d'ingénierie de l'IHM pour donner les bonnes représentations de ce que l'utilisateur peut modifier. Des avancées prometteuses ont déjà été accomplies dans le domaine des jeux vidéo, du web, etc. Par exemple l'application web Microsoft Popfly permet de créer des jeux et des mashups sans avoir de compétences particulières en programmation.

La notion de *mega-IHM* ouvre alors les portes à une classification des IHM orientées modèles par rapport à l'utilisateur auquel elles sont destinées. On peut aussi dériver de ce modèle les IHM adéquates selon le métier de l'utilisateur : un ergonome veut paramétrer et tester les transformations de modèles selon des critères d'utilisabilité. Cette partie est étudiée dans le chapitre suivant consacré à la plasticité des IHM.

Lorsqu'il produit quelque chose avec de tels outils, l'utilisateur ne sait pas toujours ce qu'il réalise, quelles sont les limites et les effets exacts. C'est en lui donnant les moyens d'observer directement les résultats qu'on peut lui livrer des indications sur les effets de sa modélisation. On va donc vers du prototypage rapide dynamique à l'exécution. C'est le cas de la réalisation technique présentée dans cette thèse dans le **chapitre VIII**.

Ce dernier point n'est pas sans rappeler le début de l'informatique, lorsque les « codeurs » et « hackers » adeptes de l'assembleur, étaient farouchement opposés aux langages de haut niveau et aux générateurs de code. Il faut dire qu'à l'époque les premiers générateurs produisaient du code moins performant que celui écrit par un bon codeur. La situation a changé peu à peu, mais selon les critères sélectionnés et le domaine concerné, l'assembleur a longtemps cohabité avec les autres langages.

5 Synthèse

Comme nous l'avons vu les communautés GL et IHM n'ont pas nécessairement les même valeurs, ce qui pose parfois des problèmes de compréhension. En dehors du clivage communautaire, souvent implicite mais néanmoins perceptible, nous avons cherché dans ce chapitre à faire la liaison entre IDM et IHM et ce, sous la forme de mega-IHM, combinaison de *megamodele* et d'IHM.

Plus particulièrement, nous avons vu qu'il était utile de définir un intermédiaire entre IHM originales et IHM systématiques. C'est le concept d'IHM malléables défini en introduction de cette thèse. Alors que les IHM originales sont entièrement conçues par des humains et que les IHM systématiques sont générées en « boîte noire », les IHM malléables proposent au concepteur ou utilisateur de modifier l'IHM en lui fournissant différents types d'IHM. Ces IHM doivent être adaptées aux préoccupations et aux compétences de celui qui les utilise et les manipule.

Cette idée repose sur quelques concepts fondamentaux que nous avons énoncés :

- le concept de **modèle**, habituellement réservé au concepteur (niveau M1), fait aussi du sens aux autres niveaux dans la pyramide des acteurs. Cependant, nous avons décidé d'utiliser le mot modèle comme un artefact abstrait dédié aux outils d'IDM. Le modèle peut être général ou décomposé en sous modèles, proposant ainsi une perspective spécialisée pour un utilisateur particulier.

- le concept de **vue ou IHM** représentant ce modèle et permettant l'interaction avec celui-ci. L'IHM est complémentaire du modèle à partir duquel elle propose à l'utilisateur une solution appropriée à ses compétences pour réaliser ses objectifs ou comprendre le fonctionnement. Elle est en quelque sorte la syntaxe concrète du modèle que nous voyons ici comme abstrait.

En croisant les relations entre les vues et les modèles on obtient une taxonomie d'IHM de malléabilité : extra-IHM pour les IHM manipulant les modèles, *meta-IHM* pour les IHM manipulant les metamodèles et enfin *trans-IHM* pour l'IHM de la transformation : l'IHM représentant l'ensemble du *megamodèle* est la *mega-IHM*

Les différentes IHM de malléabilité (ou de configurations) ne sont pas des concepts fondamentalement nouveau. C'est en effet déjà une réalité pour l'industrie du logiciel car cela est un problème pertinent en informatique. Nous avons tenu dans ce travail à poser clairement les principes de *mega-IHM* comme un pont entre IDM et IHM. C'est un concept prometteur qui pourrait être appliqué plus en profondeur pour améliorer les outils et approches tant industriels qu'universitaires.

Chapitre VII : Malléabilité : un outil pour la plasticité

La notion de *mega-IHM* nous amène logiquement vers la malléabilité des IHM. En effet cette *mega-IHM* donne accès en lecture et/ou écriture à tous les éléments qui peuvent agir sur le système et le rendre malléable à souhait. La malléabilité est utilisable dans différents domaines, pour différentes préoccupations et, comme nous l'avons vu dans le chapitre précédent, par tous les acteurs du logiciel.

Dans le cadre de cette thèse, la malléabilité est un moyen d'assurer la plasticité des IHM. Nous nous attachons ici à définir plus précisément la plasticité en termes de points de variation sur les éléments de modélisation. Nous verrons comment, en agissant, en sélectionnant les savoirs et savoir-faire en IHM, la malléabilité peut donner un cadre de travail pour assurer la plasticité.

Ce chapitre est organisé en deux parties : la première partie traite de l'adaptation au contexte d'utilisation et d'ingénierie des IHM plastiques. La seconde partie porte sur la caractérisation des moyens d'adaptation donnés à l'utilisateur (*mega-IHM*) pour les contextes d'ingénierie et d'utilisation.

1 Plasticité des IHM dirigée par les modèles

Dans cette partie, nous développons notre proposition : traiter la plasticité par la malléabilité dirigée par les modèles. Les modèles des IHM sont les points de variation qui nous permettent de rendre les IHM adaptables suivant les propriétés propres à l'interaction.

Rappelons que la plasticité traite de l'adaptation des IHM au contexte d'utilisation par deux leviers : « remodelage » et « redistribution ». En conséquence, nous commençons notre explication par détailler la prise en compte des modèles de contexte d'utilisation. Par la suite, nous présentons une définition dirigée par les modèles des deux leviers de plasticité : nous mettons en avant les points de malléabilité nécessaires pour assurer l'adaptation selon le « remodelage » et la « redistribution ». Nous parlons aussi des « plages de valeurs » limites dans lesquelles cette adaptation doit s'effectuer : les propriétés d'ergonomie. Enfin, nous introduisons les principes nécessaires pour mettre en œuvre une infrastructure de malléabilité pour la plasticité.

1.1 Megamodèle pour la plasticité

Nous avons défini un ensemble de metamodèles pour caractériser les systèmes interactifs sensibles au contexte : un megamodèle pour les IHM. La plupart de ces metamodèles sont plus ou moins directement inspirée de l'état de l'art du **chapitre II**. Une partie du *megamodèle* regroupe les

metamodèles décrivant l'IHM ainsi que leurs mises en correspondance. Pour des raisons de simplification, nous ne considérons dans nos explications que les modèles de tâche et d'interacteur.

1.1.1 Contexte (d'utilisation et d'ingénierie)

Le contexte est un point essentiel pour la plasticité. Il caractérise tout ce qu'on veut ou peut capter du triplet <utilisateur, plate-forme, environnement>. Evidemment un ingénieur peut avoir envie de réutiliser les modèles de contexte proposés. Cependant les modèles de l'état de l'art [Wagelaar, et al., 2005, Limbourg, et al., 2004] ne font pas toujours consensus. Il faut décrire les éléments pertinents pour une application donnée : par exemple on ne s'occupe pas de la luminosité pour des IHM vocales. On veut aussi être capable de savoir quels éléments sont « captés », quels éléments sont « déduits » et lesquels ne sont que « conceptuels ». Bien souvent, les (meta)modèles proposés sont un amalgame de propriétés non-captables ou non-observables. A l'inverse, les approches plus pragmatiques ne marchent que pour un système donné ou une préoccupation particulière.

Dans ce travail, nous présentons une approche générique, dirigée par les modèles comme c'est le cas tout au long de cette thèse. A l'instar de ce qui est proposé dans [Rey, 2005], nous donnons une manière de construire et d'annoter les contextes mais cette fois dirigée par les modèles.

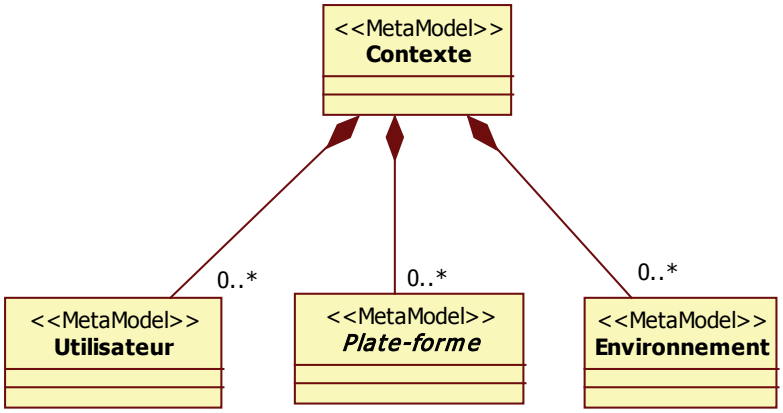

Figure 1. Metamodèle de contexte « d'utilisation » et ses trois composantes.

Dans le metamodèle de contexte proposé, nous reprenons la définition de contexte pour la plasticité, à savoir le triplet *metamodèles* <Plate-forme, Environnement, Utilisateur> (figure 1). Pour chacun des éléments qui composent ces trois *metamodèles* (*ElementAnnotable*) nous proposons plusieurs annotations : conceptuelle, calculée (déduction des actions utilisateurs par exemple) avec un niveau de certitude, ou captée (figure 2).

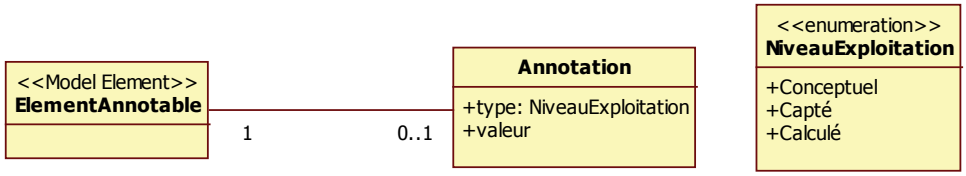

Figure 2. Annotation d'éléments du contexte.

Dans le cas d'un simple serveur internet, on peut récupérer les données fournies par le navigateur lors de la connexion. Le modèle de plate-forme est alors très simple : il contient le type de navigateur (Mozzila firefox, Internet Explorer, Safari). Dans ce cas il est fortement probable qu'on n'ait pas d'adaptation à faire sur ces navigateurs, lesquels peuvent interpréter de l'HTML. Cependant, sur petit écran (IE pour PDA, MiniMo : Mozzila pour PDA, etc.) il faut la plupart du temps adapter l'affichage du contenu à la taille de l'écran. Ce qui nous intéresse dans ce cas, ce n'est pas la « marque du navigateur » mais bien sûr le type de plate-forme sur laquelle il s'exécute (PDA, PC, etc.). Dans la figure 2 nous proposons un *metamodèle* de contexte annoté pour cet exemple orienté serveur-web.

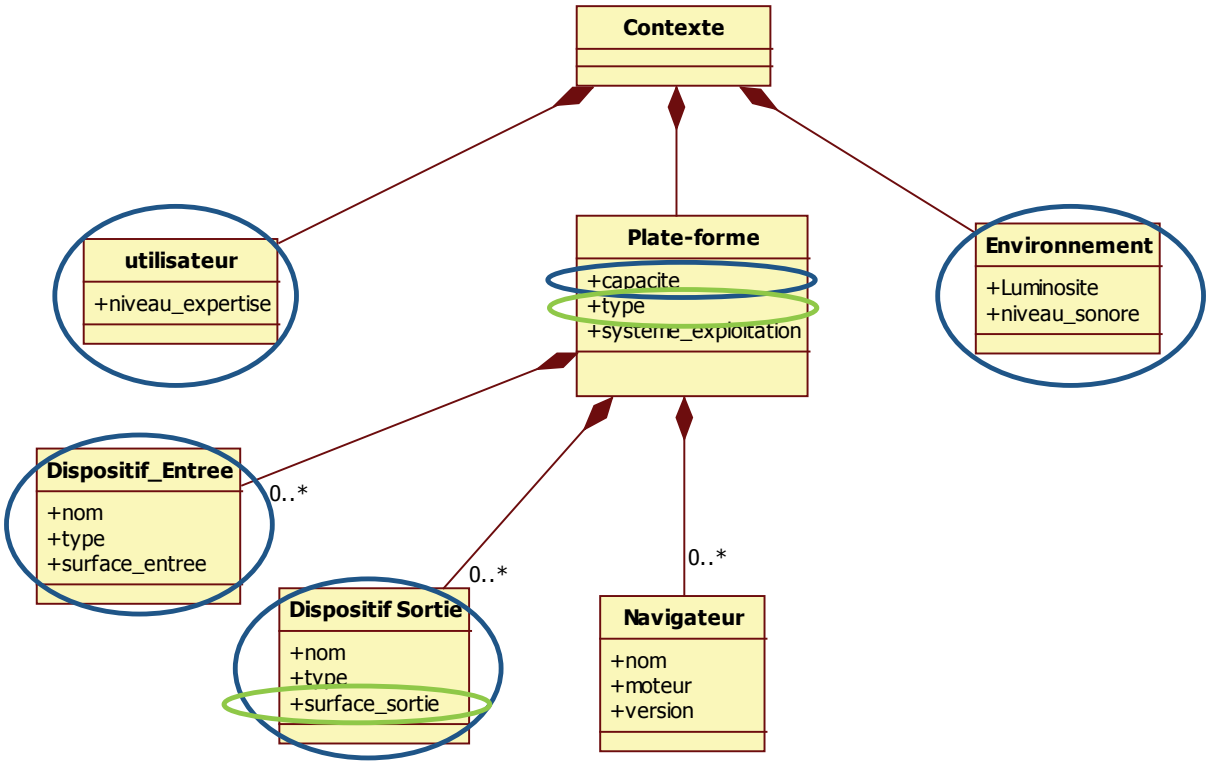

Figure 3. Un exemple de modèle de contexte annoté par des cercles.

Dans cet exemple (figure 3), le contexte proposé ne peut être donné qu'à partir des données fournies par le navigateur. A partir de celui-ci on peut obtenir (capter) un ensemble d'informations (classes et

attributs non encerclés) quant au moteur, au nom du navigateur, au système d'exploitation de la plate-forme matérielle, etc. Le type de la plate-forme PC-PDA, tout comme la surface d'affichage (attribut surface de sortie de la classe Dispositif Sortie), est calculé à partir d'une base de connaissances sur les navigateurs web.

Il importe alors de mettre ce metamodèle de contexte au sein de metamodèles de l'application. C'est-à-dire proposer des mises en correspondance ou relations entre les éléments du contexte et les éléments d'interface. La figure 4 décrit les modèles de tâche et d'interacteur mis en relation avec les éléments du contexte. La notion d'application regroupe un ensemble de tâches (un arbre des tâches complet). L'IHM d'une application quant à elle, est représentée par l'ensemble des interacteurs œuvrant dans cette application.

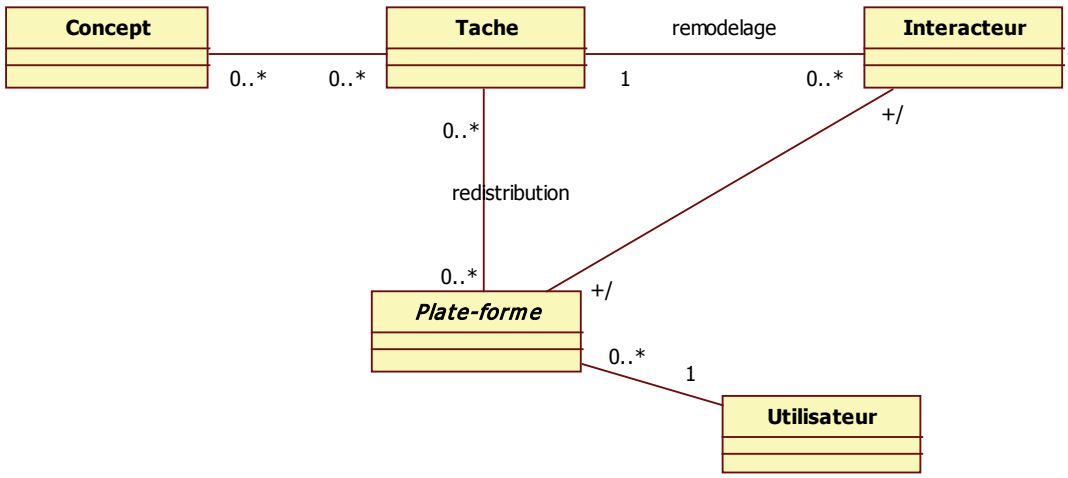

Figure 4. Megamodèle (simplifié) pour la plasticité dirigée par les modèles.

Si on veut faire de la plasticité dirigée par les modèles, on réalise la redistribution et le remodelage d'une IHM en jouant sur les mises en correspondance entre les éléments de la figure 3, notamment les relations du triangle tâches/plates-formes/interacteur.

1.1.2 Redistribution

Pour faire de la distribution et redistribution de l'IHM, nous rendons la relation tâche-plate-forme dynamique. Sur la figure 4 nous reprenons l'exemple de l'introduction : redistribution de l'IHM de gestion de chauffage du PDA vers le PC. Cette figure présente, étape par étape, le diagramme d'instance permettant d'exprimer cette redistribution.

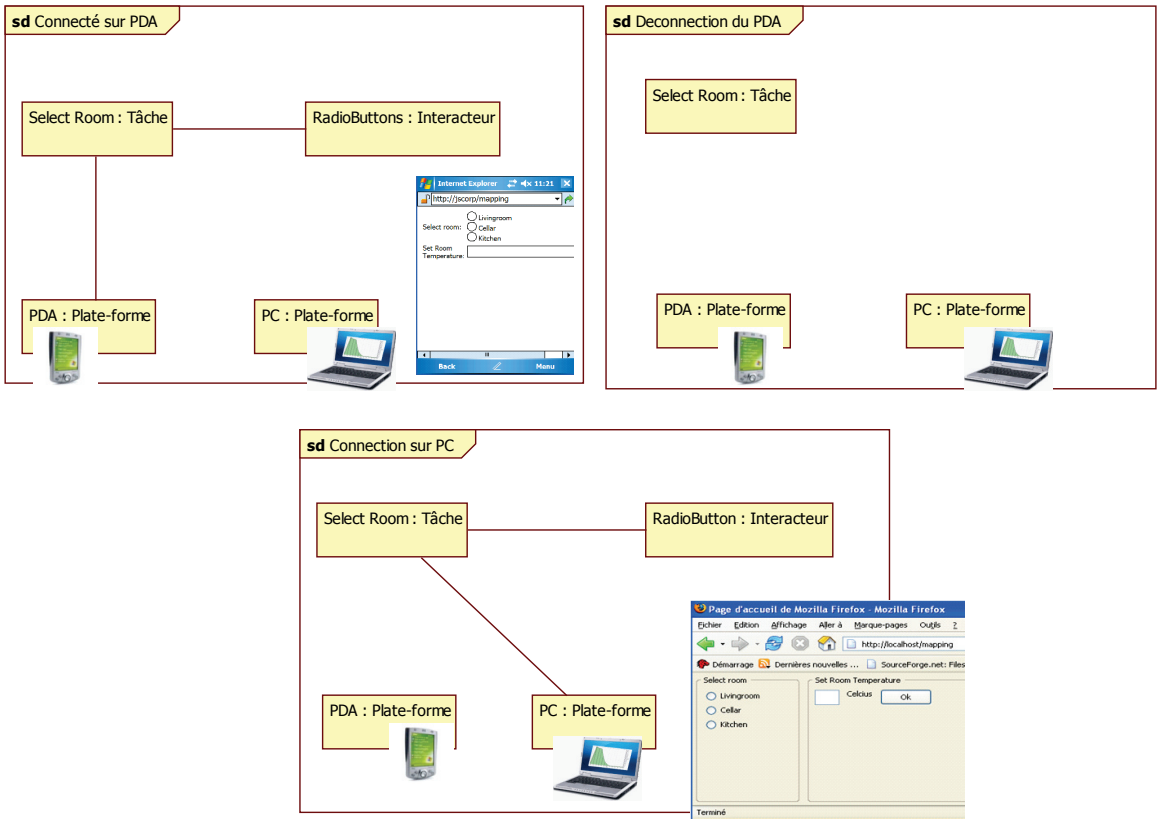

Figure 5. Trois étapes d'un scenario redistribution du PDA vers le PC.

Sur la partie en haut à gauche de la figure 5, le PDA affiche la tâche « *Select Room* » par l'intermédiaire de radio boutons (sur le diagramme une seule tâche et un seul radio bouton sont représentés). La partie droite représente un instant intermédiaire où l'IHM n'est plus mappée sur aucune plate-forme : elle peut être dans un état transitif. Enfin le diagramme inférieur représente un instant où la tâche *select room* est affichée (et redistribuée) sur la plate-forme PC. La redistribution de la figure 4 ne comporte pas de remodelage.

Bien sûr, il existe d'autres formes de distribution à partir d'autres metamodèles : on peut redistribuer des espaces de travail, des interacteurs etc. Le déploiement de telle ou telle partie dépend alors de la relation modèle d'IHM/plate-forme. L'IHM est distribuable à n'importe quel niveau de granularité. Dans notre cas, le modèle de tâche prédomine (figure 4) et la distribution se définit ainsi :

Définition 1 : La distribution est donnée par la relation entre la tâche et la plate-forme. Elle dénote l'exécution d'une tâche sur une plate-forme.

Définition 2 : La redistribution est donnée par la dynamique de la relation entre la tâche et la plate-forme.

1.1.3 Remodelage

Dans une application basée sur la génération d'IHM, c'est-à-dire que la relation tâche-interacteur est réalisée par transformation, la mise en correspondance entre l'interacteur et la plate-forme est dérivée de cette transformation. Nous nous intéressons à ce type d'application générative : il y a *remodelage* si, pour une tâche donnée, on a plusieurs transformations qui livrent des IHM différentes, c'est-à-dire un modèle d'interacteur différent. Par conséquent, le remodelage se fait dans la dynamique de la relation tâche-interacteur. Tout comme la redistribution, on peut considérer d'autres relations entre modèles de l'application, par exemple si on a un modèle d'espace de dialogue en plus. Il n'y a remodelage que si l'IHM présentée à l'utilisateur est différente.

L'exemple de la figure 6 reprend une partie du scénario donné en introduction. On reste sur la même plate-forme d'interaction fixe (relation tâche/plate-forme) ; il n'y a donc pas de redistribution. A gauche, on trouve une IHM classique pour laquelle la tâche de sélection de pièce est donnée par un ensemble de radio boutons. A droite, on vient d'effectuer le remodelage de cette IHM, c'est-à-dire que l'on change l'interacteur pour la tâche de sélection : celle-ci passe de simple radio bouton à une carte cliquable.

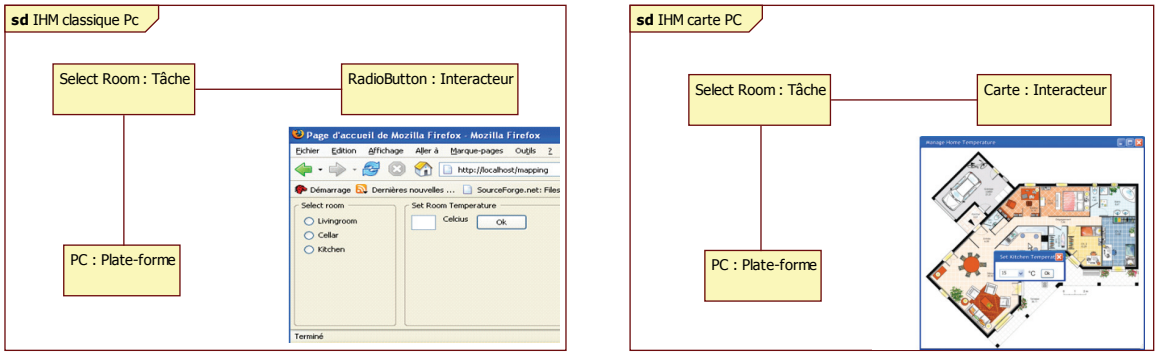

Figure 6. Exemple de remodelage d'une IHM sur PC.

Définition : Le remodelage est donné par la dynamique de la relation entre une tâche et un interacteur.

La dynamique de la relation tâche et son rendu en terme d'interacteur sont donnés par transformation de modèle : génération d'interacteur à partir d'une tâche. On peut donc dire que le

remodelage fait appel aux transformations de modèle, le deuxième point important de notre approche.

1.2 Transformations de modèles

La mise en œuvre de l'adaptation se fait, dans notre cas, par transformation de modèle : c'est une fonction qui prend en paramètres le modèle du contexte (Ctxt) , le(s) modèle(s) du système interactif (MSIn – *megamodèle -)* pour donner un nouveau modèle de système interactif : f(Ctxt,MSIn) = MSIn'.

1.2.1 Impact du contexte sur les transformations

Le modèle du système interactif (MSIn) peut être complet ou non. Dans certains cas il se compose uniquement des spécifications de haut niveau d'abstraction permettant de produire du code : par exemple un arbre de tâche. Mais le MSIn peut être entièrement préconçu jusqu'au modèle d'interacteur, voire du code de l'application.

Le contexte est un paramètre de la fonction de transformation : la conception du contexte est donc parallèle à la conception des transformations. Nous devons donc fournir au concepteur de systèmes plastiques des mécanismes permettant d'assurer la cohésion de l'ensemble (**Chapitre III**): mises en correspondance et transformations d'ordre supérieur.

Nous avons, par exemple, un ensemble de règles de transformation de modèle permettant de générer des IHM vocales ou graphiques selon les capacités de la plate-forme cible, avec une préférence pour le vocal sur les petites machines portables sans clavier adapté (type téléphone). Le metamodèle de contexte utilisé ici permet de prendre en compte le niveau sonore, de manière conceptuelle (comme celui de la figure 2). Si l'on ajoute un capteur de niveau sonore, cet attribut est alors capté. L'ingénieur qui reconçoit le nouveau système peut utiliser cette nouvelle donnée : si le volume sonore dans l'environnement de l'interaction dépasse un certain seuil, la reconnaissance vocale tout comme l'écoute devient impossible.

Pour les règles de transformation, on capitalise les transformations déjà écrites et on ajoute un filtre sur cette règle correspondant au niveau sonore. Ci-dessous un extrait de transformation ATL permettant de générer un interacteur vocal pour une tâche. Entre parenthèse, après Tâche, on ajoute le filtre relatif au capteur de niveau sonore. Pour ce faire, on parcourt le graphe de contexte (de cette tâche) et on vérifie (filtre) que le niveau sonore n'excède pas 10 (décibels) : `t.contexte.environement.niveau_sonore < 10`. Le cas échéant on génère bien un interacteur vocal.

```
Rule TâcheToInteracteurVocal {
```

```
From   t:IHMMegaModele!Tache(t.contexte.environement.niveau_sonore   <
10,…)

To i:IHMMegaModele!InteracteurVocal(…) }
```

Il est intéressant de savoir à quels « filtres » correspondent quels « éléments » du contexte. Ici on reprend le diagramme de la figure 1 et on ajoute l'association entre ce metamodèle et le filtre d'une règle de *transformation* (figure 7) : ceci permet de conserver la trace de ce qui a été réalisé lors de la conception de la règle de *transformation*. De manière simplifiée (et non complète pour des raisons de lisibilité), une règle de transformation est composée d'un patron source (après le mot clé *from*) et d'un patron cible (après le mot clé *to*). Le filtre porte sur le patron source.

Cette relation appliquée sur un *modèle* de contexte particulier aide à la décision du concepteur quant à l'adaptation au contexte, pour voir les conflits entre les éléments : vocal en situation de bruit comme dans notre exemple.

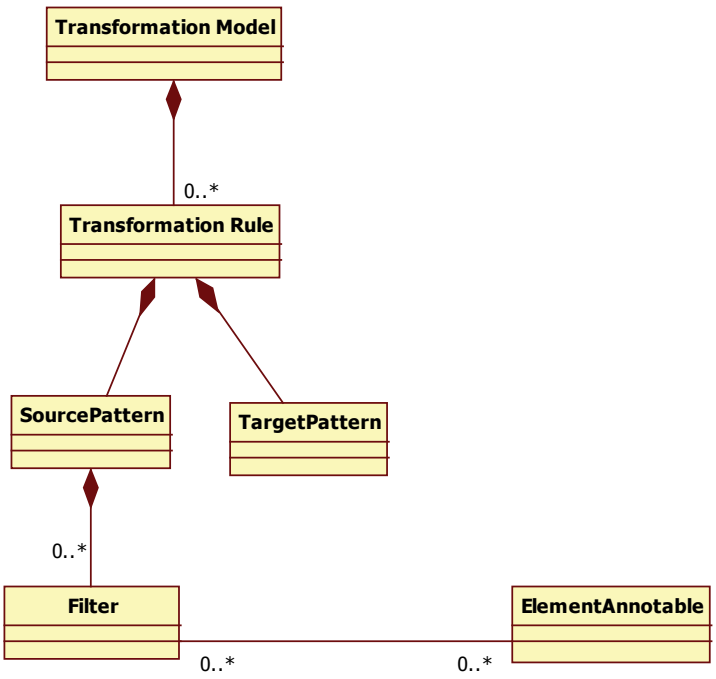

Figure 7. Extrait du *megamodèle* mettant en évidence les relations entre un filtre de règles de transformation et les éléments du contexte.

1.2.2 Réduction de la combinatoire

Il existe d'autres approches que celle que nous avons retenue en matière de capitalisation des savoirs et savoir-faire : par exemple les systèmes à composants.

Dans notre approche, le savoir-faire du concepteur est en quelque sorte « mis en boîte » par des transformations de modèles. Alors que les lignes de produits proposent plusieurs variantes de l'application en fonction du contexte ; nous insistons sur la réduction de la combinatoire de tels systèmes par transformation de modèles : c'est une écriture en intension (fonction) de cet ensemble de variables.

Contrairement à d'autres systèmes basés sur des composants (ou des services) [Balme, 2008] qui permettent de choisir (dynamiquement) une variante prédéveloppée [Demeure, 2007], notre approche, orientée transformation, réduit les coûts de développement de ces N variantes. Certaines variantes ne sont d'ailleurs jamais utilisées, selon l'usage, mais prennent tout de même de l'espace mémoire. Tout comme ces systèmes de sélection de variantes, nous ne nous attendons pas à résoudre des problèmes non prévus à la conception. Mais nous pouvons cependant exprimer un large panel de variables au travers des transformations de modèle : c'est toute la différence entre écrire une fonction (intension) et écrire l'ensemble des couples <x,y> (extension) de cette même fonction.

Les approches à composants ou services ne sont pas antagonistes à notre approche : nous pouvons travailler vers de la génération de composants ou encore profiter des mécanismes de sélection dynamique de composants (par exemple pour ne pas avoir à re-générer une IHM).

Notre approche basée sur la malléabilité permet de définir rapidement des langages de description spécifiques (**chapitre VI**) si ceux-ci ne sont pas déjà pris en compte. Ici il s'agit de répondre au contexte d'ingénierie et de simplifier les usages de conception et d'implémentation de systèmes plastiques. Pour ce faire, nous procurons un cadre de conception de metamodèle de contexte et de transformation spécifique aux problèmes traités.

L'impact de cette réduction de la complexité est surtout important pour les concepteurs d'applications. Dans l'optique du prototypage rapide, il est intéressant de pouvoir travailler avec des transformations à l'exécution. Cela permet de calibrer les modèles, les paramètres de transformations et les transformations elles-mêmes. On facilite ainsi le test des applications et des situations, par exemple, est-ce que la distribution de la partie « sélection de pièce » sur PDA et le « réglage de la température » sur PC est pertinente pour un utilisateur assis dans son salon ?

1.3 Adaptation préservant l'ergonomie

1.3.1 Ergonomie

La plasticité des IHM, c'est aussi le respect d'un ensemble de propriétés lors de l'adaptation. Nous avons proposé un métamodèle (figure 8) permettant de caractériser les transformations de modèles

à tous les niveaux : à savoir de manière *descriptive* c'est-à-dire au niveau de la trace de la transformation et de manière *prédictive* au niveau du modèle de la transformation.

Ce metamodèle nous permet de « typer » les fonctions de transformation (classe *TransformationFunction*) mais aussi leurs traces (bulle *transformationInstances*) en fonction d'un ensemble de propriétés (classe *Property*). Les mises en correspondance (classe *Mapping*) sont décrites à tous niveaux :

- Entre les metamodèles et les éléments de modèle, on parle d'éléments prédictifs permettant la spécification de fonction.
- Entre les instances de modèles, pour représenter le résultat d'une transformation.

L'ensemble des propriétés, caractérisant potentiellement tous ces niveaux, sont de nature descriptive. Elles ne servent pas à décrire le fonctionnement de la transformation.

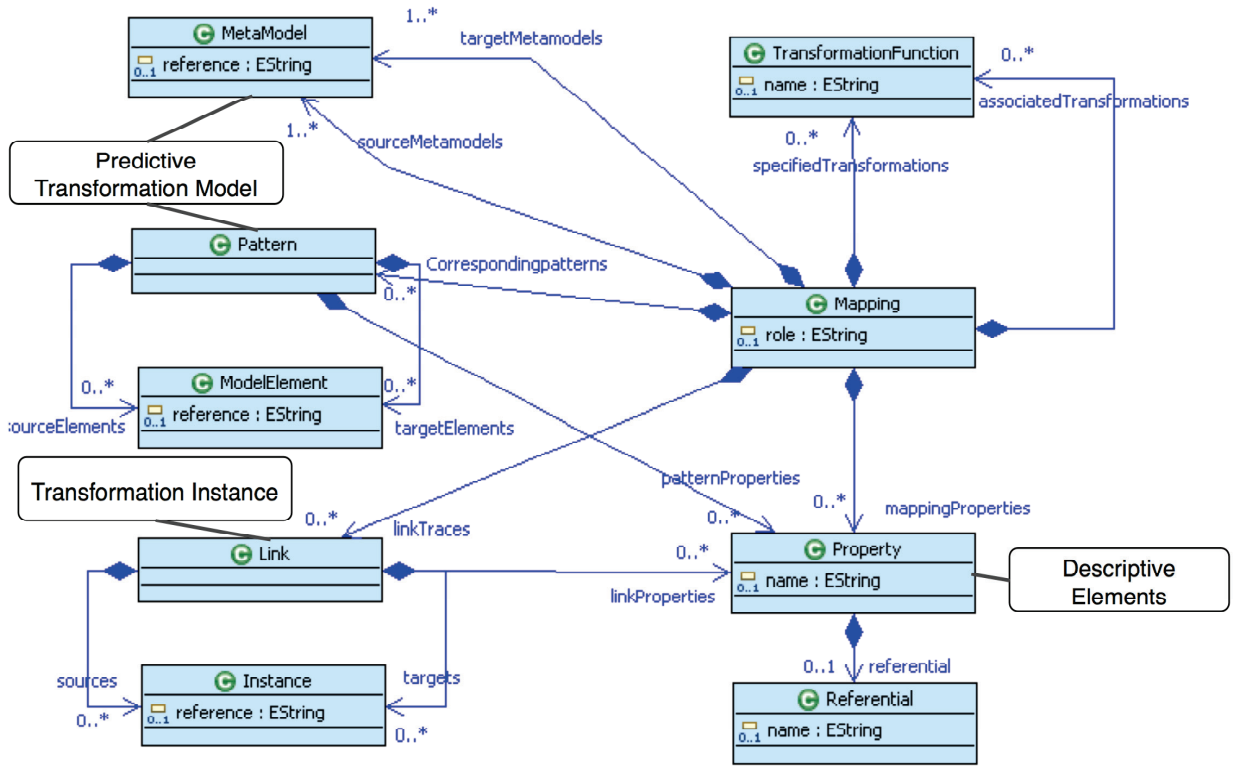

Figure 8. Metamodèle « valeur » pour les mises en correspondance et transformations.

Les mises en correspondance dans **AMW** expriment différents scenarii, par exemple la *comparaison entre metamodèles, la traçabilité, etc*. Ici, nous voulons répondre à deux nouveaux scenarii : l'assurance de valeurs pour la plasticité (classe *property* de la figure 8) et le maintien de la cohérence.

Il faut être capable de lancer des transformations directement à partir des mises en correspondance pour assurer « dynamiquement » la cohérence des modèles d'une IHM. Pour ce faire il faudrait étendre le modèle de mise en correspondance proposé dans **AMW** et offrir un mécanisme pour analyser les impacts et déclencher les transformations associées sur le changement des modèles. Au moment où nous avons réalisé ce travail, nos connaissances ainsi que le développement d'**AMW** étaient insuffisant. Nous avons donc choisi de réaliser notre propre metamodèle.

En spécialisant pour l'IHM ce metamodèle de mise en correspondance, nous qualifions selon des critères d'utilisabilité [Sottet, et al.,2006] les relations (transformations, trace, etc.) entre les modèles d'IHM. Ce travail a notamment été relayé et approfondi par [Zhao, et al., 2007].

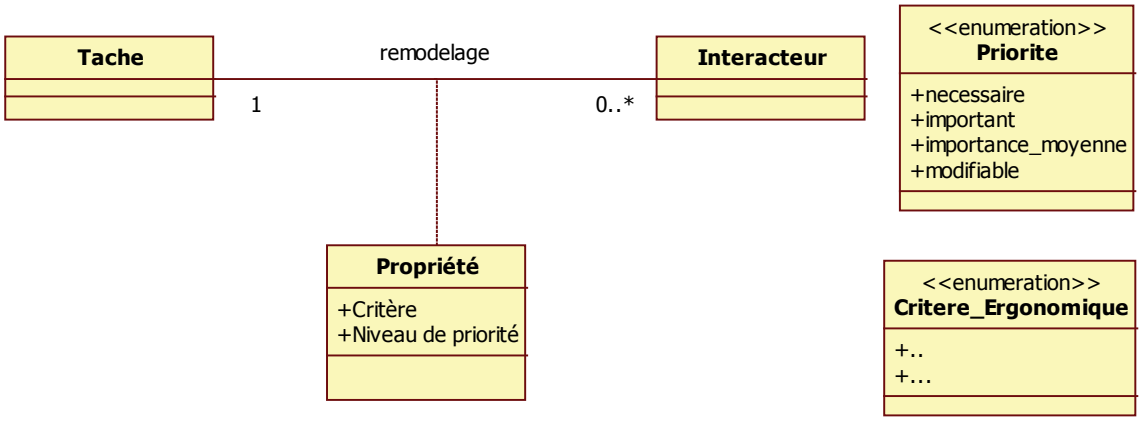

Figure 9. Transformation de tâches vers des interacteurs et propriétés ergonomiques.

La trace de la transformation est importante lorsqu'on veut raisonner pour adapter ou reconcevoir. Cette trace garde le rationnel de la conception : par exemple si un panel regroupe des informations du même ordre, il est fort probable que lors de l'adaptation il faudra garder un conteneur commun pour ces informations. L'adaptation peut se jouer à la fois sur le contexte et sur le rationnel de conception. Le concepteur définit un niveau de priorités pour les critères qu'il souhaite voir perdurer lors de l'adaptation (Figure 9, niveaux de priorités arbitraires). Mais il peut éventuellement ajouter un ensemble d'annotations sur ces propriétés concernant ses choix de conception de manière contemplative.

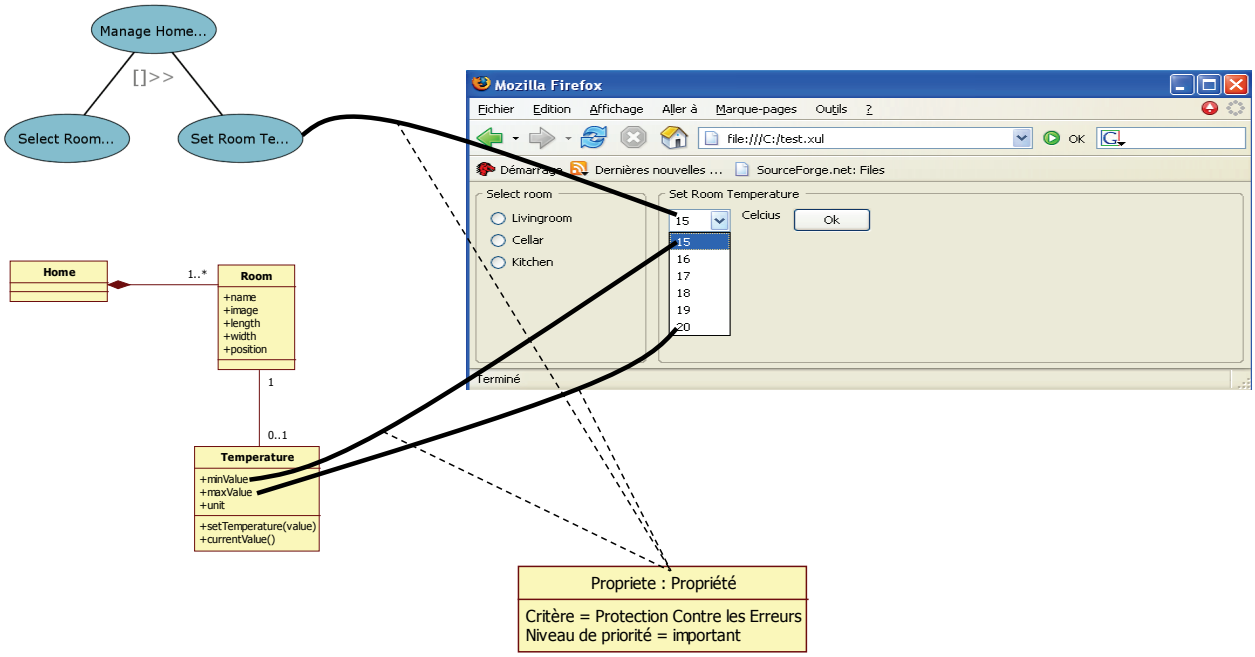

Figure 10. Mise en correspondance entre une tâche (avec ses concepts associés) et des éléments du modèle d'interacteur exposant la propriété de protection contre les erreurs.

Dans notre exemple, figure 10, la tâche « *Set Room Temperature* » est reliée à un interacteur de type liste déroulante, de même que les bornes « min et max » du concept de température. Cette mise en correspondance dénote une propriété de « protection contre les erreurs » car, pour choisir une température, l'utilisateur ne pourra entrer que des valeurs valides dans le thermostat.

Les transformations sont elles aussi caractérisées par ces critères ergonomiques : tant au niveau des paquets de règles qu'au niveau des règles elles-mêmes. Le fait de pouvoir décrire les transformations avec des critères ergonomiques permet d'envisager plusieurs gammes de transformations et des critères pour les choisir. On peut ainsi privilégier un critère au lieu d'un autre (en fonction de l'utilisateur) ou bien avantager le fonctionnel de l'application contre l'ergonomie.

On se donne deux niveaux pour assurer une certaine cohérence et un maintien des requis en termes d'ergonomie et ceci en fonction du contexte :

- Au niveau de la génération de l'IHM (des éventuels modèles associés) par le choix de la transformation la plus appropriée.
- Au niveau des traces des transformations, ou de la conception manuelle avec les liens à conserver prioritairement lors d'une adaptation.

1.3.2 Adaptation contrôlée par un système décisionnel

L'adaptation d'une IHM se fait sur deux plans : celui du choix des transformations à appliquer (politique d'adaptation) et de l'exécution d'une règle de transformation. Les transformations contiennent alors le savoir-faire de conception pour 1 à N contextes garantissant un ensemble de propriétés. Si ces propriétés ne peuvent ou ne doivent plus être valorisées, il faut choisir un autre ensemble de transformations. On peut appuyer notre choix sur les propriétés ergonomiques, telles que nous les avons définies dans la section précédente.

Les décisions quant à la meilleure adaptation à choisir, peuvent être données par un moteur de déduction/inférence spécifique et raisonnant sur le *megamodèle*. Ce dernier comprend à la fois les différents modèles de l'IHM et du contexte mais aussi des modèles de transformation et leurs caractéristiques. Ce système de décisions offre la transformation la plus adaptée : celle qui a les caractéristiques correspondant à une situation précise.

Dans [Interact07] nous avons proposé une approche de conception et re-conception dynamique d'IHM basée sur des préférences utilisateur. Le concept clé est d'inférer, lors de l'exécution, les préférences (et nécessités) de l'utilisateur. Si celui-ci fait beaucoup d'erreurs, il faut choisir les IHM permettant de limiter ses fausses manipulations, c'est-à-dire « lancer » les règles de *transformations* de modèles classées dans « protection contre les erreurs ».

Ce moteur d'inférence sur les actions utilisateur aide à reconcevoir la politique d'adaptation lors de l'exécution : donner la priorité aux propriétés supposées attendues par l'utilisateur en fonction d'un contexte donné. Cette approche originale d'adaptation à l'utilisateur est détaillée dans [Ganneau, et al., 2008].

1.4 Un Intergiciel pour l'adaptation

Dans le domaine des systèmes adaptatifs et adaptables, les articles d'IBM [Kephart, et al.] font référence. On parle notamment de systèmes dits autonomes ou autonomiques. La première catégorie, les systèmes autonomes, sont capables seuls de répondre à différentes problématiques :

- Auto-Configuration : configuration automatique du système en fonction des dispositifs et fonctionnalités logicielles disponibles.
- Auto-Optimisation : Amélioration automatique des performances du système (réseau, calcul,…).
- Autoréparation : diagnostique et réparations automatiques en cas de pertes de charge sur le réseau par exemple.

- ❖ Autoprotection : Anticipation et prévention automatiques des cascades d'erreurs, d'attaques, etc.

Ces critères définis par IBM couvrent des opérations d'adaptation extra-fonctionnelles (c'est-à-dire ne faisant pas partie des fonctionnalités de l'application) et qui peuvent être réalisées par le système. On parle alors de système adaptable (par lui-même sans intervention extérieure).

Si l'informatique autonome, au sens où les quatre critères sont satisfaits, semble utopique dans l'état actuel des connaissances, l'informatique autonomique, réponse réactive à des stimuli, est en revanche déjà une réalité. L'informatique autonomique prend en considération des adaptations qui peuvent être réalisées par des utilisateurs. C'est cette partie qui nous intéresse pour l'adaptation des IHM : la plasticité.

L'informatique autonomique s'accompagne alors d'un *intergiciel* pour l'adaptation. D'un point de vue conceptuel, ces parties logicielles s'appuient sur un ensemble de données à capter et leurs *capteurs* associés. En cas d'alertes sur les données captées, le système doit éventuellement être modifié (*analyseur*).

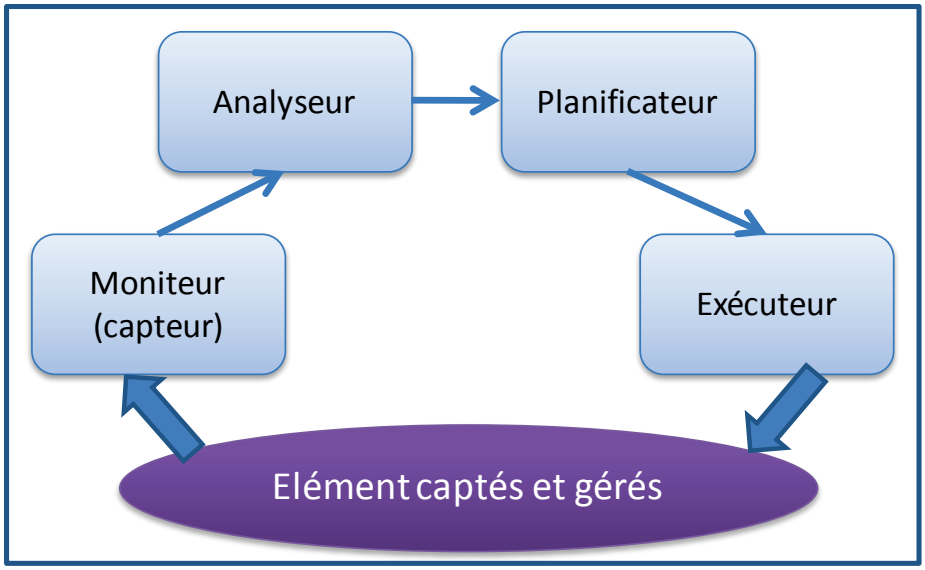

Figure 11. Modèle conceptuel de l'informatique autonomique tiré de [Kephart, et al.].

Nous utilisons cette architecture conceptuelle pour réaliser notre middleware dans le **chapitre VIII**. Le moniteur « nourrit » en informations notre *megamodèle* pour la plasticité (modèles du contexte plus modèles de l'application). L'analyseur se sert de ce *megamodèle* pour trouver la ou les transformations les plus adaptées à la modification du système interactif (si nécessaire). Enfin les phases de planification et d'exécution sont réalisées par le moteur de transformations.

Cependant sur de tels systèmes le contrôle de l'utilisateur est souvent oublié. En IHM il est la notion centrale. Dans la partie suivante nous donnons un cadre IDM pour mettre en œuvre un contrôle explicite pour l'adaptation d'IHM.

2 Contrôle de l'utilisateur

Nous avons posé une série de définitions pour rendre les IHM plastiques par la malléabilité dirigée par les modèles. Ici, nous nous intéressons à une nouvelle dimension : le contrôle de l'adaptation par l'utilisateur. On caractérise ces moyens de contrôle (remodelage et redistribution) en fonction des éléments de modélisation et des IHM de malléabilité. Nous voyons ici une forme non-automatisée d'adaptation pour l'utilisateur par l'utilisateur.

2.1 Contrôle de la plasticité par IHM de malléabilité

Une extra-IHM, peut servir d'environnement de personnalisation d'IHM pour l'utilisateur (l'IHM représente son application de travail, de loisir, etc.). Nous entendons ici par personnalisation, l'adaptation de l'IHM à l'utilisateur par lui-même : c'est une forme de plasticité.

L'extra-IHM donne accès à certains éléments de modélisation, dans le cadre d'applications dirigées par les modèles. De manière plus classique les boutons de fermeture, de réduction d'une fenêtre sont des éléments d'extra-IHM qui contrôlent l'application ou son affichage.

En reprenant la taxonomie de Coutaz [Coutaz, 2006] sur les points de « services génériques » aux IHM de configuration (les deux premiers points concernent directement la plasticité), nous donnons une correspondance avec les éléments de notre *mega-IHM*.

Notons que Coutaz, nomme cette IHM « *meta-IHM »,* terme auquel nous préférons *extra-IHM* ce qui évite la confusion avec l'IHM du metamodèle qui, elle, appelée *meta-IHM*.

- Modèle de Remodelage : *extra-IHM* donnant accès en modification aux interacteurs (ou tout autre modèle de la partie présentation d'une IHM). Egalement toute *trans-IHM* permettant de manipuler les transformations (figure 12).

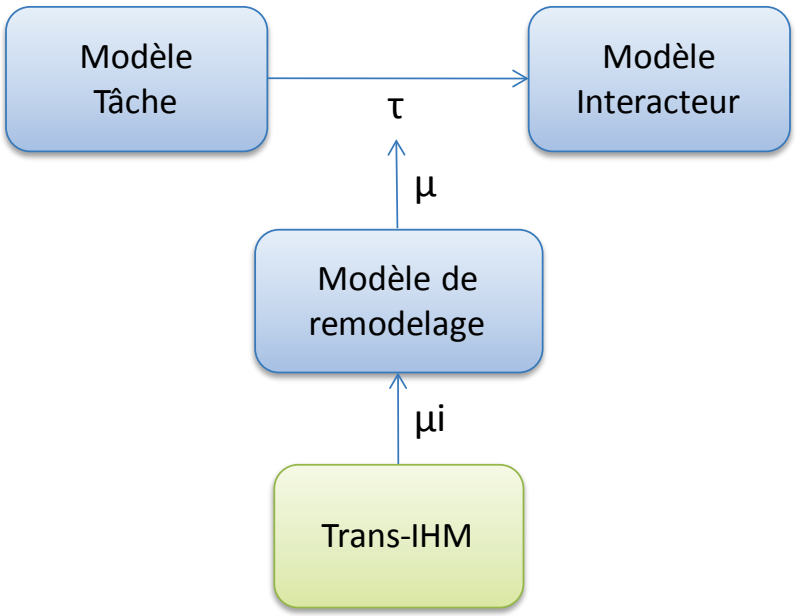

Figure 12. Trans-IHM permettant de modifier, contrôler le remodelage. La transformation représente ici ce remodelage.

- Redistribution : *extra-IHM* donnant accès aux relations entre tâches et plates-formes. Certaines de ces actions sont vues comme du paramétrage par Coutaz (exemple figure 13).

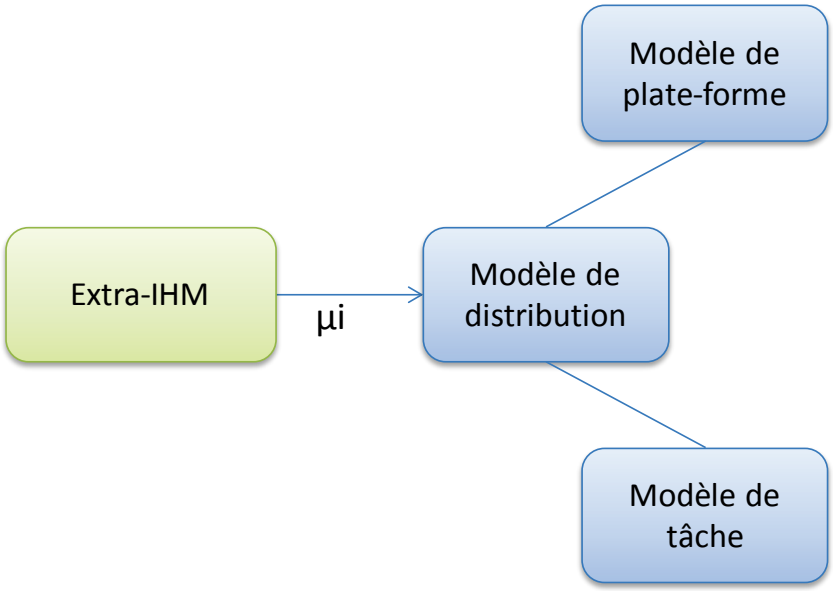

Figure 13. Modèle conceptuel des *extra-IHM* pour la redistribution des IHM à modèles.

Dans la taxonomie de Coutaz, nous notons trois autres propriétés mises en avant qui peuvent se décrire avec des éléments de notre mega-IHM :

- Découverte : C'est le cas de *meta-IHM* qui indiquent ce qu'il est possible de faire avec l'application, en donnant par exemple des règles de construction. C'est aussi le cas *d'extra-IHM* qui présentent des applications particulières.

- Paramétrage : *extra-IHM* donnant accès aux éléments de la modélisation. Un remodelage peut être issu du paramétrage. Le paramétrage a aussi un effet sur le modèle des concepts de l'application. A grain fin, le paramétrage dans une *trans-IHM* peut être utilisé pour choisir une couleur de fond ou un style

- Assemblage : C'est une forme *d'extra-IHM* qui assemble des fonctionnalités de base : une *extra-IHM* permettant de construire un arbre des tâches avec des tâches stéréotypées. On utilisera ici la compostion de (meta)modèles comme assemblage de fonctionnalités.

Toujours dans l'article de Coutaz, on parle de tissage des IHM de configuration (ici tissage de *mega-IHM*) dans l'application. Le tissage est défini par l'intégration à l'IHM des éléments de la *mega-IHM*. En termes d'arbre des tâches cela reviendrait à représenter dans le même modèle les tâches de l'application et les tâches de *l'extra-IHM* : comme par exemple la tâche « redistribuer sur autre plate-forme ». L'illustration du **chapitre VI,** montrant un bouton pour redistribuer des parties de l'IHM sur PDA, est un exemple d'*extra-IHM* tissée.

2.2 Exemple

Sans chercher d'applications exotiques, prenons Internet Explorer où l'on peut cacher les barres de menus. Ceci revient à dissimuler les tâches « *sélection d'une fonctionnalité* » dans le menu. La barre de menu outils donne accès aux tâches *d'extra-IHM* de paramétrage. Celles représentées sur le cadre bleu font référence à des paramètres ayant une incidence sur l'IHM.

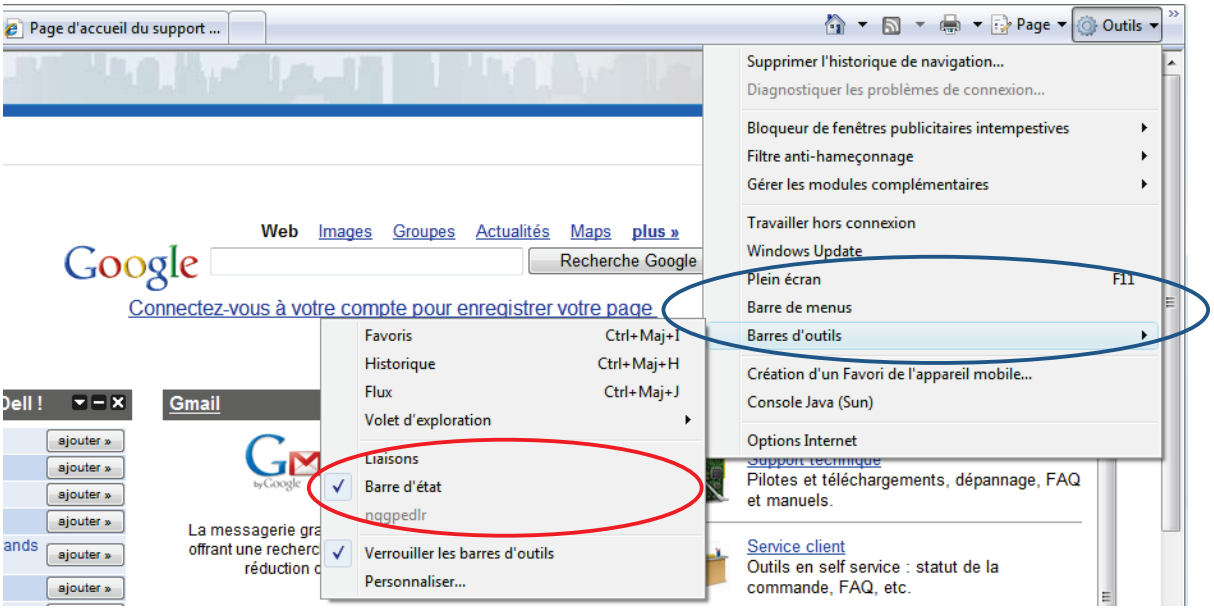

Figure 14. Extra-IHM spécifique aux tâches de « menu ».

D'après la terminologie, l'IHM de la figure 14 est une *extra-IHM* tissée qui *paramètre* son environnement. Dans notre cas, ceci revient à supprimer le lien entre la tâche « *sélectionner item du menu* » et la plate-forme. Ce n'est pas un paramétrage mais une nouvelle distribution où justement cette tâche n'est pas distribuée !

Le contrôle utilisateur se fait par le biais de la *mega-IHM* qui semble couvrir au moins en termes de services génériques les travaux déjà réalisés dans le domaine des IHM. Après avoir fait le lien entre les services de contrôle utilisateur en IHM et la *mega-IHM* nous voyons comment donner les moyens de contrôler l'adaptation par l'utilisateur final.

2.3 Adaptation au contexte contrôlée par l'utilisateur

Il est nécessaire que l'utilisateur puisse parfois contrôler ce qui se passe lors de l'adaptation de son IHM sur changement de contexte. Il doit avoir les moyens de « consulter » ce qui se passe lors d'une adaptation (semi)automatique initiée par le système. Comme nous l'avons vu dans la partie 2.1, il faut fournir des *extra* et *trans-IHM* à l'utilisateur pour qu'il continue à garder le contrôle sur la plasticité ou au moins comprenne ce qui se passe lors de l'adaptation.

La redistribution n'est pas automatisée mais il y a des cas où c'est nécessaire pour préserver l'interaction. Par exemple dans le cadre d'une application critique (ou d'une tâche critique) il est intéressant de migrer automatiquement cette application d'une machine vers une autre si la batterie de la première tombe en panne. Mais une fois de plus, on peut demander à l'utilisateur vers quelle machine migrer, qu'est-ce qu'il désire réellement migrer, etc.

L'utilisateur, par le biais d'une *extra-IHM,* va contrôler une partie du remodelage. Par exemple, en changeant explicitement un bouton par un autre sur le modèle. Dans notre cas c'est la *trans-IHM* qui agit sur le remodelage en paramétrant la transformation Tâches vers Interacteur (figure 12).

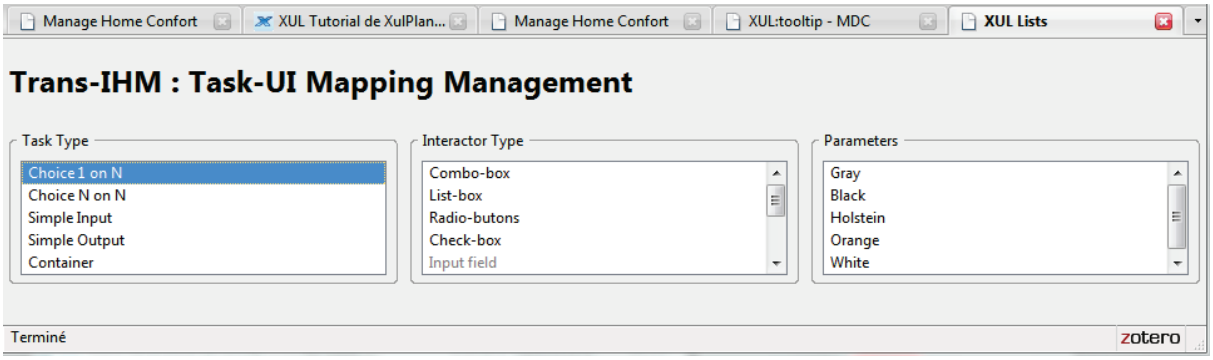

Figure 15. *Trans-IHM* spécifique à un DSTL (types de tâches).

Pour réaliser à la volée une *trans-IHM* spécifique à l'application que nous considérons, il faut prendre les éléments (metamodèles ou modèles) qui pilotent le modèle source de la *transformation*. Par exemple, la *trans-IHM* présentée dans le **chapitre VI** (figure 15), propose de sélectionner (colonne de gauche) un type de tâche particulier : celui-ci est spécifique à notre application.

Ensuite il faut combiner ces éléments-là aux éléments cibles (éléments du metamodèle cible). Sur la figure 15, colonne centrale, les éléments cibles sont donnés par un ensemble d'interacteurs provenant du metamodèle d'interacteur. L'ensemble des transformations possibles (mises en correspondances entre éléments source et cible) est symbolisé dans la *trans-IHM* par la sélection d'une valeur à la fois dans la colonne de gauche et la colonne du centre.

Cette *trans-IHM* étant elle-même une IHM, elle est définie par un modèle d'interacteur (réflexivités des éléments de la *mega-IHM*).La figure 16 décrit la vision générale de génération d'une *trans-IHM* pour le remodelage spécifique à une application. La première transformation est une transformation qui permet de générer le modèle d'interacteur spécifique décrivant la *trans-IHM*. Cette transformation prend en entrée un modèle de tâche (spécifique à l'application) et le metamodèle d'interacteur.

Une fois cette *trans-IHM* obtenue, elle va agir sur un modèle de transformation spécifique à lui-même issu des mêmes modèles mais par une autre transformation dite d'ordre supérieure. Cette dernière n'est pas représentée sur le schéma de la figure 16 pour des raisons de lisibilité. Ce modèle de transformation spécifique représente la relation τ2 entre un modèle d'interacteur et le modèle de tâche de l'application. Il est, par conséquent, également un modèle de remodelage.

Enfin par l'intermédiaire de la *trans-IHM*, l'utilisateur va pouvoir agir sur le modèle de transformation spécifique pour contrôler le remodelage. L'exécution de ce modèle de transformation (par exemple avec **ATL**) va paramétrer la génération d'une IHM pour l'application. Le changement opéré par la *trans-IHM* sur cette transformation est par définition du remodelage.

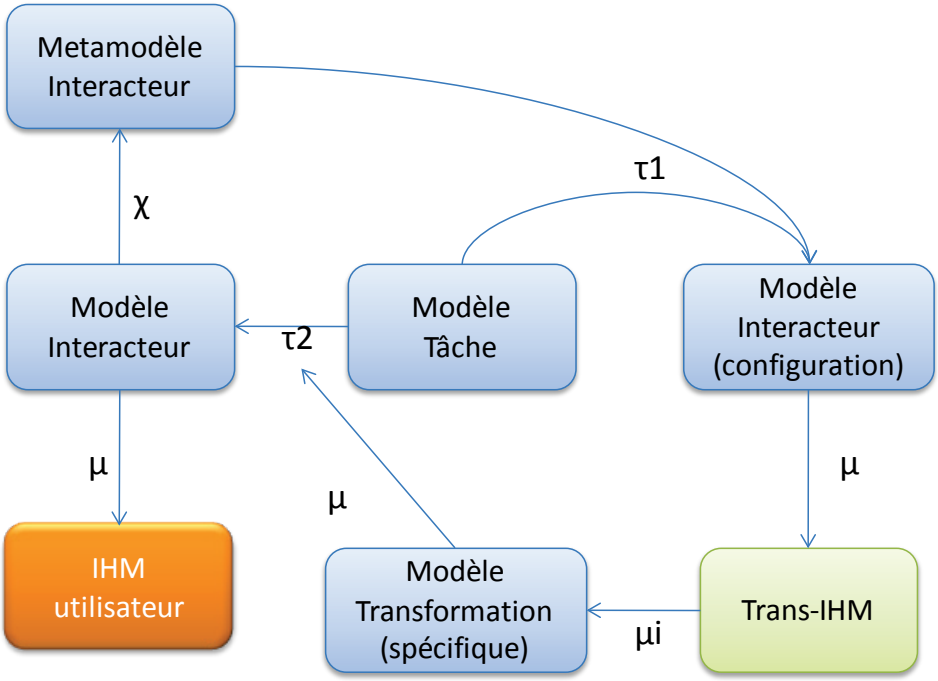

Figure 16. Génération d'IHM de transformation spécifique à une application considérée. Contrôle du remodelage.

Un « concepteur ergonome » peut également utiliser ce type de génération afin d'élire ses critères favoris dans le cadre de prototypages rapides par exemple. Il peut aussi paramétrer lui-même les règles de transformation et les « typer » en fonction des critères qu'il estime devoir favoriser lors de son paramétrage.

3 Synthèse

Nous avons vu dans ce chapitre comment appliquer les concepts de l'IDM à l'IHM pour donner lieu à la malléabilité. Une fois cette malléabilité mise en place, elle nous sert de cadre de travail pour rendre plastique la conception et l'exécution d'IHM.

Dans le **chapitre II** nous avons vu que l'état de l'art en IHM ne couvrait pas, ou peu, les points suivants:

- l'adaptation dynamique au contexte

- la souplesse face aux différents langages utilisés en conception (metamodèles et transformations)
- La prise en compte de « valeur » d'ergonomie pour la plasticité

La malléabilité apporte une réponse à ces lacunes. Dans la partie 1 de ce chapitre nous avons vu comment nous pouvons répondre aux points 1 et 3. L'adaptation dynamique (sections 1.3.2 et 1.4) se fait par l'intermédiaire d'un *intergiciel*. Ce dernier est équipé d'une partie décisionnelle, à l'instar de l'informatique autonomique, visant à automatiser l'adaptation de manière « intelligente ». Les valeurs pour la plasticité sont portées par les transformations et les mises en correspondance à l'aide du metamodèle que nous donnons. Cette forme d'annotation permet de prendre en compte l'ergonomie lors de la conception mais aussi pour toute re-conception ultérieure.

Dans la partie 2 de ce chapitre nous avons montré comment les principes du **chapitre VI** permettent d'assurer une certaine souplesse face aux metamodèles, modèles et transformations en les rendant accessibles par les utilisateurs ou les ingénieurs.

.

Chapitre VIII Le démonstrateur MARA

Dans cette thèse, nous avons proposé de nouveaux concepts. Notre démonstrateur **MARA** pour *Model At Run-time Architecture* est basé sur ces concepts. Ce démonstrateur a été élaboré avec l'aide précieuse de Xavier Alvaro. **MARA** illustre la plasticité des IHM (**chapitre VII**) en utilisant l'IDM (**chapitre III**) ; il est basé sur les langages bien établis dans la communauté IHM (**chapitre II**). En outre cette plasticité utilisant la malléabilité fait appel à différentes IHM de configuration pour tous les éléments de modélisation et de transformation (**chapitre VI**).

L'objectif de ce chapitre est double : d'une part de montrer les aspects techniques de notre démonstrateur et d'autre part présenter de nos contributions en terme de formalisation IDM des savoirs et savoir-faire en IHM.

La version que nous avons développée répond à la plasticité des IHM par transformation de modèles à l'exécution tout comme à la conception. La première partie est consacrée à l'implémentation de l'infrastructure d'un intergiciel pour l'adaptation. La deuxième partie considère les transformations de modèles ainsi que les annotations pour la plasticité. Enfin dans la troisième partie, on décrit la mise en œuvre de certains éléments de la *mega-IHM*.

1 Infrastructure pour l'adaptation

1.1 Architecture CAMELEON-RT

L'infrastructure que nous avons développée reprend les grandes lignes du modèle conceptuel de l'informatique autonome. L'architecture de notre implémentation se base sur celle de CAMELEON RT [Balme, et al., 2004] présentée dans la figure 1.

Cette architecture propose notamment une partie pour les capteurs qui récoltent des données sur le système interactif. Ces données captées correspondent au contexte d'utilisation (plate-forme, environnement, utilisateur) mais aussi aux éléments du système interactif (*interactive system observer*) lui-même, par exemple ce que l'utilisateur a fait précédemment. A un instant T, le modèle du système interactif en contexte est donné par le synthétiseur de situation (*situation synthetizer*).

La première étape consiste à calculer la meilleure adaptation (*Evolution Engine*), ce qui correspond à la phase de *planification* du modèle autonomique. La seconde phase consiste à réaliser l'adaptation (*Adaptation Producer*) : on peut facilement en déduire que c'est l'équivalent de la partie *Exécuteur* de la figure 1.

Dans notre cas, les éléments captés sont conservés dans un *megamodèle* qui représente le système informatique en contexte. L'adaptation est alors réalisée par transformation (*UI Model Transformers* figure 2) de ce modèle *d'écosystème* (tout ou une partie seulement).

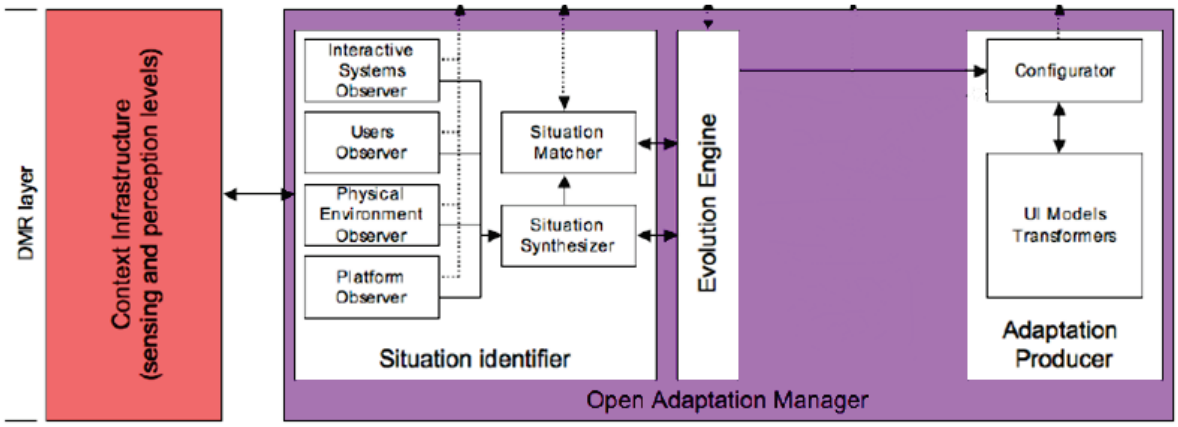

Figure 1. Architecture conceptuelle proposée par Balme.

1.2 Architecture MARA

Notre architecture reprend les principaux composants de la figure 1. Nous avons unifié la fonctionnalité du « *situation identifier* » par un *megamodèle* pour les IHM plastiques que nous avons réalisées. En effet ce *megamodèle* nous donne l'état courant, les acteurs, les machines mises en œuvre lors de l'interaction.

Dans un premier temps, nous verrons le cœur de l'implémentation de **MARA** et les différents composants de cette implémentation. Dans un second temps, nous détaillerons l'interface de **MARA** avec internet. Cette interface permettra de générer des IHM web à partir de **MARA**.

1.2.1 MARA : architecture et composants

Nous conservons l'ensemble des « *observers* » qui permettent de mettre à jour les éléments du *megamodèle*. Ces *observers* sont situés dans le gestionnaire du *megamodèle (ecosystem model manager).* Seul le contrôle des plates-formes (arrivée, départ, capacités) est externalisé dans le composant « *platform controller* ».

« *L'adaptation producer* » est uniquement un transformateur de modèle qui travaille alors sur le *megamodèle* (modèle du système interactif en contexte plus *metamodèles*). Nous avons séparé en deux composants distincts la gestion du modèle et celle du metamodèle (*EcosystemModel et metamodel managers*)

Le composant « *storage space* » contient la base des règles de transformations applicables (*transformation rule manager*). Le moteur d'évolution (*evolution engine*) se base sur les annotations des règles de *transformations* pour appliquer l'ensemble (module) de règles les plus adaptées.

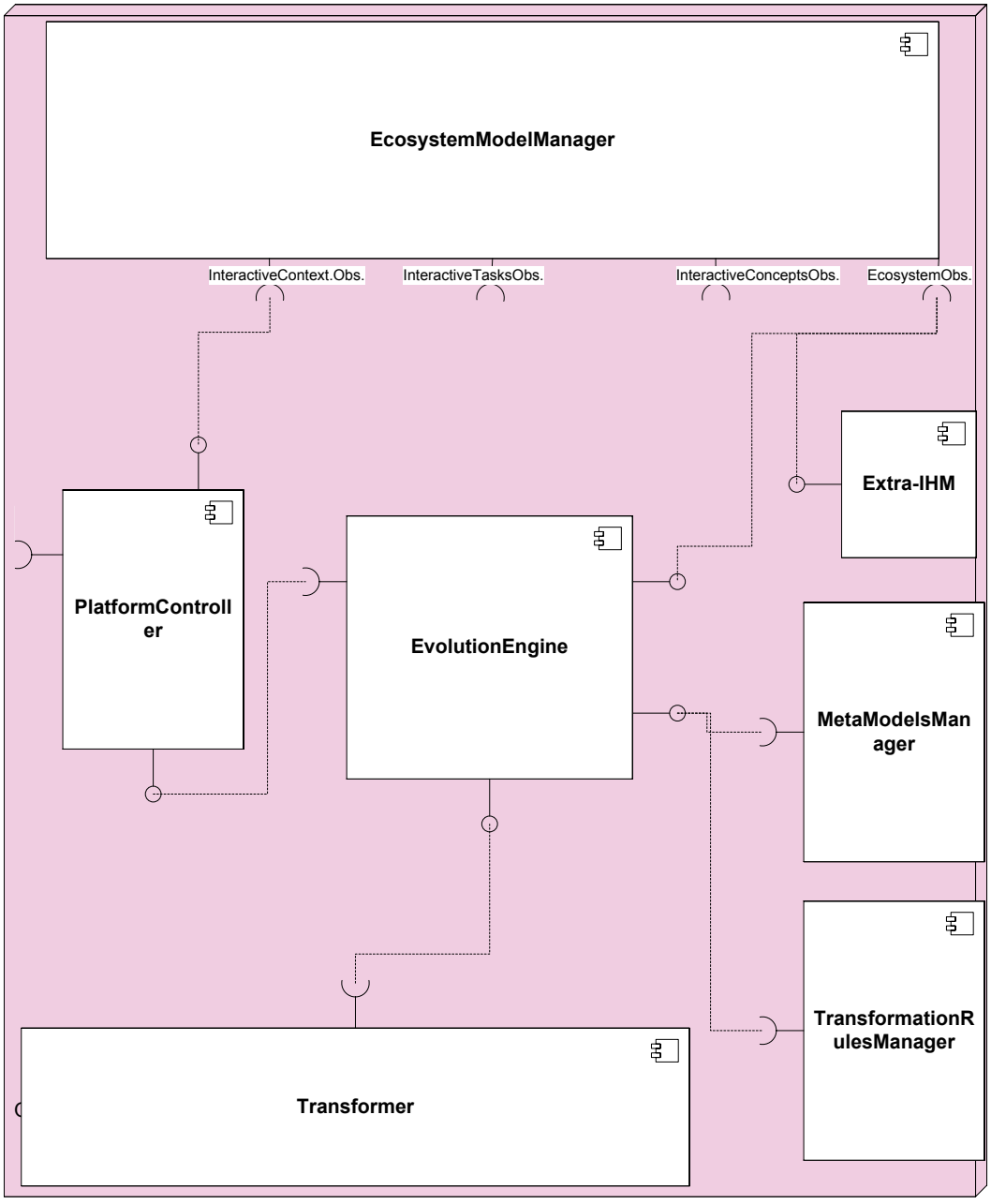

Figure 2. Architecture de MARA.

1.2.2 Implémentation concrète

L'implémentation est basée sur les services OSGI [Osgi] tournant sur la plate-forme **OSCAR**. La spécification OSGI définit une plate-forme d'exécution d'applications Java localisées au sein de la

même machine virtuelle Java. Cette plate-forme commune permet à une diversité de services logiciels d'être chargés et exécutés. OSGI permet également l'administration du cycle de vie d'un logiciel existant sur cette plate-forme, depuis n'importe où dans le réseau local ou distant. La plate-forme déploie dynamiquement des applications sans arrêter la plate-forme OSGI. Les services peuvent par la suite être démarrés, mis à jour, arrêtés ou désinstallés sans que la machine virtuelle java ou **JVM** ait besoin de redémarrer. Un grand avantage de cette spécification **OSGI** est son modèle puissant acceptant la cohabitation de différentes applications au sein d'une seule **JVM**. Cela permet de réduire les quantités de ressources utilisées par application, augmentant ainsi les performances.

Pour **MARA** dans l'état actuel, nous n'avons qu'une seule version de nos « composants **OSGI** » appelés « *bundle* ». Le redéploiement de composants nous a servi ici surtout lors des phases d'implémentation et de mise au point de **MARA**.

Nous envisageons à terme d'employer plusieurs machines de transformation de modèle. On choisira alors la plus appropriée en fonction du modèle, de sa forme : préférer XSLT lorsque les modèles sont en XML, etc.

1.2.3 MARA interface Web

L'objectif de MARA est de générer et d'adapter dynamiquement des IHM par transformation de modèles. Pour cela, nous avons aussi besoin d'interfaces entre **MARA** et les plates-formes qui vont être le lieu de l'interaction. Nous n'avons implémenté qu'une interface web pour démontrer la plasticité par malléabilité des IHM dirigées par les modèles.

Nous avons réalisé une interface « serveur web » qui permet de se connecter au cœur de **MARA** (figure 3). Le navigateur web « *Web Browser* » est ici représenté par un composant qui n'est pas un bundle OSGI (par exemple Mozzila Firefox). Le Bundle « *HTTPServer* » est donné par défaut sous **Oscar**. Le bundle « *WebInterface* » est un composant générique qui permet la communication entre un navigateur/serveur web et le bundle « *platformController* » (figure2).

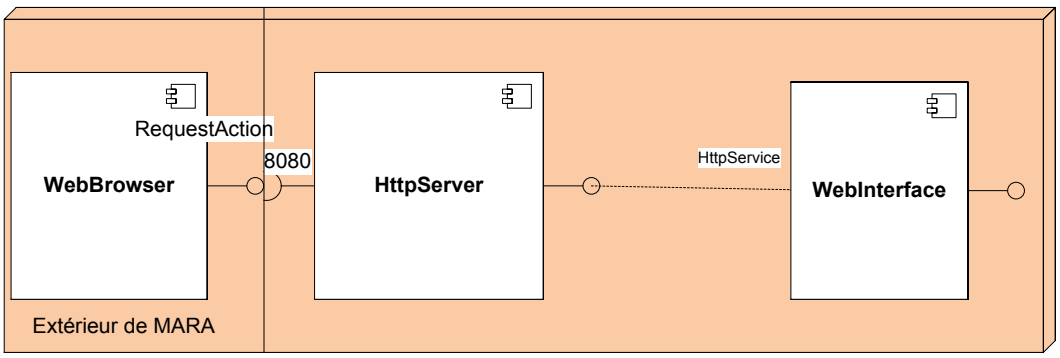

Figure 3. Client-serveur web pour MARA.

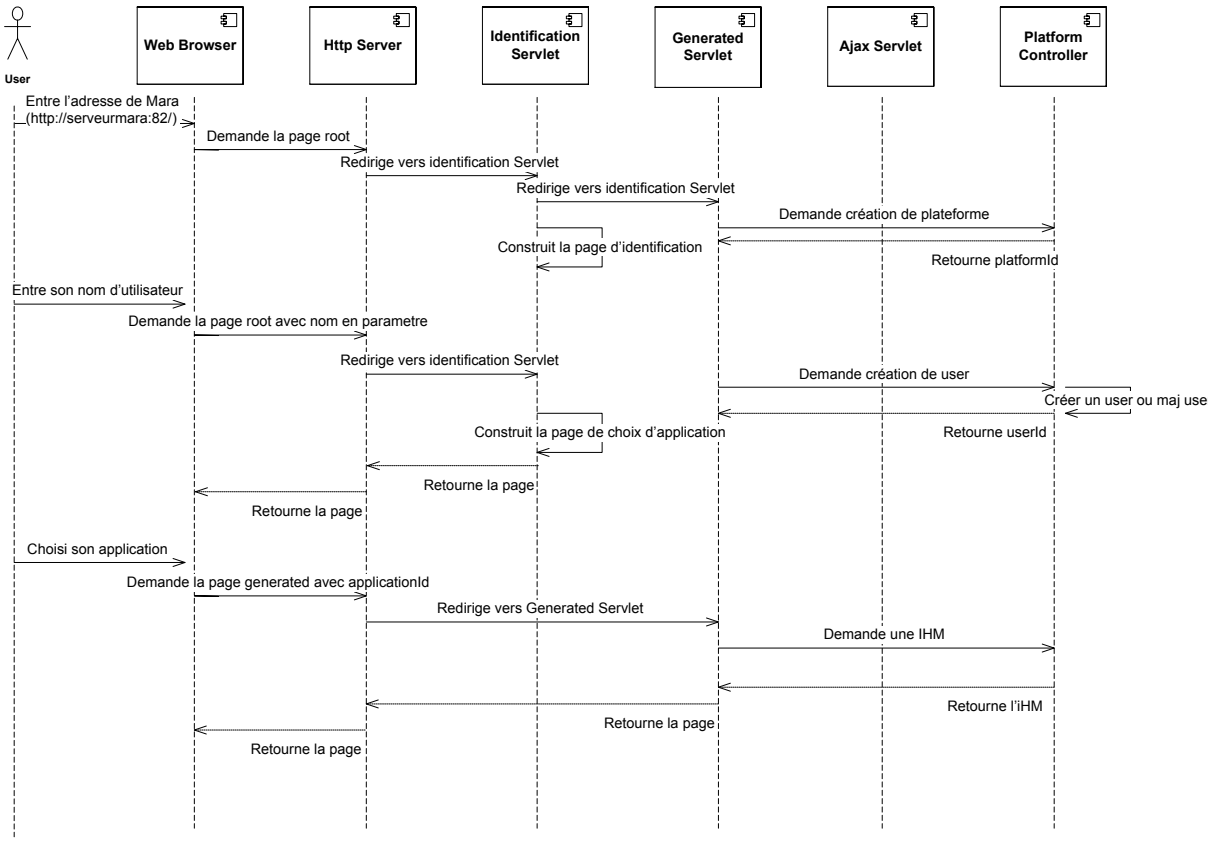

Figure 4. Diagramme de séquences d'une connexion à **MARA** du point de vue du serveur (modèle conceptuel de séquences).

Lorsqu'on se connecte au serveur http pour **MARA** (figure 4), la partie « *WebInterface* » notifie le « *PlatformController* » de l'arrivée d'une nouvelle plate-forme. Ensuite l'utilisateur passe par le mécanisme d'identification (login, mot de passe, figure 5). Dans la figure 4, l'*HTTPServer* est composé de différentes « *Servlet* ». Enfin il choisit son application en fonction de celle réalisable (figure 6) par **MARA**. Une IHM est calculée par transformation à l'aide des bundles de la figure 2.

Figure 5. IHM d'identification d'utilisateur sur MARA sur serveur web.

Figure 6. Sélection d'application dans MARA ; ici deux applications Confort domestique ou Réservation de train.

Enfin lorsque l'application a été choisie et la transformation lancée, l'IHM résultante est affichée sur le navigateur web connecté à **MARA** (figure 7).

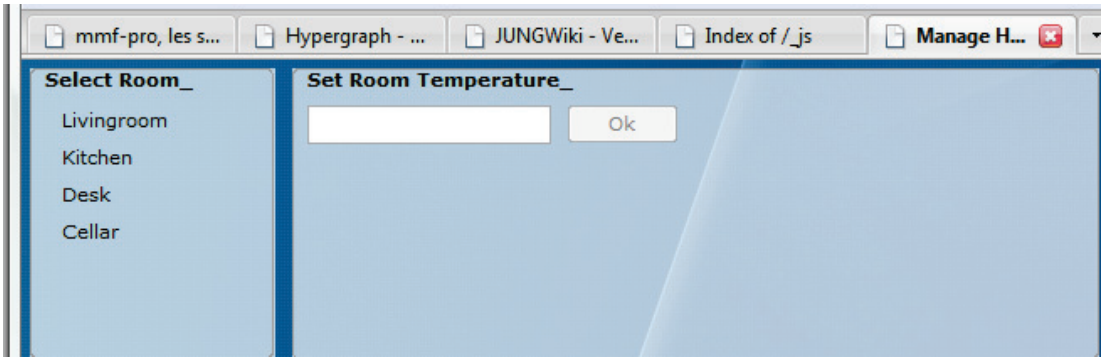

Figure 7. IHM générée sur le navigateur connecté à **MARA**.

2 Transformations de modèles et metamodèles

2.1 Technologies IDM dans MARA

Les transformations de modèle dans **MARA** sont écrites dans le langage **ATL**. Les différentes transformations sont organisées sous forme de modules. Chacun des modules correspond à un ensemble de règles de transformations (cohérentes) ; afin de pouvoir générer le modèle ou le code de l'application dans nos chaînes de transformation. Nous utilisons deux types de transformations **ATL** : modèle vers modèle et modèle vers texte. La première (ATL rule) nous sert à réaliser différentes transformations de modèles vers un modèle pivot. La seconde (ATL query) nous sert à générer du code pour une plate-forme cible.

Le composant *Transformer* de notre architecture (Figure 2) est réalisé à partir d'une machine **ATL** disponible en **JAVA** (élaborée par Jon Oldevik). Nous avons adapté cette première version de la machine **ATL** pour en faire un bundle **OSGI**.

L'exécution d'une règle de transformation se fait si elle a été choisie en fonction de son annotation. Les (*meta)modèles* sont réalisés en **EMF**, à l'aide de l'outil **Topcased** [Topcased].

Topcased est un éditeur de (meta)modèle, développé par Airbus, notamment au moyen de langages comme UML, AADL etc. Il reprend les principaux éléments d'Eclipse pour la modélisation, à savoir **EMF**. Nous nous servons de **Topcased** pour l'aspect « édition graphique » afin de réaliser des metamodèles **EMF**.

EMF ou **Eclipse Modelling Framework** est un cadre de travail permettant de réaliser des metamodèles et modèles : nous l'avons décrit dans le **chapitre III**. Ici nous nous servons d'**EMF** comme *langage* de *metametamodélisation*. Nos metamodèles, tout comme nos modèles, sont exprimés en **EMF**. On se sert à la fois de la forme XMI de **EMF** pour réaliser nos transformations en **ATL** et de l'API **JAVA** pour mettre à jour des éléments provenant du contexte.

Certaines transformations « statiques » pour générer du code sont réalisées avec **Acceleo** [Acceleo]. Les templates **Acceleo** permettent d'obtenir rapidement du code à partir d'un modèle **EMF**. D'une grande souplesse d'utilisation il peut générer quelques prototypes d'IHM rapidement.

En résumé, nous utiliserons les transformations ATL pour toute la partie dynamique de génération d'IHM ; pour assurer la plasticité des IHM par malléabilité. Les transformations en JAVA *via* **EMF** serviront surtout dans le cadre d'ajout/suppression ponctuel d'éléments. C'est le cas notamment lors de l'arrivée d'une plate-forme ou de la suppression d'un élément dans une extra-IHM. Enfin **Acceleo** nous sert à écrire rapidement des règles de génération de prototype d'IHM (au niveau code) qui n'ont pas besoin d'être dynamiques.

2.2 Annotation & metamodèle pour la transformation

Nous avons des annotations « type de tâche » pour les classes tâches qui nous permettent de cibler un ensemble d'interacteurs correspondants. Ces annotations (types de tâches) sont une manière de réaliser une mise en correspondance syntaxique entre modèle de tâche et modèle d'interacteur.

2.2.1 Annotation du metamodèle de tâche

Dans notre cas, les types de tâches ne sont pas décrits par une énumération dans le metamodèle. La nature d'une annotation « type de tâche » est donnée par le concepteur lors de l'élaboration du modèle de l'application.

On peut cependant avoir quelques types de tâches interactives génériques comme le choix d'un élément parmi n, l'entrée de valeur, l'affichage de valeur. Ici valeur a le sens des types primitifs en java : chaîne de caractères, caractères, entiers, flottants, etc.

Les types de tâches sont donc spécifiques à chacune des applications et des transformations qui permettent leur génération. Par exemple, on peut cibler une technique d'interaction spécifique : un menu circulaire pour faire un choix dans une liste d'éléments.

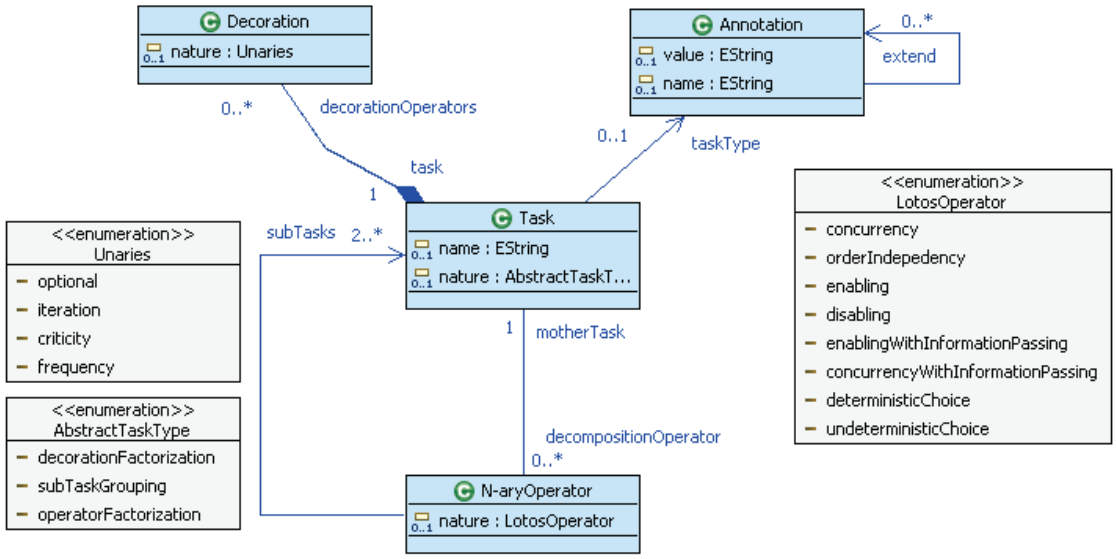

Figure 8. Metamodèle de tâches. Une tâche est annotée spécifiquement pour une application (classe Annotation).

```
rule TaskChoiceToGroupBox {

      from tsk : MMEcosystem!Task (tsk.taskType.name='Choice 1/n'),
      …}
```

Dans le listing ci-dessus nous présentons un extrait de règle ATL dont on ne considère que le « patron source ». Nous mettons en évidence le filtrage sur l'ensemble des tâches du type 'Choice 1/n' qui a été précisé à la co-conception de l'arbre des tâches et de la transformation. Une action particulière sera réalisée sur cette tâche : par exemple générer une IHM abstraite ou bien directement cibler un ensemble d'interacteurs.

Ce type d'annotations permet de garder une malléabilité pour la transformation par rapport au metamodèle de tâche tel qu'il est défini dans la figure 8.

2.2.2 Metamodèle de tâche - concept en contexte

Le metamodèle de tâche de la figure 4 doit être relié au modèle des concepts du domaine (au moins ceux qui sont utilisés lors de l'interaction). Une application interactive ne peut être uniquement spécifiée par un modèle de tâches : il faut alors représenter les données qui sont manipulées.

Le modèle de concept de domaine dans **MARA** est constitué de deux niveaux de modélisation : les concepts (Classe en **UML**) et les instances de concepts (Objets en **UML**). Ce modèle permet alors de décrire de manière générale les classes qui sont manipulées par les tâches, et lors de l'exécution de l'application, de représenter les objets manipulés lors de l'interaction.

Le modèle de contexte est ici limité à son plus simple élément : la plate-forme. Dans **MARA** nous considérons aussi l'utilisateur, celui-ci est alors identifié par l'intermédiaire de la plate-forme. Mais le modèle de l'utilisateur n'a, dans l'état actuel du développement, aucune influence sur les transformations de modèle à modèle.

La classe *platform,* qui peut modéliser un type de navigateur web, possède un attribut « active ». Cet attribut est un booléen. Il permet de définir s'il faut générer (*active = true*) ou non (*active = false*) une IHM pour cette plate-forme, c'est-à-dire si celle-ci se connecte au cœur de **MARA** pour réaliser une transformation.

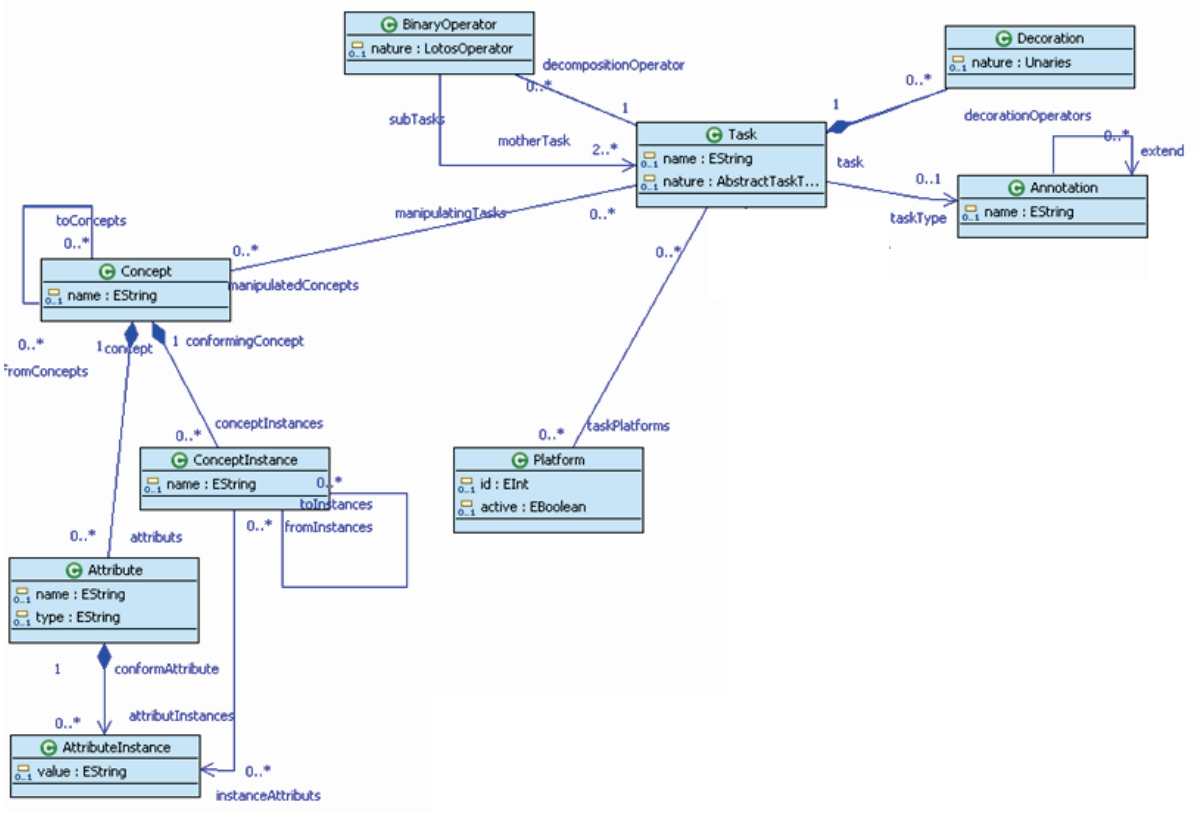

Figure 9. Metamodèle tâche-concept-plate-forme.

Les mises en correspondance explicites entre les tâches, les concepts du domaine et les plates-formes ont un impact sur les transformations. Par exemple, on ne fait rien si la relation entre une tâche et la plate-forme (active) n'existe pas. Nous nous référons au principe de distribution de l'IHM que nous avons énoncé dans le chapitre précédent.

```
rule TaskChoiceToGroupBox {

      from tsk : MMEcosystem!Task (tsk.taskType.name='Choice 1/n'
                                   and tsk.manipulatedConcepts->size()>=1
--La tâche manipule au moins 1 concept
                                   and tsk.manipulatedConcepts->collect(e |
e.conceptInstances)->flatten()->size() <= 6
--La tâche manipule moins de 6 instances de concepts
                                   and tsk.taskPlatforms->select( e |
e.active=true)->size() >= 1
--La plate-forme cible est « active » : elle demande une IHM
}
```

Dans le listing **ATL** ci-dessus, nous reprenons l'exemple précédent : une tâche de type Choix d'un élément dans une liste. La première partie de règle permet de valider le fait que la tâche ciblée manipule au moins un concept. Si ce n'est pas le cas, c'est une erreur de conception du modèle et la transformation n'est pas exécutée. Ce test navigue dans la relation entre une tâche et un concept par le rôle *manipulatedConcepts* (figure 9).

Ensuite on vérifie que la tâche ne manipule pas plus de six instances de concepts : c'est-à-dire que pour l'application considérée nous avons à réaliser un choix parmi une à six possibilités. On remonte alors la relation vers les concepts comme pour la règle précédente. A partir des concepts on récupère l'ensemble des instances de concepts en remontant la relation entre les concepts et leur instances par un collect (collect (e | e.conceptInstances). Pour finir, on vérifie que la liste contient entre zéro et six éléments.

La validation selon moins de six instances de concept est ici un filtre ergonomique basé sur les remarques de [Vanderdonckt, 1998]. Afin d'assurer la complétude de la transformation, une deuxième règle est réalisée pour les tâches de choix 1 parmi N manipulant plus de six instances de concepts.

Enfin la dernière règle vérifie que la tâche cible est bien liée à une plate-forme et que cette dernière demande une transformation. Pour ce faire, on remonte la relation entre la tâche et la plate-forme par le rôle *taskPlatForms*.

2.2.3 Metamodèle d'interacteur

Les transformations décrites dans les parties précédentes (sous section 2.2.1 et 2.2.2) prennent comme patron cible des éléments d'un metamodèle d'interacteur. Pour ce faire, nous avons défini

un metamodèle d'interacteur générique : il est basé sur les interacteurs qu'on retrouve couramment dans les boites à outils tels que SWING ou XUL. Il permet entre autres de décrire un ensemble de conteneurs (*Container*) et d'interacteurs pour le contenu (*Containement*). Les différents héritages ne sont pas représentés dans la figure 10. Pour illustration, la classe Windows est un conteneur. La classe *RadioGroup* représente le groupement d'interacteurs de type radio bouton et contient exclusivement des radio boutons.

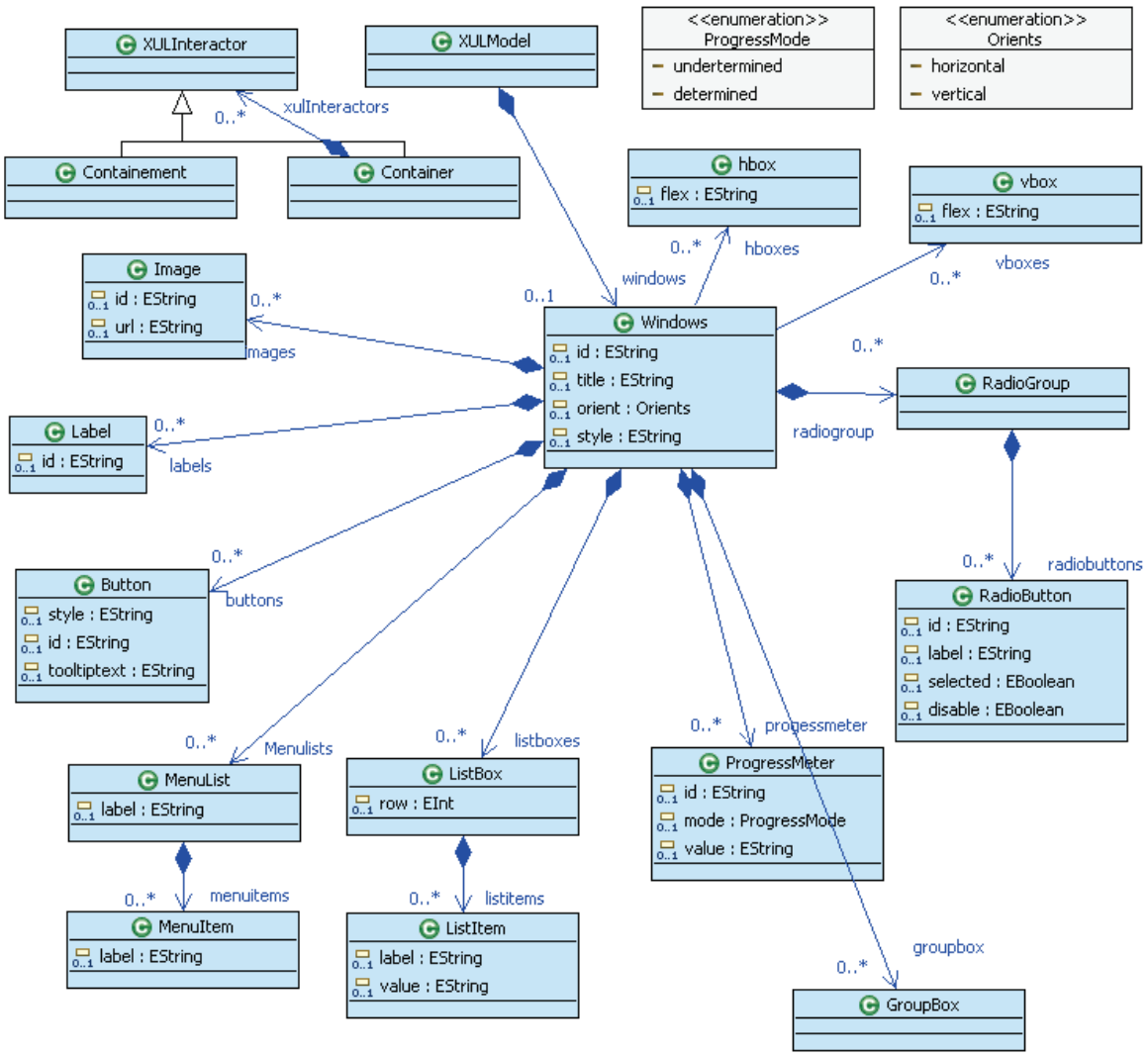

Figure 10. Metamodèle d'interacteur.

La transformation de modèle utilisée ici en exemple (pour les tâches de type choix d'un élément dans une liste 'Choice 1/n') cible les interacteurs définis par les classes *RadioGroup* et *RadioButton*. Nous ne représentons ici que la partie patron cible (après le to en **ATL**)

```
to gp_box : XULMetaModel!GroupBox (id <- 'group'+ tsk.name,
                                   flex <- 1,
                                   xulInteractors <- Sequence {capt,vbx,}),
```

```
   capt : XULMetaModel!Caption(label <- tsk.name),

      vbx : XULMetaModel!RadioGroup(id <- 'radio_'+ tsk.name)
do {
            for (e in tsk.manipulatedConcepts) {
                  for (z in e.conceptInstances) {
                        vbx.radiobuttons <- thisModule.radioBuild(z,tsk);
                  }
            }
      }
```

L'extrait de transformation ci-dessus est composé de deux blocs : déclaratif et impératif. Le bloc repéré après le mot clé « *do* » autorise l'emploi d'une syntaxe impérative si la condition (après le « *to* ») est respectée. Dans notre cas, on génère un radio boutons pour chacune des instances de concepts liée à la tâche considérée par la présente règle.

Pour améliorer la modélisation dans **MARA** il nous faut introduire des metamodèles spécifiques aux techniques d'interaction. Associer un metamodèle d'interacteur spécifique à ces interactions : ce metamodèle de technique d'interaction est ensuite tissé avec le modèle de tâche et permet, à terme, de remplacer les annotations.

2.3 Transformation en « JAVA »

Pour tout ce qui est arrivé/disparition de plate-forme nous n'utilisons pas **ATL** pour mettre à jour le modèle de l'application (ou son état) mais un contrôle en **JAVA**. Ce contrôle est simplifié par la génération des accesseurs aux classes **JAVA** du (mega)modèle par **EMF**.

Par exemple si un navigateur web se connecte à **MARA** par l'intermédiaire du serveur web, le modèle de plate-forme est mis à jour avec cette nouvelle instance de plate-forme. Au moment où il est mis à jour, son attribut « *active* » est affecté à la valeur *true*. Plus tard il sera mis à *false*.

```
if (htmlHeaderType.contains("Windows CE") || htmlHeaderType.contains("Opera
Mini"))
type = "SMALLSCREEN_HTML";
else
type = "BIGSCREEN_HTML";
```

Le listing ci-dessus est situé dans le paquet de bundle « web » (voir figure 3). Il présente l'identification des différentes plates-formes web à partir de l'en-tête http. Si cet en-tête contient Windows CE ou Opéra, le type de la plate-forme est « petit écran » (*SmallScreen*).

Ensuite on fait un appel explicite au bundle « *PlatformController* » qui est en charge de mettre à jour les différents modèles de plate-forme (voir appel ci-dessous). On conserve ainsi l'indépendance entre le modèle, le contrôle des plates-formes et les technologies cibles pour lesquelles **MARA** doit fournir une IHM.

```
platformController.AddPlatformType(HtmlCode.SMALLSCREEN_HTML_PLATFORMTYPE_NAME,HtmlCode.SMALLSCREEN_HTML_METAMODEL_NAME);
```

Enfin, c'est au Bundle « *EcosystemModelManager* » qui gère notre megamodèle, de mettre à jour les instances de plate-forme au regard de ce qui lui est passé en paramètres par le « *PlatformController* ». On crée ainsi une nouvelle plate-forme en faisant appel au code généré par **EMF** auquel nous avons ajouté la gestion d'événements.

```
public int AddPlatform(String platformTypeName){
return ecosystemModelObserver.CreatePlatForm(0,platformTypeName).getId();
}
```

Au moment de la connexion entre une plate-forme et une application dans **MARA** : on réalise l'ensemble des mises en correspondance entre les tâches représentant l'interaction dans l'application et cette plate-forme.

2.4 Gestion et génération d'applications interactives dans MARA

MARA permet de gérer plusieurs modèles d'application. Si un composant extérieur (navigateur web par exemple) vient à demander l'exécution d'une application, alors une chaine de transformations spécifiques au contexte est lancée. On passe en paramètre l'identifiant correspondant à l'arbre des tâches modélisant l'application ; les autres éléments, comme les concepts, sont retrouvés par association. En effet dans **MARA**, une application est spécifiée par un modèle « central », ici l'arbre des tâches.

Le modèle de contexte est mis à jour par une « *transformation* » JAVA décrite dans la section 2.3.

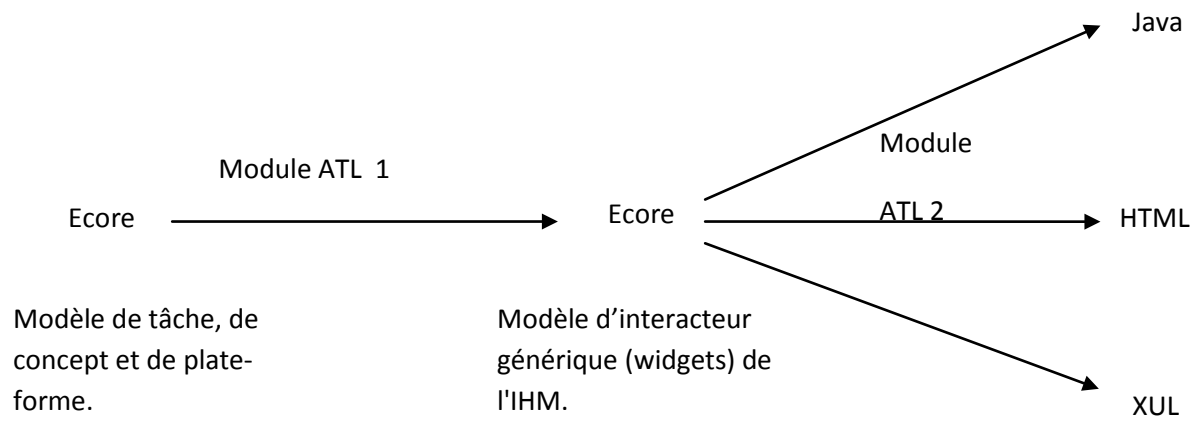

Figure 11. Chaîne de transformation.

Sur ce schéma nous avons simplifié les étapes de transformation de modèles à deux exécutions successives de module de transformations. On exécute en premier un module regroupant l'ensemble des règles de transformations permettant, pour une plateforme donnée, de générer le modèle

d'interacteur. Ce dernier correspond aux relations tâche-concept représentant l'interaction d'une application.

On fait appel pour la transformation numéro deux à une « *query* » **ATL** : en d'autres termes une transformation qui permet de générer du texte ou du code.

Les transformations de modèles sont stockées dans la base « *TransformationRuleManager* ». Cette base de transformations est elle-même un modèle reliant des règles de transformations à des propriétés. Ces propriétés sont ensuite sélectionnées par un moteur JAVA pour être chaînées comme dans la figure 11.

2.5 Sélection (automatique) des transformations

En fonction du contexte d'utilisation les modules de transformations sont sélectionnés. Par exemple, si un navigateur HTML sur PDA se connecte au cœur de MARA, on utilise les transformations spécifiques aux « petites plates-formes ».

Le bundle « *EvolutionEngine* » remplit la fonction de sélection de transformation de modèle. Il est couplé au bundle « *TransformationRuleManager* ». Voici une illustration de ce que contient le modèle de la base de transformation utilisée par l' « *EvolutionEngine* ». Dans l'extrait ci-dessous, nous avons une propriété de « grand écran ». Cette propriété caractérise, entre autres, la règle MMTaskToMMXUL qui permet d'obtenir un modèle d'interacteur à partir d'une tâche.

```
<TransformationPropertie Name="Screen" Value="Big" Id="1"/>
<TransformationRule Name="MMTaskToMMXUL"
      Path="TransformationFiles/BigScreenTskToXULMetaModel.asm"
      InputMetaModel="MMEcosystem" OutputMetaModel="XULMetaModel">
            <TransformationPropertie Id="1"/>
</TransformationRule>
```

Comme nous l'avons vu précédemment, les propriétés de transformation de modèle peuvent être des critères ergonomiques, des contraintes d'implémentation, etc.

Dans le bundle *EvolutionEngine* (écrit en JAVA), on peut sélectionner la transformation à appliquer en fonction de la plate-forme qui s'y connecte. La ligne de JAVA suivante interroge, dans le modèle de plate-forme, celle qui requiert la transformation : elle stocke dans outputModel le format de sortie de la « query » qui doit générer le code (soit XUL, HTML, SWING, etc.).

```
outputModel=platFormType.getPlatformProperties().get(i).getValue();
```

Ensuite on récupère les caractéristiques de cette plate-forme (getName et getValue) qui permettent par la suite d'obtenir les transformations dotées des propriétés adéquates.

```
TransformationPropertie trpr = new TransformationPropertie();
trpr.setName(platFormType.getPlatformProperties().get(i).getName());
trpr.setValue(platFormType.getPlatformProperties().get(i).getValue());
properties.add(trpr);
```

On ajoute aux transformations à exécuter le résultat de la correspondance entre propriétés et transformations.

```
transformationList.add(transformationRulesManager.GetTransformationRule("MM
Ecosystem", "XULMetaModel", properties));
```

```
transformationList.add(transformationRulesManager.GetTransformationRule("XU
LMetaModel", outputModel, null));
```

Dans le cas du listing ci-dessus, nous avons limité notre choix aux transformations pour les metamodèles proposés dans **MARA** (figure 9). Nous n'avons pas réalisé de propriétés ou de mises en correspondance plus complexes qu'un alignement syntaxique. Nous laissons cependant cette possibilité d'extension dans les futures versions de **MARA**.

Enfin on fait appel explicitement à la transformation de modèle en interrogeant la liste des transformations à effectuer : on récupère ainsi tous les chemins d'accès aux metamodèles, modèles et transformations. Pour terminer, le bundle « Transformer » est appelé par l'interface créée pour la machine ATL : ATLTransformationEngine.ExecuteTransformation(…).

```
File inputMMModelFile = new
File("MetaModelFiles//"+transformationList.get(i).getInputMetaModel() +
".ecore");
File outputMMModelFile = new
File("MetaModelFiles//"+transformationList.get(i).getOutputMetaModel() +
".ecore");
File transformationFile = new File(transformationList.get(i).getPath());
result=aTLTransformationEngine.ExecuteTransformation(result,inputMMModelFil
e, outputMMModelFile, transformationFile);
```

2.6 Evénements

Prenons une application réalisée sur **MARA** : la gestion de température à la maison. Lorsqu'un utilisateur se connecte, l'ensemble des tâches de l'application « gestion de température à la maison » est liée à la plate-forme de l'utilisateur par le biais de « *l'EcosystemModelMAnager* ».

Ce bundle est composé d'un ensemble *d'observer/listener* qui lui permet d'être notifié d'une arrivée de plate-forme mais aussi de notifier aux bundles qui lui sont abonnés, des modifications opérées sur le modèle. Le code ci-dessous montre la mise à jour d'événement sur le *TaskModelListener* lorsqu'une tâche est créée. Il y autant d'événements que de possibilités pertinentes pour l'application (création/suppression classe, relation etc.).

```
private void fireTaskCreated(int taskModelId, Task task)
      {
```

```
            Object[] listeners = tasksModelListeners.getListenerList();
            int numListeners = listeners.length;
            for (int i = 0; i < numListeners; i += 2)
            {
            if (listeners[i] != null && listeners[i] ==
            TaskModelListener.class)
                  {
try{
((TaskModelListener) listeners[i + 1]).taskCreated(taskModelId, task);

} catch (Exception e)
      {
…
```

Cependant **MARA** n'est pas entièrement automatisée. On peut réaliser un ensemble d'actions à l'aide des différentes interfaces de configuration : ces interfaces sont des implémentations de la *mega-IHM* présentée dans le **chapitre VI**.

3 Interface(s) de configuration

L'interface de configuration principale de **MARA**, appelée **MARAve** pour **MARA** Visual Editor, autorise la distribution du système interactif sur les plates-formes. Nous avons donné dans le chapitre sur la *mega-IHM* l'exemple d'une *extra-IHM* que nous reprenons ici plus précisément. Celle que nous avons réalisée pour **MARA** nous sert d'outil pour le prototypage rapide, permettant de visualiser directement les résultats de modification de conception ou de distribution d'IHM.

3.1 Extra-IHM : outil de conception dynamique

L'extra-IHM contrôle le modèle de tâche-concept et contexte (figure 9) par l'intermédiaire du bundle « *EcosystemModelManager* ». Chacun des événements envoyés par l'extra-IHM (modification du (mega)modèle) peut être l'origine d'une transformation pour re-générer une IHM. En effet, chaque modification sur l'« *EcosystemModelManager* » peut déclencher une transformation par l'intermédiaire de *l'EvolutionEngine*.

L'extra-IHM proposée ici autorise la création, modification et suppression des concepts du domaine ainsi que des tâches.

```
public void createConcept(String name, Point2D point) {
      ecosystemModelObserver.AddConcept(window.getFilter().getShowConceptMo
del().getSelectedIndex(), name);
```

La fonction ci-dessus, située dans la partie contrôle de *l'extra-IHM*, permet de mettre à jour le modèle (code **JAVA** généré par **EMF**) lors de l'ajout graphique d'un concept. Ce code a son pendant pour les tâches.

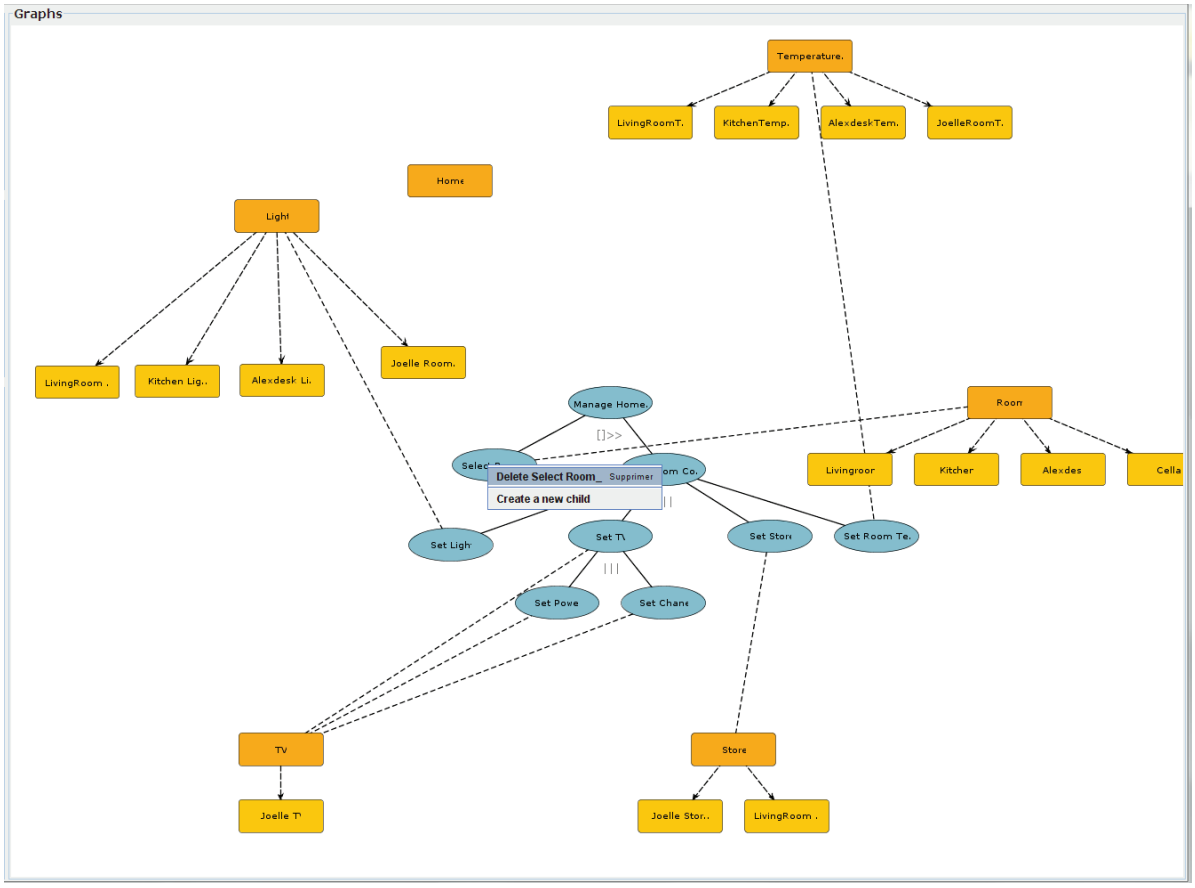

Figure 12. Edition d'un modèle de tâche et des concepts (ajouts-suppressions).

On peut ainsi éditer l'interaction et les données relatives à l'interaction. Nous joignons un panel de propriétés qui correspondent aux attributs des classes. On peut ainsi manipuler les types de tâches qui pilotent les transformations.

Dans *l'extra-IHM* que nous proposons au concepteur, nous ne permettons pas l'édition des types de tâches. En effet ces types ont été définis lors de la réalisation du modèle initial et des transformations de modèle. Les types de tâches sont un point de malléabilité pour un concepteur de plus haut niveau et ne doivent pas être redéfinis lors du test de l'IHM par *l'extra-IHM*. Ceci impliquerait de pouvoir agir de concert sur les transformations.

Cependant il est possible de changer dynamiquement le type de la tâche en fonction des types qui sont pré-paramétrés dans le modèle initial. En changeant ce type de tâche, on cible par la même occasion une règle de transformation différente pour la tâche éditée.

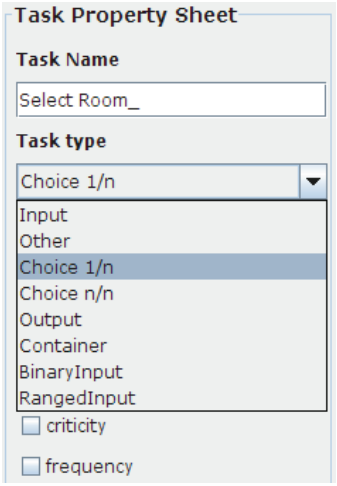

Figure 13. Edition des propriétés d'une tâche en particulier.

3.2 Extra-IHM : un outil pour la distribution

Le contexte d'usage, surtout le modèle de plate-forme, est non éditable dans *l'extra-IHM* de **MARA**. Cependant on peut manipuler par l'intermédiaire de *l'extra-IHM* les relations entre une plate-forme et une tâche. Comme nous l'avons vu dans le chapitre précédent ces relations contrôlent la distribution d'une IHM sur un ensemble de plates-formes.

A l'instar de la conception, on utilise le code JAVA généré à partir du metamodèle par **EMF** pour mettre à jour les relations entre la tâche et la plate-forme. La partie du code ci-dessous se trouve dans le bundle de l'extra-IHM (partie contrôle) et réalise la mise à jour d'une relation créée dans l'éditeur. L'appel à la fonction getShowTaskModel() renvoie l'application (représentée par son modèle de tâche).

```
public void bindTaskToPlatform(Task task, Platform platform) {

ecosystemModelObserver.BindTasktoPlatform(window.getFilter().getShowTaskMod
el().getSelectedIndex(), task.getName(), platform.getId());

}
```

Grâce aux transformations lancées dynamiquement lors de créations de liens entre tâches plates-formes, on peut contrôler si les distributions d'IHM sont pertinentes ou non. Les figures 14 et 15 présentent une situation où l'on veut afficher une tâche (pour allumer/éteindre la lumière) sur le navigateur **Firefox** de l'utilisateur. Pour ce faire, on établit une relation entre la plate-forme et la tâche (figure 9). Lorsque la relation tâche-plate-forme est établie, l'IHM est automatiquement (par transformation) créée sur le navigateur (figure 15).

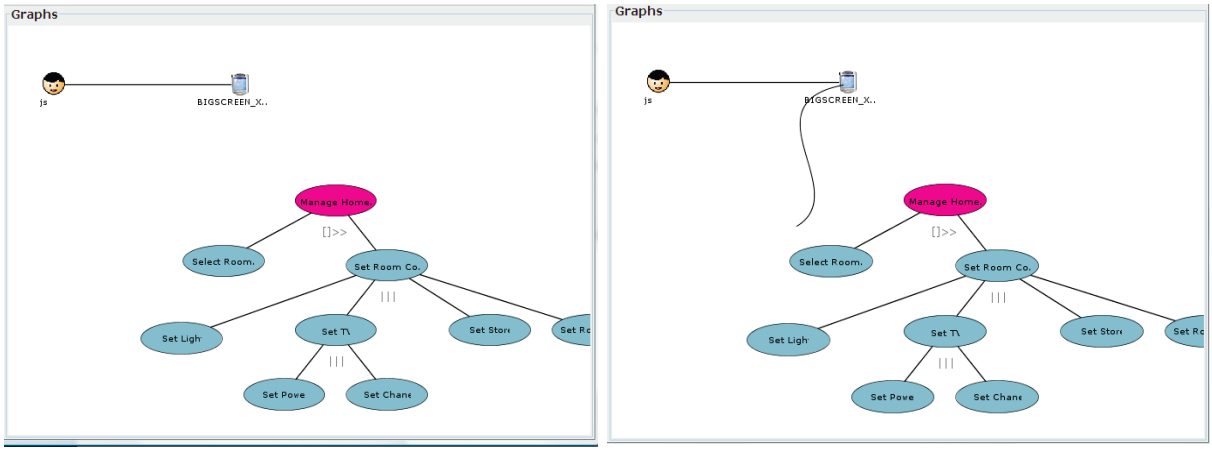

Figure 14. Distribution d'une tâche sur une plate-forme.

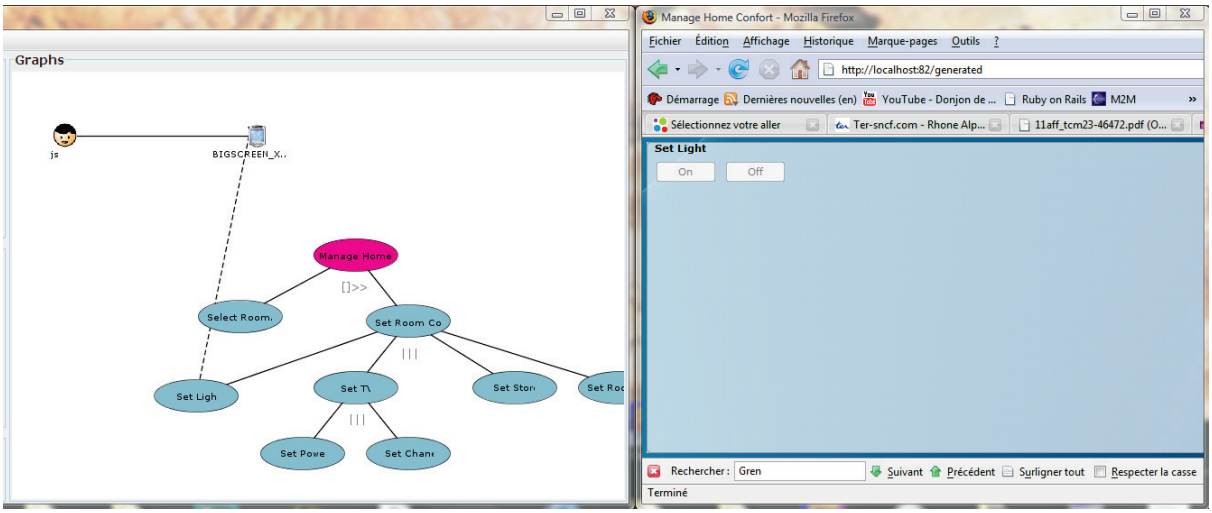

Figure 15. Tâche « allumer/éteindre » la lumière distribuée sur la plate-forme Firefox (à droite).

3.3 Extra, Meta, Trans-IHM tissées

On peut embarquer dans l'IHM générée un ensemble de vues de la *mega-IHM,* ce qui permet à l'utilisateur final de contrôler des différentes parties du *megamodèle*.

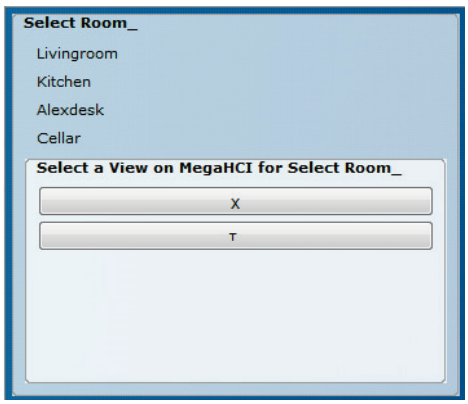

Figure 16. Accès aux Meta et Trans-IHM tissées dans l'IHM générée.

Pour notre démonstrateur nous avons réalisé des *meta,extra et trans-IHM* tissées, servant cette fois-ci au test (concepteur). Les différents boutons d'accès à la mega-IHM (figure 16) sont insérés dans la transformation de modèle.

```
megagbx : XULMetaModel!GroupBox(id<- ' mega_group'+tsk.name,
                                flex <- 1,
                                xulInteractors <- Sequence
{megacapt,button_chi,button_tau} ),


megacapt : XULMetaModel!Caption(label<-'Select a View on MegaHCI for '+
tsk.name),

button_chi : XULMetaModel!Button(label <- '&#967;', disabled <- false,
events <- thisModule.MegaUIEventBuild('chi')),

button_tau : XULMetaModel!Button(label <- '&#964; ', disabled <- false,
events <- thisModule.MegaUIEventBuild('tau'))
```

L'extrait du code **ATL** ci-dessus est inséré dans la transformation originale prise en exemple dans la section 2. Il permet de créer un sous-espace (*GroupBox*) pour chacune des tâches. On génère deux boutons (*button_chi* et *button_tau*) qui pilotent le lancement de « popups » (*MegaUIEventBuild*) correspondant respectivement à la *meta* et à la *trans-IHM*.

Dans le démonstrateur, nous n'avons pas réalisé de transformation d'ordre supérieur pour obtenir la *trans-IHM*. Nous utilisons, pour effectuer nos vues, une transformation « statique » basée sur l'outil **Acceleo**. Pour produire *l'extra-IHM* (représentant l'arbre des tâches) nous utilisons une vue arborescente (à structure tabulaire) proposée par XUL. La gestion des colonnes et des lignes est alors réalisée par appels récursifs de scripts (la récursivité est en fait ici un appel enlacé). Les feuilles de l'arbre (treecell) sont remplies du nom de la tâche (`<treecell label= "<%name%>")`.

```
<%script type="MMEcosystem.Task" name="threatTask"%>
<%if (decompositionOperator==null){%>
      <treeitem container="false" flex="1" >
                        <treerow>
                              <treecell label= "<%name%>"  />
                        </treerow>
      </treeitem>
<%}else{%>
      <treeitem container="true" flex="1" >
                        <treerow>
                              <treecell label= "<%name%>"  />
                        </treerow>
                  <treechildren>
      <%Recursive%>

<%}%>

<%script type="MMEcosystem.Task" name="Recursive"%>
<%for (decompositionOperator.subTasks){%>
            <%threatTask%>
```

```
<%}%>
</treechildren>
</treeitem>
```

On obtient à partir de ce script une *extra-IHM* tissée figure 16. Elle est accessible à partir des boutons que nous co-générons avec l'IHM. Le lien entre le bouton et cette extra-IHM se fait par nommage : l'action du bouton pointe explicitement vers le fichier généré par la transformation **Acceleo**.

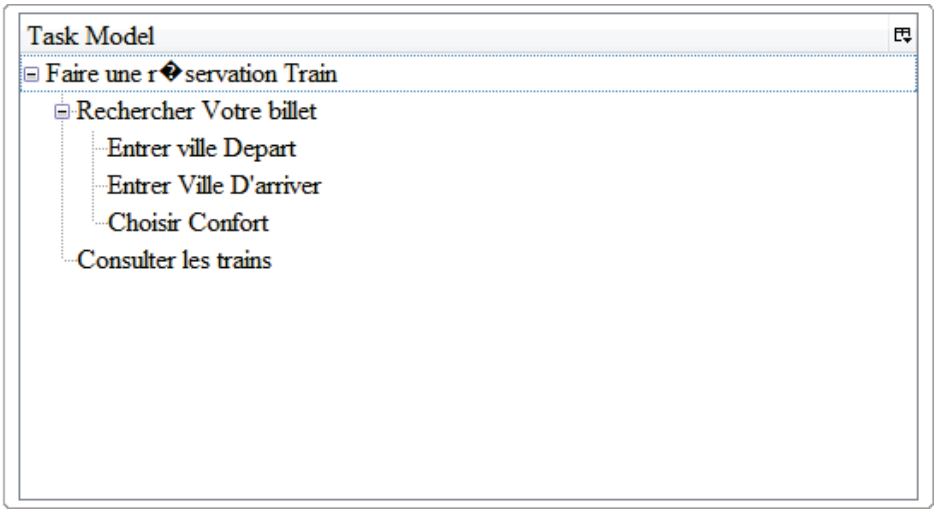

Figure 17. *Extra-IHM* d'arbres des tâches.

3.4 Impact d'une extra-IHM

L'utilisation de *l'extra-IHM* a le même impact sur **MARA** que la connexion d'une nouvelle plate-forme, c'est-à-dire qu'elle va lancer une nouvelle série de transformations par l'intermédiaire du « cerveau » de **MARA** autrement dit : « *L'EvolutionEngine* ».

L'extra-IHM agit sur le modèle du système mais réagit aussi par rapport à celui-ci par le mécanisme classique d'observer/listener (section 2.6). Dans l'état de développement actuel de **MARA**, nous n'avons pas de « bouclage infini » de modification. *L'extra-IHM* et le bundle « *platformController* » ne modifient pas la même partie de l'écosystème : une plate-forme n'est qu'observable par *l'extra-IHM*.

Cependant il faudra, dans une évolution de **MARA**, être vigilant aux éléments des modèles qui pourraient être modifiés/observés par différents *bundles*. On pourra alors employer des appels directs ou utiliser des « événements spéciaux » selon l'acteur de la modification : implémentation d'un anti-patron observer.

4 Synthèse

Dans ce chapitre, nous avons vu une infrastructure permettant la mise en œuvre des transformations dynamiques (à l'exécution de modèles dans le cadre de la plasticité). On peut facilement étendre

l'infrastructure de **MARA** pour l'appliquer de manière plus générale aux systèmes sensibles au contexte.

MARA propose également une première ébauche pour le concepteur quant aux *extra-IHM*. Un tel système est utile pour le prototypage rapide permettant de visualiser en direct le rendu d'une modélisation.

En perspective de ce travail, nous voulons rendre **MARA** indépendant du *metamodèle* autour duquel il est conçu. En effet beaucoup de composants sont spécifiques au metamodèle que nous avons développé pour la plasticité. Pour des raisons pratiques (introduction d'une nouvelle technologie) ou d'habitudes, **MARA** devra accepter la modification ou l'introduction d'un *metamodèle*. La *meta-IHM* pour le concepteur prendra alors toute son importance.

Enfin **MARA** lui-même devrait être modélisé pour permettre une auto-génération : par un noyau de base composé de transformations, de modèles et de metamodèles de **MARA**. En outre ceci nous permettrait d'approcher l'auto-adaptation à laquelle nous faisons référence dans [Sottet, et al., 2007].

Chapitre IX : Conclusion

Le concept de plasticité introduit en 1999 se place entre autres, dans la lignée des systèmes autonomes, adaptables et adaptatifs. La thèse de David Thevenin nous éclaire sur les nouveaux besoins en termes d'ingénierie logicielle pour les IHM plastiques. Nos recherches, partant de ses résultats, ont permis d'affiner au vu de l'IDM les concepts et cadres de références en IHM. Puis nous avons recherché les moyens de mettre en œuvre une adaptation dirigée par les modèles : la malléabilité. Elle sert alors d'outil pour une plasticité rigoureuse dirigée par les savoirs et savoir-faire.

Ce travail consigne la mise en œuvre de la malléabilité pour assurer la plasticité des IHM, tout en montrant une ouverture plus générale pour le Génie Logiciel. Nous en rappelons tout d'abord les contributions puis nous en détaillons les avantages et limites. Enfin nous présentons les travaux à moyen et à long terme.

1 Résumé des contributions

Nous avons centré nos réflexions sur une vision croisée entre les systèmes et environnements basés modèles pour les IHM et les concepts et technologies issues de l'IDM. Ce choix nous a amené à reconsidérer les cadres de références en IHM. Puis, forts de cette réalisation, nous avons commencé à travailler autour de l'adaptation des IHM dirigées par les modèles. Le résultat de ce travail a donné lieu à un démonstrateur validant l'approche de la plasticité par la malléabilité.

Cette démarche exploratoire nous a logiquement amenée à plusieurs contributions tant en IHM qu'en IDM. D'un point de vue conceptuel pour l'IHM nous avons proposé plusieurs axes à suivre dont l'objectif est d'assurer l'intégration et l'évolution des technologies en IHM :

- Pour capitaliser le savoir par les metamodèles et les modèles, nous avons proposé une ébauche de démarche et mis en avant les principes pour concevoir des cartographies sur des domaines d'étude. Nous avons également proposé un état de l'art dirigé par les modèles, ce qui est important pour ne pas contraindre à une seule modélisation mais préserver les usages des ingénieurs et faciliter l'intégration.

- Pour capitaliser les savoir-faire nous avons conçu un ensemble de transformations de modèles. Les transformations factorisent les axes de remodelage et de redistribution. Ces transformations respectent des propriétés d'utilisabilité [Sottet, et al.,2007]; elles se veulent concises et manipulables. Elles ne sont pas enfermées dans un outil.

Nous avons également contribué à la mise en œuvre de la malléabilité ; cette propriété sortant du cadre strict de l'IHM :

- IHM (ou vues) pour mettre en œuvre la malléabilité, c'est-à-dire rendre le système adaptable par l'humain quel que soit son rôle autour du système : utilisateur, ingénieur. On propose alors différentes IHM selon le niveau de modélisation (*metamodèle*, *modèle*, *transformation*,...) ou le rôle de l'acteur par rapport au système.

- Simplification de l'intégration et l'évolution des systèmes basés modèles.

- Réalisation d'un graphe de modèles interconnectés représentant l'application dans son contexte. Ce graphe sert de base de raisonnement pour l'adaptation, la modification et l'évolution du système qu'il représente.

D'un point de vue réalisation technique, nous avons montré plusieurs avancées par rapport à ce qui existe :

- Réalisation d'un ensemble de metamodèles liés pour représenter l'IHM en contexte.

- Transformations de modèles réalisées à la volée lors de l'exécution de l'IHM. Ceci est un atout pour le prototypage rapide car on peut modifier la modélisation lors de l'exécution et apercevoir directement les résultats.

- Réalisation de différentes IHM de configuration : meta, extra et trans-IHM.

2 Originalités et points forts

2.1 Approche pour l'évolution et vers la communauté

La malléabilité telle que nous l'envisageons passe par des représentations explicites des savoirs et savoir-faire à tout niveau d'abstraction. Par rapport aux systèmes existants, qui bien souvent cachent heuristique et langages, nous prônons une ouverture des langages couverts par les outils. Nous nous tournons vers la communauté IHM (en particulier) pour réaliser une cartographie de ces modèles et transformations : vers la réalisation d'un *Zoo* pour l'IHM.

L'ouverture des langages revêt alors plusieurs facettes dont celui de la capitalisation des connaissances mais aussi de leur manipulation par les différents acteurs autour du logiciel. Les systèmes de modélisation d'IHM existants ne couvrent bien souvent qu'une partie édition de modèles pour l'ingénieur. En réalité, la conception est plus complexe et fait appel à d'autres compétences, notamment pour les transformations.

Notre objectif se tourne aussi vers l'évolution de tels outils, en particulier pour faire face à de nouvelles contraintes, comme intégrer de nouvelles techniques d'interaction. Nous nous tournons une fois de plus vers les communautés (dans notre approche : celle de l'IHM) pour réaliser les modèles et extraire les langages inhérents à ces techniques. Notre objectif n'est pas de fournir une infrastructure pour intégrer du code mais plutôt une démarche dirigée par les modèles pour capitaliser les savoirs et savoir-faire.

2.2 Démonstrateur

Notre démonstrateur s'appuie fortement sur les metamodèles que nous avons développés. Ces metamodèles permettent de raisonner pour adapter et transformer le système interactif. Par exemple, l'arrivée d'une nouvelle plate-forme va changer le graphe de l'IHM et celui de son contexte. Toute l'originalité de ce démonstrateur réside dans la possibilité de réaliser les transformations sur le graphe de modèles lors de l'exécution de l'IHM.

En unifiant ainsi conception et exécution par l'utilisation des transformations de manière statique et dynamique, on permet le prototypage rapide d'application. Grâce aux IHM donnant accès à tous les éléments de modélisation (transformations, metamodèles, etc.) on peut modifier et tester les règles de génération, les langages, etc.

3 Limites

Les limitations sont surtout localisées au niveau du démonstrateur actuel. Notons en particulier la dépendance de notre application (**MARA**) aux *metamodèles* que nous avons réalisés. Il est coûteux en temps de concevoir une telle infrastructure, capable de modifier à la volée les langages qu'elle véhicule. C'est un facteur limitant quant à la réutilisation et l'évolution d'une infrastructure comme **MARA** ; d'autant plus que ce démonstrateur est aussi une plate-forme d'intégration de connaissances.

MARA lui-même devrait être modélisé. Il est nécessaire pour adapter **MARA,** de la même manière qu'on le fait pour les IHM, d'en posséder les metamodèles et modèles sous-jacents. En outre, cela nous permettrait de simplifier le déploiement de **MARA** en ayant un noyau minimal (*boot-strap*) de modèles et de transformations décrivant son infrastructure.

Nous n'avons pas pu, dans le temps imparti, développer complètement les représentations nécessaires à tous les acteurs (ingénieurs, utilisateur,…). Les extra, meta et trans-IHM font appel à des représentations spécifiques pour lesquelles nous avons besoin de collaborer avec des spécialistes de la visualisation et de la manipulation d'information.

4 Perspectives

4.1 A court terme

Les premières améliorations à apporter sont d'ordre implémentationnel : d'une part il s'agit de réaliser d'autres IHM de configuration et rendre par conséquent **MARA** plus malléable, notamment au niveau des transformations (*trans-IHM*). D'autre part, il est nécessaire d'établir une documentation permettant de comprendre et d'assimiler l'infrastructure de **MARA**, et surtout d'appréhender la philosophie de ce système fortement ancré dans l'IDM. C'est-à-dire que les extensions, améliorations et évolutions de **MARA** doivent passer par des metamodèles, modèles et transformations. L'idée majeure est de capitaliser les savoir-faire de manière accessible dans les modèles et leurs IHM, plutôt que de les diluer dans un code qui ne cesse de grossir malgré sa modularité.

Enfin pour l'IHM il est préférable d'améliorer les règles de transformations : avoir une base plus étoffée, une meilleure couverture des critères ergonomiques choisis et de nouvelles technologies cibles comme **SWING**.

4.2 A Moyen terme

D'un point de vue conceptuel, tout comme implémentationel, il est nécessaire d'intégrer de nouvelles modélisations comprenant : les nouvelles techniques d'interaction, de nouveaux aspects tels que la sécurité, etc. Ceci va bien entendu de pair avec les transformations associées à ces nouvelles techniques.

D'un point de vue purement conceptuel, pour l'IHM, il est primordial d'améliorer l'intégration de nouveaux critères et de références ergonomiques dans nos transformations. Ceci doit amener à trouver de nouvelles métriques « ergonomiques » sur les modèles et les relations entre modèles.

4.3 A long terme

MARA doit être instrumenté pour pouvoir mettre en œuvre le principe fondamental pour la malléabilité : toutes évolutions ou extensions passent par une modélisation et une représentation explicite des savoirs et savoir-faire. Ceci implique d'avoir à la fois des metamodèles non figés et une infrastructure qui ne repose plus sur un seul metamodèle.

Tous les niveaux de modélisation doivent alors être observables/manipulables par des IHM de configuration. Ces IHM s'intégreront dans cette nouvelle infrastructure évolutive ; ainsi nous offrirons une malléabilité complète.

Les nouvelles techniques de visualisation et de manipulation de l'information pourront être proposées pour la mise en place de la mega-IHM. Ceci pourra être le fruit d'une collaboration entre

spécialiste IDM et spécialiste de la visualisation d'information. L'intérêt est de proposer les bonnes représentations (IHM) aux différents acteurs selon leur niveau de compétence et selon leur but par rapport au système qu'ils le conçoivent ou qu'ils l'utilisent.

.

Annexes

Annexe 1 : exemple de scénario & motivation pour les passerelles entre langages

Dans ce travail, un scénario a été étudié, répondant à une problématique de multiples langages dans le contexte d'un réseau d'excellence Européen : Similar [Similar]. Dans le cadre de ce projet, plusieurs partenaires travaillent sur des problématiques proches, avec des langages différents et donc des outils différents (**CTTe**, **IdealXML**, **KMade**, etc.). Nous avons étudié comment intégrer les outils de **IdealXML** dans une chaîne de transformations pour, d'une part profiter des outils du langage **UsiXML** (utilisé par **IdealXML**), d'autre part favoriser le travail d'utilisateurs plus habitués à **UsiXML**.

La figure1 présente un ensemble de transformations permettant le passage d'un langage vers un autre. Deux types de scénarii sont possibles :

- l'utilisateur décide de passer d'un modèle de tâche décrit vers un modèle de tâche écrit en **UsiXML** (**IdealXML**) pour, par exemple, réaliser une interface utilisateur abstraite (IUA) car il est expert **d'IdealXML**. (Transformation du haut de la figure).
- L'utilisateur veut avoir recours à la transformation qui permet de générer l'IUA dans **MARA** et la visualiser dans **IdealXML**. (Transformation Diagonale et basse).

Dans ce deuxième cas, on fait appel à la retro-conception de transformation (voir section 3.2).

Le premier scénario demande de connaître non seulement le metamodèle de « départ », ici **MARA**, mais aussi le metamodèle « d'arrivée » (*Task **UsiXML***) ainsi que son format utile (XSD de **UsiXML** pour les tâches). On enchaîne alors deux transformations, modèles vers modèles puis modèles vers texte.

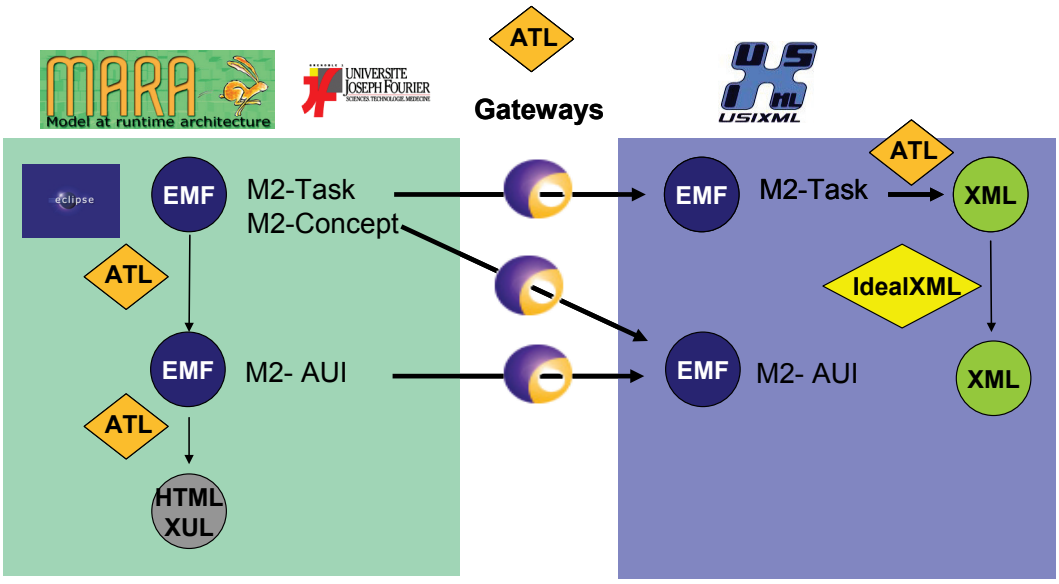

Figure 1. Ponts entre langages et outils : **MARA** vers **IdealXML.**

Nous avons instrumenté ce passage dans une ancienne version de MARA. Dans la figure 2, nous présentons dans notre éditeur de tâche (*extra-IHM*) un moyen d'enregistrer cet arbre au format **UsiXML** par transformation de modèle conformément à la figure 1.

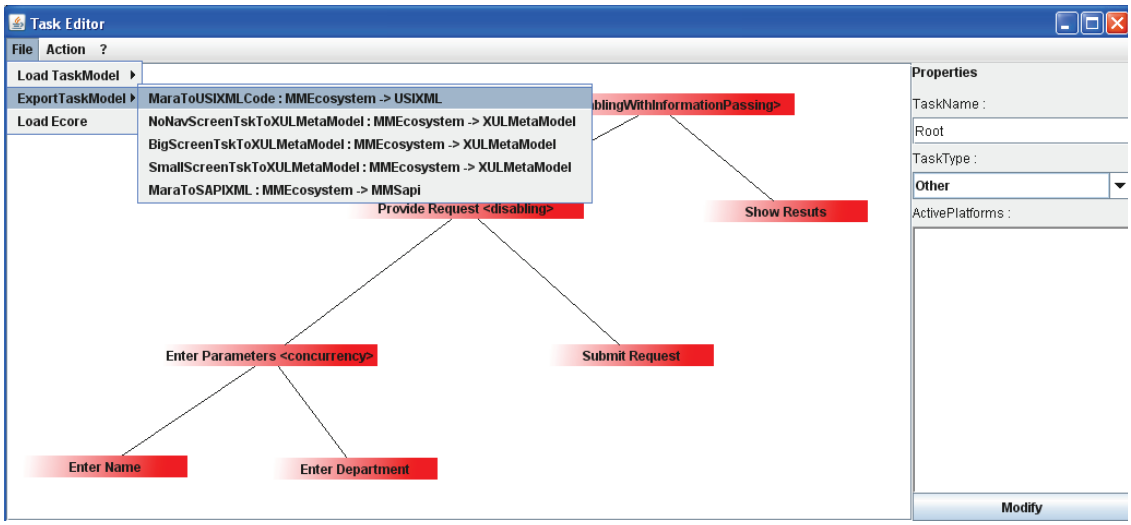

Figure 2. Export de modèles dans **MARA.**

Annexe 2 : comparaison de la structure entre MARA et K-MADe

Nous avons choisi ces outils pour la proximité de leur inspiration au niveau du modèle de tâche. **CTTe** et **IdealXML** sont très liés, ils conservent la même structure d'arbre de tâches : une décomposition des tâches en sous-tâches et, sur un même niveau hiérarchique, une relation binaire entre deux tâches. Ce sont deux éditeurs qui conservent le même objectif : faciliter la conception et dériver facilement un premier modèle d'interface utilisateur abstraite.

MARA et **K-MADe**, proposent une structure arborescente plus épurée : un seul opérateur par niveau de hiérarchie qui « pilote » la décomposition. Ceci facilite l'objectif de ces deux systèmes : pour **MARA** la génération d'IHM et pour **K-MADe** la simulation de l'exécution d'un arbre. Les annotations dans **MARA** sont le cœur de la génération d'IHM : ce sont elles qui pilotent les transformations de modèles. Les annotations (*Label*) dans **K-MADe** sont uniquement là pour le concepteur.

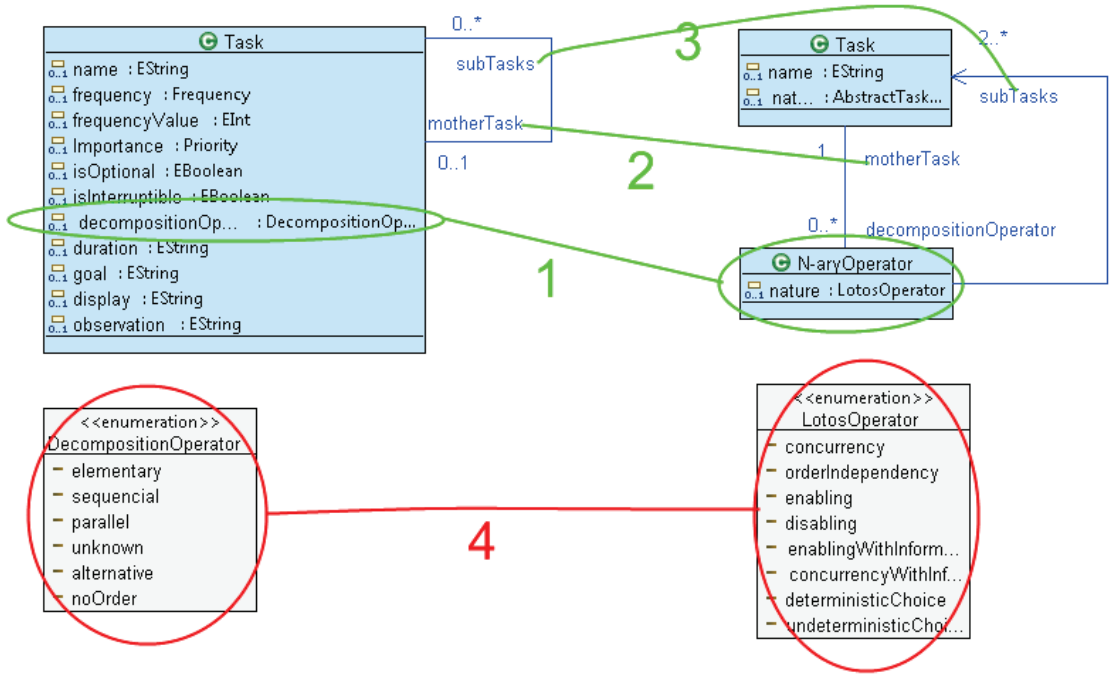

Figure 3. Comparaison (équivalence) entre la structure de tâche de **K-MADe** (à gauche) et **MARA** (à droite).

Sur deux metamodèles ayant une structure de tâche équivalente on se rend compte qu'il ya plusieurs manières de modéliser les choses. Dans le cas de l'application **K-MADe** (figure 3 à gauche) la décomposition d'une tâche en sous-tâche se fait par l'intermédiaire d'un *decompositionOperator* attribut de la tâche. Celui-ci est obligatoire pour une tâche qui a des sous-tâche. En **OCL** cela donnerait : Task.subTasks->exists() implies Task.decompositionOperator->exists(). Cette contrainte

est vérifiée dans **MARA** de par la structure du metamodèle (figure 3 à droite). On peut alors comparer les deux metamodèles sur leur structure en présentant leur équivalence (points 1, 2, 3 de la figure 3) et les différences (point 4 de la même figure).

Dans la figure 2, on constate que l'attribut *decompositionOperator* (figure 3 partie gauche **K-MADe**) trouve son pendant dans la classe *N-aryOperator* (figure 3 partie droite **MARA**). Il en est de même pour la décomposition d'une tâche en sous-tâche. On retrouve d'ailleurs les mêmes noms de rôle sur les relations de **MARA**. L'operateur de décomposition de **MARA** vient prendre place au centre de la relation *subTasks-MotherTask de* **K-MADE** : point 2 et 3 de la figure 3.

Par contre, la nature des opérateurs de décomposition est différente entre les deux metamodèles (point n°4 de la figure 3). Par exemple, l'opérateur *unknown* n'a aucune équivalence en operateur Lotos.

Annexe 3 : cartographie des environnements et outils pour l'IIHM

Nous délivrons ici, une vision globale de la cartographie (état de l'art) peu lisible : il faudrait des outils d'explorations. Chacun des nœuds représente un élément de modélisation (modèle, metamodèle, transformation).Par conséquent il faut voir ce graphe selon différents niveaux de zoom sémantiques : à l'intérieur d'un nœud il y a différents diagrammes et textes pour définir plus précisément chacun des éléments.

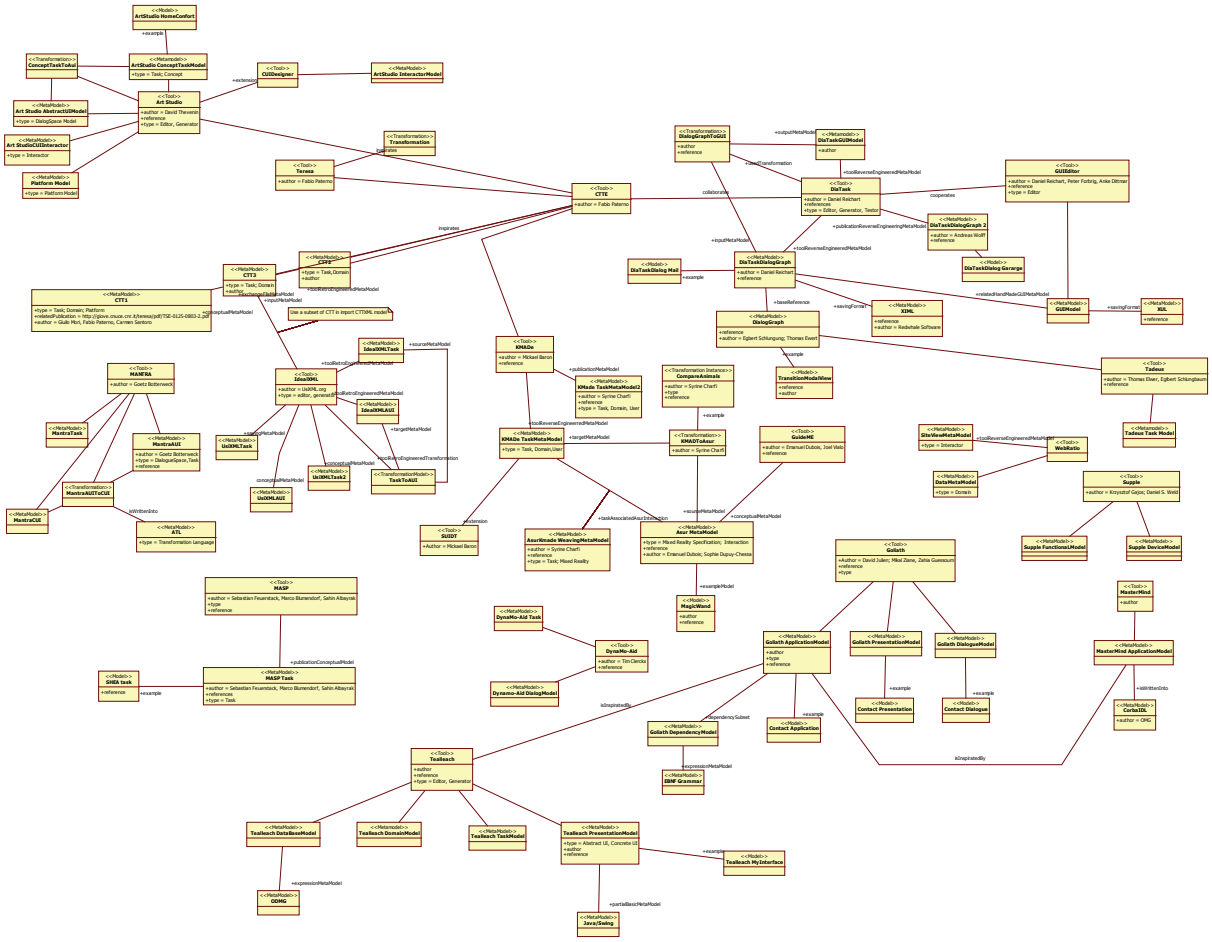

Figure 4. Cartographie des outils pour l'IIHM : vue générale.

Visualisation globale de la cartographie par rapport aux différents pays.

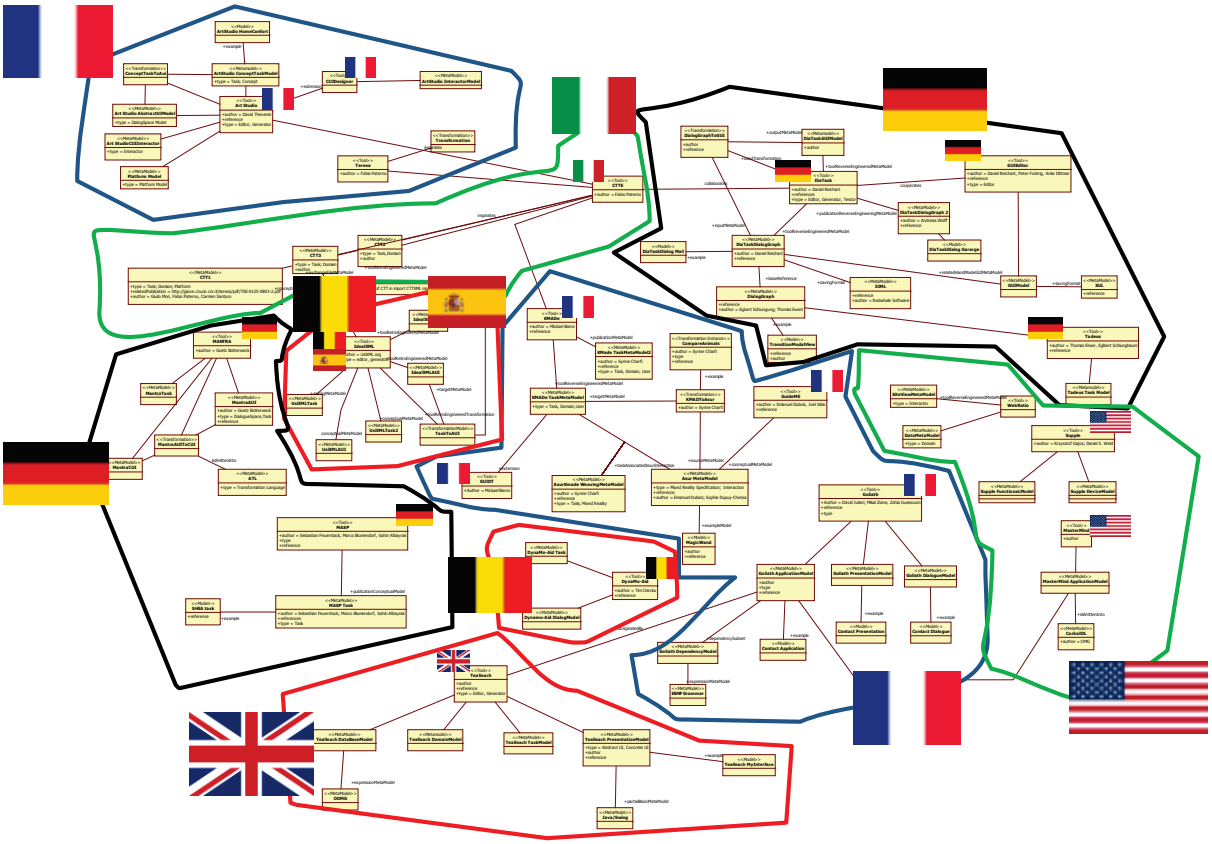

Figure 5. Cartographie des outils par rapport au pays de leurs auteurs.

Bibliographie

Acceleo Acceleo : http://www.acceleo.org .

AMW-UseCases Calculating differences between models. - http://www.eclipse.org/gmt/amw/usecases/diff/ .

Annett J and Ducan K D Task analysis and training desing Journal of Occupation Psychology. - 1967. - Vol. 41. - pp. 211-221.

AtlanticZoo Atlantic Zoo : http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/ .

Bézivin Jean On the unification power of models Software and System Modeling. - 2005. - pp. 171-188.

Bézivin Jean [et al.] Bridging the MS/DSL Tools and the Eclipse Modeling Framework Software Factory workshop at OOPSLA2005. - 2005.

Balme Lionel [et al.] CAMELEON-RT: a Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces - EUSAI 2004.

Balme Lionel Interfaces homme-machine plastiques : Une approche par composants dynamiques Université Joseph Fourier. - Grenoble 1. - 2008.

Baron Mickael [et al.] K-MADe : un environement pour le noyau du modÈle de description de l'activité Interaction Homme Machine (IHM 06). - 2006.

Bastien J M and Scapin D Ergonomic Criteria for the Evaluation of the Human-Computer. -INRIA Technical Report 1993.

Behring Alexander [et al.] EMODE-D2.3-German_Transformations. Delivrable of ITEA project .

Bergh Jan Van and Coninx Karin Model-Based Design of Context-Sensitive Interactive Applications: a Discussion of Notations TAsk MOdel and DIAgram TAMODIA2004. - Prague Czech Republic , 2004.

Biermann Enrico [et al.] EMF Model Refactoring based on Graph Transformation Concepts Electronic Communications of the EASST. - 2006.

Bodart François [et al.] Computer-Aided Window Identification in TRIDENT Conference on Human-Computer Interaction INTERACT'95 1995.

Bouillon Laurent [et al.] Reverse Engineering of Web Pages based on Derivations and Transformations 3rd Latin American Web Congress LA-Web'2005 . - 2005. - p. 13.

Bruening Jens, Dittmar Anke and Forbrig Peter Getting SW Engineers on Board: Task Modelling with Activity Diagrams Engineering Interactive Systems, Salamanca, Spain. - 2007.

Burnett Margaret [et al.] Workshop Report : From End-User Programming to End-User Software Engineering - 2006.

Calvary Gaelle, Coutaz Joelle and Thevenin David A Unifying Reference Framework for the Development of Plastic User Interfaces Engineering for Human Computer Interaction(EHCI'01). - 2001.

Clerckx Tim, Luyten Kris and Coninx Karin The Mapping Problem Back and Forth : Customizing Dynamic Models while preserving Consistency TAsk MOdel and DIAgram (Tamodia'04). - Prague Czech Republic , 2004.

Clerckx Tim, Luyten Krix and Coninx Karin DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development Engineering Human Computer Interaction and Interactive System Joint working conferences EHCI-DSVIS. - 2004.

Coninx Karin [et al.] Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems, MobileHCI03 2003.

Costa Duarte, Nobrega Leonel and Nunes Nuno Jardim An MDA Approach for Generating Web Interfaces TAsk MOdel and DIAgrams for user interfaces design (Tamodia'06). - 2006.

Coutaz Joëlle Meta-User Interfaces for Ambient Spaces Invited talk. In proceeding TAsk MOdel and DIAgrams for user interfaces design (Tamodia'06) 2006.

Demeure Alexandre Modèles et outils pour la conception et l'éxécution des Interfaces Homme-Machine Plastiques Université Joseph Fourier. - Grenoble 1. - 2007.

DSL-TOOLS - http://msdn.microsoft.com/en-us/library/bb126235.aspx

Dupuy-Chessa Sophie and Emmanuel Dubois Requierements and Impacts of Model Driven Engineering on Mixed Systems Design Premières Journées sur l'Ingénierie Dirigée par les Modèles (IDM05). - Paris. - 2005.

Eclipse-M2M Eclipse Model to Model (M2M) project - http://www.eclipse.org/m2m .

EMF - The Eclipse Modelling Framework - http://www.eclipse.org/modeling/emf/.

Emode Project ITEA Enabling Model Transformation-Based Cost Efficient Adaptive Multi-modal User Interfaces (EMODE). - http://int.emode-projekt.de/. - 2005.

Fabro Marco Didonet [et al.] AMW: a generic model weaver Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles. - 2005.

Favre Jean-Marie Foundations Of MetaPyramids Languages Vs Metamodels EpisodeI I Story Of Thotus The Baboon. - Daghstull Germany. - 2004.

Favre Jean-Marie Foundations Of Model Driven Reverse Engineering Models Episode II Stories Of The Fidus Papyrus And Of The Solarus Dagshtull. - Germany. - 2004.

Favre Jean-Marie, Godet-Bar Guillaume and Sottet Jean-Sébastien Atelier sur les LAngages Notations Ontologies Transformations (ALANOTr) 2007.

Feuerstack Sebastian, blumendorf Marco and Albayark Sahin Prototyping of Multimodal Interactions for Smart Environments based on Task Models European Conference on Ambient

Intelligence: Workshop on Model Driven Software Engineering for Ambient Intelligence Applications. - Germany. - 2007. .

Feuerstack Sebastian, Blumendorf Marco and Sahin Albayark Bridging the Gap between Model and Design of User Interfaces European Conference on Ambient Intelligence: Workshop on Model Driven Software Engineering for Ambient Intelligence Applications. - 2007.

Gajos Krzysztof and Weld Daniel S SUPPLE: Automatically Generating User Interfaces Intelligent User Interfaces IUI 04. - 2004.

Ganneau Vincent, Calvary Gaelle and Demumieux Rachelle Learning Key Contexts of Use in the Wild for Driving Plastic User Interfaces Engineering Engineering Interactive Systems 2008 (2nd Conference on Human-Centred Software Engineering (HCSE 2008) and 7th International workshop on TAsk MOdels and DIAgrams (TAMODIA 2008) Pisa, Italy. - 2008.

Gauffre Guillaume, Dubois Emmanuel and Bastide Remi Domain Specific Methods and Tools for the Design of Advanced Interactive Techniques MDDUI 2007 @Models Conference. - 2007.

Gerber Anna [et al.] Transformation : The Missing Link of MDA Graph Transformation. - 2002.

GME - Generic Modeling Environment - http://www.isis.vanderbilt.edu/projects/gme/

GMT Eclipse the Eclipse Generative Modeling Technologies (GMT) project - http://www.eclipse.org/gmt .

Goetz Botterweck and Hampe J Felix Caputring The Requierements For Multiple User Interfaces MDDAUI06 @ Models Conference. - Italy , 2006.

Gray Jules Whiteand and Schmidt Douglas C Constraint-based Model Weaving Transactions on Aspect-Oriented Software Development, submitted to the Special Issue on Aspects and Model Driven Engineering. - 2008.

Greenyer Joel A study of Model Transformation Technologies : Reconciling TGGs with QVT. - Master Thesis - University of Panderborn. - 2006.

Griffiths Tony [et al.] Teallach : a model-based user interface development environment for object databases Interacting with Computer. - 1999.

Groupe Object Management OMG's MetaObject Facility (MOF) Home Page.

Heinrich Matthias [et al.] MDA Applied: A Task-Model Driven Tool Chain for Multimodal Applications Task Model and Diagrams for USer Interfaces Design. - 2007.

Horvath Akos and Varro Daniel The VIATRA2 Model Transformation Framework . Advanced school on visual modelling techniques (segravis) 2006.

IEEE 1471 -IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - IEEE Std 1471 www.pithecanthropus.com/~awg/public_html/ -2000.

IKV++ Medini QVT Medini QVT - Transformation Engineers http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77 .

Jouault Frédéric, Jean Bézivin and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering GPCE'06 : Proceedings of the fifth international conference on Generative Programming and Component Engineering. - 2006.

Kaltz Joachim Wolfgang An Engineering Method for Adaptive - Context-Aware Web Applications Duisburg-Essen. - 2006.

Kephart Jeffrey O and Chess David M The Vision of Autonomic Computing. - Computer Magazine. - 2003.

Kermeta - http://www.kermeta.org/.

Kurtev Ivan, Bézivin Jean and Aksit Mehmet Technological Spaces : an Initial Appraisal - In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.

Limbourg Quentin [et al.] UsiXML: a Language Supporting Multi-Path Development of User Interfaces Workshop on Design, Specification, and Verification of Interactive Systems, HYPERLINK "http://www.se-hci.org/ehci-dsvis04/" \t "_blank" EHCI-DSVIS'2004 ,Hamburg, July 11-13, 2004.

Limbourg Quentin and Vanderdonckt Jean Addressing the Mapping Probelm in User Interface Design with UsiXML TAsk MOdel and DIAgrams - TAMODIA'04. - 2004.

Lohmann Steffen, Kaltz J wolfgang and Ziegler Jurgen Dynamic Generation of Context-Adaptive Web User Interfaces through Model Interpretation: MDDAUI06, 2006.

Lyria Lyria Solution pour developer des IHM multi-cibles.

MDR - MetaData Repository - http://mdr.netbeans.org/.

Mens Tom, Czarnecki Krzyzstof and Gorp Pieter Van A taxonomy of Model Transformations Dagsthul Seminar. - Germany. - 2004.

Milewski Miroslaw and Roberts Graham The Model Weabing Description Language (MWDL) - towards a formal Aspect Oriented Language for MDA model Transformations. - 2005.

Modisco Model Discovery (Modisco) - http://www.eclipse.org/gmt/modisco/.

MOFLON - http://www.moflon.org/.

Mori Giulio, Paterno Fabio and Santoro Carmen Design and development of MultiDevice User Interfaces through Multiple Logical Descriptions IEEE Transactions on software Engineering. - 2004.

Mori Guilo, Paterno Fabio and Santoro Carmen CTTE : Support for Developing and Analyzing Models for Interactive System Design IEEE Transactions on software Engineering. - 2004.

Myers Brad A, Hudson Scott E and Pausch Randy Past, Present and Future of User Interface Software Tools. - ACM TOCHI. - 1999.

NetFlective NetFlective.

Nichols Jeffrey [et al.] Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances. - UIST'06. - 2006.

Nichols Jeffrey [et al.] Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances IEEE International Conference on Multimodal Interfaces. - 2002.

Nichols Jeffrey Huddle Project Page 2006.

Nichols Jeffrey, Myers Brad A and Rothrock Brandon UNIFORM: Automatically Generating Consistent Remote Control User Interfaces Computer-Human Interaction 2006 : CHI. - Montreal, Canada , 2006.

Nielsen J Heuristic evaluation - *Usability Inspection Methods* **-** 1994.

OMG-MDA The Model Driven Architecture - http://www.omg.org/mda/ .

OMG-MOF Meta Object Facility (MOF) - http://www.omg.org/mof .

OMG-QVT - Query View Transformation - HYPERLINK "http://www.omg.org/docs/ptc/05-11-01.pdf" \o "http://www.omg.org/docs/ptc/05-11-01.pdf" http://www.omg.org/docs/ptc/05-11-01.pdf .

Org Joomla. Joomla! .

Osgi Osgi Alliance : osgi.org .

Phanouriou Constantinos UIML : A Device-Independent User Interface Markup Language - PhD thesis Virginia Polytechnic Institute and State University. - 2000.

Ponnrkanti Shankar R [et al.] ICrafter : A Service Framework for Ubiquitous Computing Environments UbiComp 2001. - 2001.

Popfly Microsoft http://www.popfly.com/ .

Reichart Daniel, Forbrig Peter and Dittmar Anke Task Models as Basis for Requirements Engineering and Software Execution TAsk MOdel and DIAgrams for user interface design TAMODIA. - Prague , 2004.

Rey Gaetan Contexte en Interaction Homme-Machine : le contexteur - Thèse de doctorat Université Joseph Fourrier. - 2005.

Scapin Dominique and Pierret-Goldbreich C MAD : une méthode analytique de description des tâches. - *Colloque sur l'ingénierie des interfaces homme-machine, Sophia-Antipolis.* - 1990.

Schlungbaum Egbert and Elwert Thomas Dialogue Graphs - a Formal and Visual Specification Technique for Dialogue Modelling BCS-FACS Workshop on Formal Aspects of the. - United Kingdom , 1996.

Seffah A, Donayee M and Kline R Usability and quality in use measurement and metrics : An integrative model. Software Quality Journal. - 2004.

Silva Paulo Pinheiro and Paton Norman W A UML-based Design Environment for Interactive Applications 2nde International Workshop on User Interface to Data Intensive Systems (UDIS'01). - Zurich , 2001.

Silva Paulo Pinheiro and Paton Norman W User Interface Modeling in UMLi IEEE Software. - 2003. - pp. 62-69.

Silva Paulo Pinheiro User Interface Declarative Models and Development Environments : A Survey 7th International Workshop, DSV-IS 2000. - Ireland : Springer. - 2000.

Similar Similar Network of Excellence : http://www.similar.cc .

SLE08 Software Language Engineering Conférence : http://planet-sl.org/sle2008.

Sottet Jean-Sébastien, Calvary Gaelle and Favre Jean-Marie IDM05-Sottet Premieres Journées sur L'ingénierie Dirigée par les Modèles 2005. - Paris, France. - 2005.

Sottet Jean-Sébastien, Calvary Gaelle and Favre Jean-Marie Towards Mapping and Model Transformation for Consistency of Plastic User Interfaces In Workshop on The Many Faces of Consistency in Cross-platform Design, ACM conf. on Computer Human Interaction, Montréal, 2006.

Sottet Jean-Sébastien [et al.] A Model-Driven Engineering Approach for the Usability of User Interfaces (2007) In Proceedings of Engineering Interactive Systems (EIS) 2007.

Sottet Jean-Sebastien [et al.] Model-Driven Adaptation for Plastic User Interfaces In i International Conference on Human-Computer Interaction, Interact- Rio de Janeiro 2007.

Szekely Pedro Retrospective and Challenges for Model-Based Interface Development - 1996.

Thevenin David Adaptation en Interaction Homme-Machine : le cas de la plasticité: 2001.

Thevenin David, Coutaz Joelle and Calvary Gaelle A Reference Framework for the Development of Plastic User Interfaces Multiple User Interfaces. - 2003.

Topcased http://topcased.gforge.enseeiht.fr/.

University Carnegie Mellon Pebbles Personal Universal Control The Pebbles project is investigating how a hand-held computer can be used as a "Personal Universal Controller".

Vanderdonckt Jean Règles ergonomiques de sélection d'objets interactifs pour une information simple. - *6ième Colloque Ergonomie et Informatique Avancée : ErgoIA'98. Biarritz 4-6 Novembre* - 1998.

Wagelaar Dennis and Jonckers Viviane Explicit Platform Model for MDA. -Models Conference. - 2005.

Welie M van Task-based User Interface Design PhD Thesis -Vrije Universiteit Amsterdam. - 2001.

White, Jules; Gray, J. and Schimdt, D. C -.Contraint-based Model Weaving - *Transactions on Aspect-Oriented Software Development* (Special Issue on Aspects and MDE). - 2008

Wolff Andreas [et al.] Linking GUI Elements to Tasks Supporting an Evolutionary Design Process TAsk MOdel and DIAgrams for user interface design TAMODIA 2005. - 2005.

Wolff Andreas, Forbrig Peter and Reichart Daniel Tool Support for model-based Generation of Advanced User-Interfaces MDDAUI 05 @ Models Conference. - Jamaica : CEUR, 2005.

Zhao Xulin and Zou Ying A Framework for Incorporating Usability into Model Transformations MDDAUI07@Models Conference. - Nashville, Tennessee, USA, 2007.

Zoomm Zoo of MetaModels (Zooomm)- http://zooomm.org/ .