Vers une programmation des systèmes interactifs centrée sur la spécification de modèles exécutables

Olivier Beaudoux

▶ **To cite this version:**

Olivier Beaudoux. Vers une programmation des systèmes interactifs centrée sur la spécification de modèles exécutables. Informatique [cs]. Université d'Angers, 2014. tel-01087628

HAL Id: tel-01087628
https://inria.hal.science/tel-01087628v1

Submitted on 27 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Vers une programmation des systèmes interactifs centrée sur la spécification de modèles exécutables

Olivier Beaudoux
Équipe TRAME, Groupe ESEO, Angers

Mémoire présenté pour l'obtention de
l'Habilitation à Diriger des Recherches en Informatique
délivrée par l'Université d'Angers

HDR soutenue le 28 août 2014 à l'ESEO devant le jury composé de :

Stéphane Chatty	Directeur du LII, ENAC, Toulouse	Rapporteur
Philippe Palanque	Professeur, Université Paul Sabatier, Toulouse	Rapporteur
Nicolas Roussel	Directeur de Recherche, INRIA Lilles	Rapporteur
Michel Beaudouin-Lafon	Professeur, Université de Paris Orsay	Examinateur
Jean-Marc Jézéquel	Professeur, Université de Rennes 1	Examinateur
Frédéric Saubion	Professeur, Université d'Angers	Président
Olivier Paillet	Directeur du groupe ESEO, Angers	Membre invité

Remerciements

Je tiens à remercier celles et ceux qui ont contribué à ce que je puisse poursuivre les travaux initiés dans ma thèse pendant plus de 10 ans, travaux qui trouvent un nouvel aboutissement dans ce mémoire d'HDR.

Je remercie en premier lieu l'ESEO, en particulier la direction générale, Victor Hamon, Jacky Charruault et Olivier Paillet, et la direction de la recherche, Patrick Plainchault, pour m'avoir fait confiance en me donnant un contexte favorable à la construction de mon projet de recherche. Je remercie également l'ensemble des enseignants-chercheurs de l'ESEO avec lesquels les échanges scientifiques ont nourri ma passion pour la recherche. Je n'énumère pas la liste, elle serait trop longue !

Je remercie chaleureusement les membres du jury, Michel Beaudouin-Lafon, Stéphane Chatty, Jean-Marc Jézéquel, Philippe Palanque, Nicolas Roussel et Frédéric Saubion, d'avoir accepté de puiser dans la ressource rare qu'est le temps et j'espère vivement que leur engagement dans cet exercice leur a donné la satisfaction légitimement attendue.

Je remercie Olivier Barais, Benoît Baudry, Arnaud Blouin, Benoît Combemal, Jean-Marc Jézéquel, pour m'avoir intégré à leur équipe de recherche DiverSE de l'IRISA en tant que collaborateur extérieur. Ils ont pleinement participé, par les échanges riches, les projets et les publications, à mon engouement pour le domaine du génie logiciel et plus particulièrement pour celui de l'ingénierie dirigée par les modèles.

Je remercie Jean-François Bourdet, Pascal Leroux, Philippe Teutsch, de l'université du Maine, pour les projets sur lesquels nous continuons d'œuvrer dans le domaine des TICE.

Je remercie Stéphane Loiseau, professeur au LERIA et co-encadrant de la thèse d'Arnaud Blouin, pour son implication dans cette thèse. Je remercie Arnaud Blouin pour l'engagement qui a été le sien pendant sa thèse. Son travail a beaucoup contribué à la réflexion menée sur l'ingénierie des systèmes interactifs, notamment pour la partie modèle conceptuel.

Je remercie Mengqiang Yang pour le travail qu'il a réalisé dans sa thèse CIFRE, laquelle n'a pas pu aboutir pour des raisons indépendantes de sa volonté. Je remercie également Khalil Khalifa pour m'avoir fait confiance en me demandant de co-encadrer cette thèse.

En je n'oublie pas, bien entendu, ma famille, Emmanuelle, Alice et Audrey, qui m'ont soutenu pendant ce chemin et ont su supporter ma passion avec tout ce que cela comporte. Je leur en suis infiniment reconnaissant.

Table des matières

Introduction **1**
Constat . . . 1
Approche adoptée . . . 2
Structure du mémoire . . . 2

1 État de l'art **3**
1.1 Évolution des composantes de développement des SI . . . 3
1.1.1 Les plates-formes matérielles et logicielles . . . 3
1.1.2 Les boîtes à outils . . . 6
1.1.3 Les langages . . . 7
1.1.4 Conclusion . . . 8
1.2 Modèles et ingénierie des IHM . . . 9
1.2.1 Modèles conceptuels . . . 9
1.2.2 Ingénierie dirigée par les modèles . . . 12
1.2.3 Conclusion . . . 14

2 Contribution **17**
2.1 Présentation du parcours . . . 17
2.1.1 Ingénierie des systèmes interactifs . . . 17
2.1.2 Transition vers l'Ingénierie Dirigée par les Modèles . . . 18
2.1.3 Une approche fondée sur l'IDM . . . 19
2.1.4 Projets actuels . . . 19
2.2 Vue d'ensemble de la contribution . . . 20
2.3 Le modèle DPI . . . 21
2.3.1 Modèle conceptuel . . . 22
2.3.2 Modèle de composant . . . 25
2.3.3 Conclusion et discussion . . . 28
2.4 Transformations actives et correspondances . . . 29
2.4.1 eXAcT : un processeur de transformations actives . . . 30
2.4.2 Malan : Un langage de correspondances . . . 32
2.4.3 AcT.NET : Un environnement centré sur les correspondances . . . 35
2.4.4 Conclusion . . . 39
2.5 Vers un cadre logiciel unifié . . . 40
2.5.1 Le cadre logiciel Malan/Malai . . . 40
2.5.2 Théorie des opérations actives . . . 46
2.5.3 Le cadre logiciel Loa . . . 58

3 Conclusion et perspectives **67**
3.1 Synthèse des travaux 67
3.2 Perspectives 68
3.2.1 Passerelles entre les modèles 68
3.2.2 Diversité des plates-formes d'exécution 69
3.2.3 Application de contraintes 70
3.2.4 Documents composites et partage 70
3.2.5 Visualisation d'information 71

Bibliographie **73**

Glossaire **81**

Annexes **83**

Introduction

Ce mémoire présente l'ensemble des travaux de recherche que j'ai menés depuis 2000 autour de l'ingénierie des Systèmes Interactifs (SI). Ces travaux ont pris naissance lors de ma thèse soutenue en 2004 [13], ont été poursuivis pendant une dizaine d'années, et ont finalement donnés lieu à la réalisation d'un cadre logiciel unifiant cet ensemble de travaux. Ce cadre a notamment été présenté dans différentes conférences internationales [24, 25, 29, 28].

Ce chapitre introduit le contexte de l'ensemble de ces travaux. La problématique adressée est présentée sous la forme d'un constat dans la première section. La section suivante donne les grandes lignes de l'approche que je propose pour tenter de résoudre cette problématique. La dernière section précise la structure générale du mémoire.

Constat

Le développement des SI, au sens de la programmation, met en jeu un nombre significatif de composantes logicielles et matérielles sélectionnées parmi un panel très large et souvent versatile. Ainsi, chaque développement s'appuie sur un langage de programmation, une ou plusieurs boîtes à outils dédiées au développement d'IHM, une plate-forme d'exécution logicielle et une plate-forme matérielle comportant les périphériques d'entrée et de sortie utilisateur. Par exemple, une application interactive pour plate-forme mobile Android est construite sur la base du langage Java et de la boîte à outils Android, et doit pouvoir s'exécuter sur des dispositifs très variés que sont les tablettes et les « smartphones ». Une application interactive pour le Web est développée en utilisant les langages tels que JavaScript et HTML, mais aussi Java ou C# selon la plate-forme d'exécution logicielle retenue (Java ou .NET), et doit pouvoir s'exécuter dans une infrastructure client/serveur compatible avec l'Internet.

Cette grande variabilité induit une difficulté quant aux choix des composantes de développement à arrêter. Une fois ces choix arrêtés, le développement est, d'une part, contraint par les capacités offertes par les composantes retenues et, d'autre part, non nécessairement pérenne. Le premier point ne facilite guère le développement de SI allant au-delà des systèmes habituels (utilisation classique du tandem clavier+souris) [10]. Le second point implique une refonte intégrale du système dès lors que l'une voire plusieurs des composantes utilisées deviennent obsolètes.

Une solution à la difficulté inhérente au développement de la partie interactive des SI est de focaliser l'effort de réalisation sur l'élaboration de modèles du SI plutôt que sur sa programmation. La modélisation devient ainsi la partie non contrainte aux choix technologiques des composantes retenues et reste pérenne car indépendant de ces composantes. Cependant, si de nombreux méta-modèles ont été proposés dans la littérature dans ce sens, force est de constater qu'ils ne sont pas nécessairement mis en œuvre par les langages de programmation ou/et les boîtes à outils. Ainsi, les modèles restent trop souvent des modèles documentaires formalisant l'expression du besoin et servant de point d'entrée à la programmation. Dans cette approche usuelle, la cohabitation des modèles et des programmes qui les implémentent n'est pas nécessairement simple : les modèles se cantonnent à la couche *spécification* du SI, tandis que les programmes restent dans la couche *implémentation*.

Approche adoptée

Les travaux présentés dans ce mémoire visent à montrer comment il est possible de travailler à un niveau *implémentation* en se focalisant malgré tout sur le niveau *spécification*. Ceci est rendu possible en définissant un méta-modèle des SI sur la base des méta-modèles déjà existants (tels que MVC [75], PAC [52] et l'interaction instrumentale [8]), et en outillant ce méta-modèle de manière à ce que les modèles instances de ce méta-modèle puissent être décrits dans un langage de programmation simple et basé sur des usages éprouvés.

Les différents modèles, de part leur capacité à s'exécuter directement sur les plates-formes matérielles et logicielles cibles, ne sont plus de simples spécifications de programme ; de plus, ils préservent une certaine indépendance quant aux plates-formes ciblées ce qui en garantit une meilleure réutilisabilité et plus grande pérennité. Chaque modèle adressant une préoccupation particulière du SI, l'application interactive peut être vue comme une *composition* de modèles [50, 49]. De tels modèles sont alors liés à des disciplines différentes : modélisation du métier, « design » graphique et ergonomie de l'interaction.

Les modèles abordés dans les travaux présentés dans ce mémoire sont donc à considérer comme situés « juste au-dessus » des boîtes à outils et des langages de programmation génériques associés (GPL). Sont donc exclus les modèles de plus haut niveau que sont, par exemple, les modèles de tâche des utilisateurs.

Structure du mémoire

Ce mémoire est structuré en trois chapitres. Le chapitre 1 présente la problématique de la conception des SI sous la forme d'un l'état de l'art portant, d'une part, sur l'évolution des SI et, d'autre part, sur l'usage des modèles pour leur développement. Le chapitre 2 présente l'essentiel de la contribution autour de cette problématique, depuis ma thèse sur le modèle DPI [13] jusqu'à la synthèse de mes travaux ultérieurs [24, 25, 29, 28]. Le chapitre 3 clôt le mémoire en concluant sur la contribution et en précisant les perspectives des travaux réalisés.

Chapitre 1

État de l'art

Ce chapitre présente l'état de l'art autour de la problématique de la conception des SI. La section 1.1 présente l'évolution chronologique des composantes jugées comme essentielles pour le développement des SI. La section 1.2 présente quant à elle l'état de l'art des approches centrées sur les modèles pour l'ingénierie des SI, lesquelles fournissent différentes solutions au problème de la diversité et versatilité des composantes identifiées dans la section 1.1.

1.1 Évolution des composantes de développement des SI

Cette section présente une synthèse de l'évolution chronologique des nombreuses composantes entrant en jeu dans le développement des SI, dans l'objectif de cerner au mieux la complexité intrinsèque d'un tel développement. La section 1.1.1 présente la diversité des plates-formes matérielles et logicielles sur lesquelles les SI s'exécutent. La section 1.1.2 présente une vue d'ensemble des boîtes à outils offrant un accès aux plates-formes précédentes et permettant la programmation des SI. La section 1.1.3 montre l'évolution des langages de programmation utilisés conjointement à ces boîtes à outils. La section 1.1.4 conclut sur la forte variabilité des différentes composantes des SI et introduit le concept de modèle comme une solution possible à la gestion de la complexité induite par cette variabilité.

1.1.1 Les plates-formes matérielles et logicielles

Une plate-forme est un ensemble de composantes permettant l'exécution d'applications, ces composantes pouvant être matérielles ou logicielles. Il est ainsi possible de décomposer une plate-forme en une *plate-forme matérielle* regroupant les composantes matérielles, et en une *plate-forme logicielle* regroupant les composantes logicielles. Les composantes matérielles et logicielles des plates-formes sont cependant étroitement liées si bien qu'il est difficile de les considérer séparément. Cette section présente dans un premier temps l'évolution des systèmes (composantes matérielles) et de leurs interfaces graphiques (composantes logicielles) puis, dans un second temps, l'évolution des principales techniques d'interaction (composantes logicielles) fondées sur l'usage de dispositifs d'entrée/sortie (composantes matérielles).

Les systèmes et leurs interfaces graphiques

La figure 1.1.1 présente une frise chronologique des différents systèmes matériels pourvus d'une interface graphique et jugés les plus significatifs.

Une première lecture de la frise illustre l'évolution de l'ordinateur personnel (PC) de 1973 à nos jours. Amorcée en 1973 par le système « Alto », suivi en 1978 de l'Apple II et dans la foulée, en

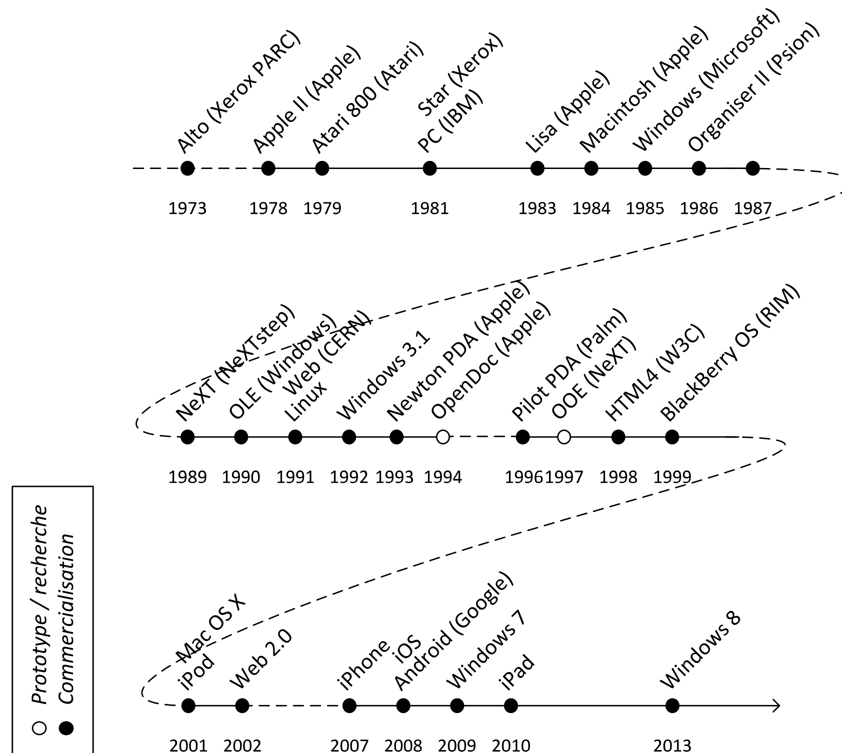

Figure 1.1.1 – Une chronologie des interfaces graphiques

1979, de l'Atari 800, cette évolution amène en 1981 au premier système muni d'un « bureau », le Xerox Star [105, 70, 47]. Apple popularise les principes du Star en 1983 par le Lisa, et surtout par le Macintosh en 1984. L'évolution majeure de son système d'exploitation s'opère en 2001 avec la sortie de Mac OS X. La même année que le Star, IBM sort le premier PC, lequel était dépourvu d'interface graphique. Microsoft propose en 1985 la première version graphique mais rudimentaire de son système d'exploitation Windows, lequel évoluera nettement en 1992 avec la version 3.1. Beaucoup d'autres versions voient le jour jusqu'à Windows 7 en 2009. Enfin, la famille de systèmes UNIX se développe parallèlement dans les entreprises sur lesquels certains systèmes graphiques voient le jour, tel que le système client/serveur XWindow en 1987. Le système d'exploitation NeXT, créé par Steve Jobs en 1989 sur une base UNIX, est l'ancêtre de l'actuel Mac OS. La version libre Linux d'UNIX est diffusée en 1991 et de très nombreux systèmes graphiques sont bâtis sur ce noyau.

Une seconde lecture de la frise chronologique qualifie l'évolution qui a mené aux nouveaux dispositifs mobiles. Si la première lecture reste centrée sur le Xerox Star, cette seconde lecture prend son point d'appui sur les systèmes de type « Personal Digital Assistant » (PDA). Le premier organisateur personnel est créé en 1986 par Psion puis, sept années après apparait le Newton PDA créé par Apple. Si le Newton est un échec commercial, le Pilot de Palm est un succès qui popularise en 1996 les PDA. Le Web, inventé en 1991 par les chercheurs physiciens du CERN, se consolide très fortement avec l'apparition de la version 4 du langage HTML, standardisée en 1998. Les téléphones mobiles BlackBerry envahissent dès 1999 les entreprises grâce à des fonctionnalités de messagerie avancées. Steve Jobs, en inventant l'iPod en 2001, popularise les systèmes développés par Apple pour en faire jusqu'à des objets sociétaux. Les usages du Web deviennent plus sociaux avec le Web 2.0 de 2002, ce qui va contribuer grandement à l'essor des dispositifs mobiles. S'en suivent l'iPhone en 2007 et l'iPad en 2010. Si Apple unifie de manière propriétaire ses systèmes mobiles autour de l'iOS en 2008, Google lance l'offensive avec son système d'exploitation Android sous l'angle du logiciel libre. Microsoft tente de combler son retard dans le marché des téléphones et

tablettes en proposant une approche tentant d'unifier les deux mondes, celui du PC et celui des dispositifs mobiles, rapprochant alors les deux lectures de notre frise chronologique.

Une troisième et dernière lecture qualifie l'évolution des systèmes permettant la visualisation et l'édition de documents composites. Cette évolution trouve sa racine dans l'approche des systèmes centrés sur les documents initiée en 1990 par Microsoft avec OLE [44]. OLE reste un format propriétaire relativement complexe à mettre en œuvre, si bien que seules les applications Microsoft offrent les fonctionnalités OLE. L'approche a été reprise par Apple en 1994 avec OpenDoc [2]. Bien que ce système ne fût pas lié à un système d'exploitation particulier, il est resté trop contraignant par rapport aux habitudes de développement des applications et ne permettait pas l'utilisation de formats de stockage standard. Le système OOE propose quant à lui une approche simple en utilisant l'affichage basé sur le standard PostScript proposé en natif par le système NeXTstep [3]. Les formats tels que HTML, SVG et MathML sont des *standards* aujourd'hui largement utilisés et préférés aux solutions propriétaires pour construire des documents composites, notamment pour le Web.

Les techniques d'interaction

La frise chronologique, présentée figure 1.1.2, synthétise les principaux périphériques et techniques d'interaction qui définissent de manière appariée les modalités d'interaction entre utilisateurs et systèmes.

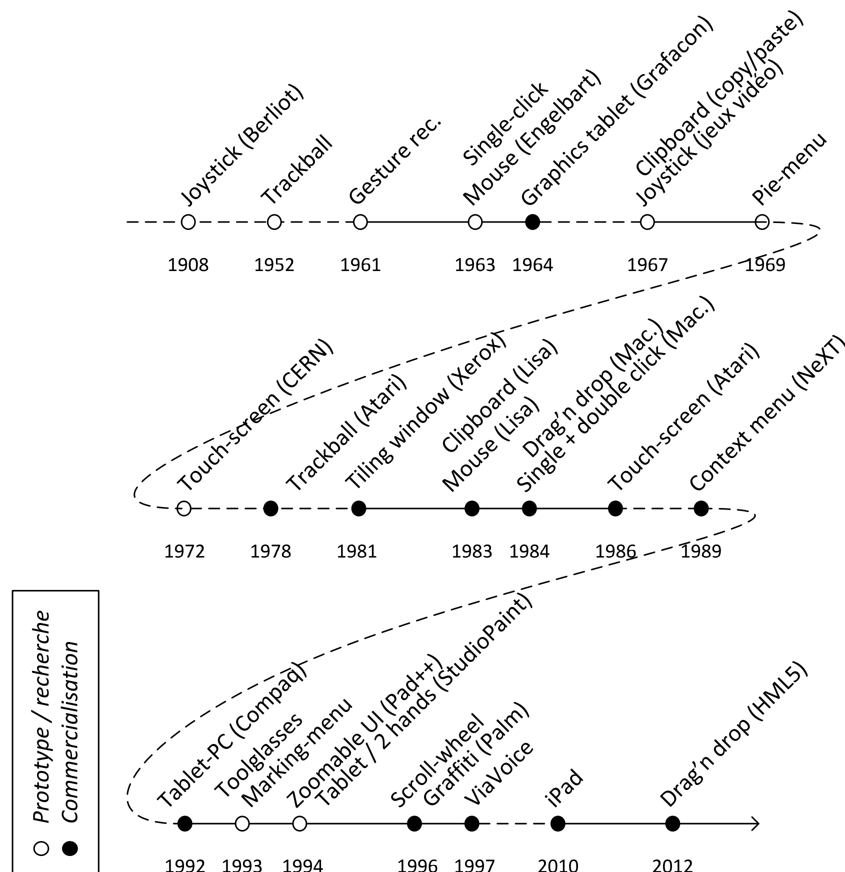

FIGURE 1.1.2 – Une chronologie des interactions

Les périphériques d'entrée des interfaces graphiques sont des dispositifs de pointage tels que le joystick (inventé en 1908 au début de l'aviation et utilisé dès 1957 par les jeux vidéo), le « track-

ball » (premier prototype en 1952, puis commercialisé par Atari en 1978), la souris (inventée par Engelbart en 1963, et commercialisée par Apple en 1983 avec le Lisa), et enfin la tablette graphique (la Grafacon en 1964). La combinaison d'un écran et d'une tablette graphique a donné naissance aux « tablet-PC » (Compaq en 1992). Les écrans tactiles (« touch-screen ») sont apparus avant les « tablet-PC » (premier prototype développé en 1972 au CERN et première commercialisation en 1986 par Atari) et sont devenus aujourd'hui très populaires avec l'avènement des « smartphones » et des tablettes (iPad en 2010).

Chaque périphérique définit un jeu d'interactions élémentaires permettant aux utilisateurs d'interagir avec les systèmes. Le clic simple est naturellement apparu avec la souris (1963), pour évoluer vers le double clic (sur le Macintosh en 1984). Si le jeu d'interaction reste généralement modeste, son utilisation conjointe avec des techniques d'affichage élargit le spectre des possibilités. Par exemple, le glisser-déposer (sur le Macintosh en 1984) permet d'effectuer un couper-coller (inventé en 1967) en un unique geste. L'affichage d'un menu devient contextuel avec le clic droit (sur le NeXT en 1989). Les menus circulaires hiérarchiques (« pie menu » en 1969) permettent la sélection rapide d'un item en dessinant rapidement la trace liée à la sélection de l'item (« marking menu » en 1993), sans avoir nécessairement besoin d'attendre l'affichage des menus et sous-menus [76, 45].

La modification d'un périphérique standard définit un nouveau périphérique qui pourra être facilement adopté car il ne remet pas en cause son usage précédent mais, au contraire, l'augmente. Par exemple, l'ajout de la molette aux souris (la souris « scroll-wheel » de Microsoft en 1996) permet un défilement direct du contenu d'une fenêtre ou l'application d'un facteur de zoom. Les «trackballs» et souris sensitifs utilisent l'information de présence de la main pour augmenter l'interaction [68]. De même, le retour de force augmente le feedback donnant une meilleure sensation (pour les jeux notamment), voire une précision accrue (pour les opérations chirurgicales par exemple) [82].

D'autres modalités sont venues rapidement compléter le panel des interactions usuelles. Les gestes capturés par un dispositif de pointage peuvent être reconnus pour être transformés en actions [98] ou, s'ils correspondent à une écriture manuscrite, en chaînes de caractères (dictée vocal « ViaVoice » d'IBM en 1997). L'interaction bimanuelle élargit les possibilités d'interaction avec les dispositifs de pointage existants. Par exemple, les « toolglasses » (1993) mettent en jeu l'interaction bimanuelle « click-through » qui permet de pointer une cible et de sélectionner un paramètre de l'action en un unique geste bimanuel [32, 33]. L'application StudioPaint illustre comment deux dispositifs de pointage peuvent être utilisés sur un même écran interactif par des graphistes pour reproduire l'utilisation conjointe de gabarits et de stylos [77].

Les différentes techniques d'interaction précédentes montrent que l'affichage et l'interaction sont couplés et ne peuvent en conséquence pas être considérés séparément. Certaines techniques d'affichage ne mettent cependant pas en jeu de nouveaux périphériques. Les fenêtres superposables n'ont été mises en œuvre qu'après l'invention de la souris et elles ont vu certaines évolutions ultérieures, comme les fenêtres empilées de manière irrégulière [9] ou utilisant des onglets.

1.1.2 Les boîtes à outils

La frise chronologique de la figure 1.1.3 retrace l'évolution des principales boîtes à outils (BO) dédiées au développement des IHM graphiques. N'y sont représentées que celles connues dans le monde des développeurs. D'autres boîtes à outils issues du monde de la recherche seront présentées de manière plus spécifique dans les sections 2.4 et 2.5.

Trois classes de BO se dégagent de cette frise. Les premières boîtes à outils ont été élaborées pour simplifier l'usage des Systèmes d'Exploitation (SE) à interface graphique. En ce sens, ces boîtes à outils sont souvent qualifiées d'Application Program Interface (API) puisque chacune définit une couche logicielle entre le SE et l'application interactive. L'environnement MacApp dès 1985, puis Cocoa en 1999, permettent le développement de SI sur MacOS. Les applications sous Windows ont longtemps été bâties depuis 1992 en utilisant la bibliothèque MFC qui masquait la complexité du « Software Development Kit » (SDK) fourni avec Windows. L'API XWindow, mise au point pour les SE de type UNIX en 1987, définit les structures et fonctions permettant la gestion de

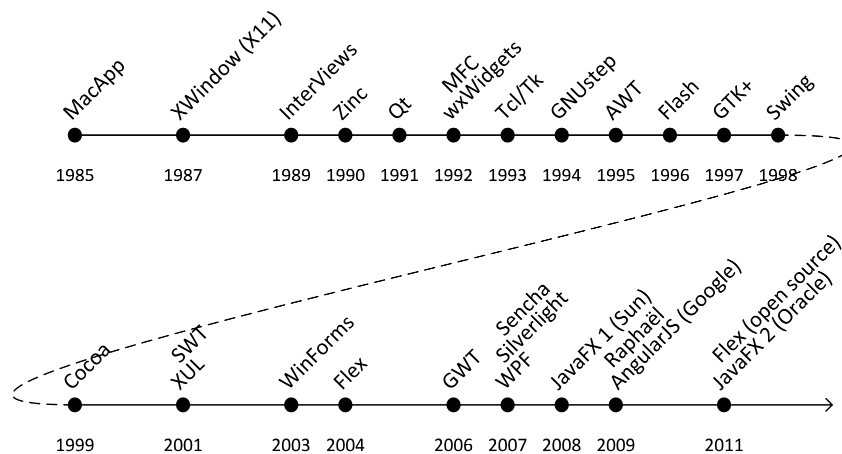

FIGURE 1.1.3 – Une chronologie des boîtes à outils IHM

l'affichage graphique et des périphériques d'entrée dans l'environnement client-serveur propre à UNIX. A cette API ont succédé différentes boîtes à outils visant à masquer sa complexité, par exemple Interviews en 1989, wxWidgets en 1992, GNUstep en 1994, et GTK+ en 1997. Ces boîtes à outils ne permettent pas le développement d'applications pouvant s'exécuter indifféremment sur n'importe quel SE.

De cette limitation ont vu le jour différentes bibliothèques indépendantes du SE formant la seconde classe des boîtes à outils. Le précurseur utilisé dans le monde industriel est Zinc sorti en 1990. Le fait que le langage Java soit indépendant du SE a permis la création de diverses bibliothèques telles que AWT en 1995, Swing en 1998 et SWT en 2001. La bibliothèque Qt, initialement créée en 1991 pour le développement d'applications Linux, a été portée pour Windows et MacOS quelques années après, illustrant là encore l'intérêt des développements indépendants du SE. Enfin, XUL est une boîte à outils utilisant le langage XML pour décrire l'apparence des interfaces graphiques et le langage JavaScript pour décrire leur comportement. Ces deux langages n'étant pas associés à un SE spécifique, les implémentations de XUL peuvent s'exécuter sur différents SE.

Ce dernier exemple a probablement inspiré la dernière et troisième classe de BO qui est apparue avec le développement des applications Internet riches (RIA). Dans un bref laps de temps sont apparues de nombreuses boîtes à outils RIA, telles que Flash en 1996 devenue Flex en 2004, JavaFX 1 en 2008 réécrite intégralement pour définir la version 2 en 2011, et GWT lancée par Google en 2006. De son côté, Microsoft a développé en 2007 la bibliothèque WPF dédiée à la plate-forme .NET et Silverlight pour plate-forme du Web. Le fait que la version 5 d'HTML apparaisse en 2012 a redessiné le paysage des boîtes à outils RIA. Adobe abandonne Flex qui passe alors dans la communauté du logiciel libre, et Microsoft ne propose plus d'évolution à Silverlight. Cet ultime constat laisse la porte ouverte à un nombre significatif de bibliothèques JavaScript qui tirent profit des capacités d'HTML5. La bibliothèque Raphaël simplifie l'usage de SVG, et la bibliothèque AngularJS permet une gestion complète de l'interactivité des pages HTML.

1.1.3 Les langages

La figure 1.1.4 donne une chronologie des principaux langages utilisés pour développer des interfaces graphiques.

Tout comme pour les boîtes à outils, une première classe de langages regroupe les langages dont l'usage est assez fortement lié aux SE. Le Pascal a été conçu en 1970 pour l'enseignement de la programmation, et fut notamment utilisé pour les développements sur MacOS. Le C, dont la première version ANSI a été créée en 1973, est devenu le standard des API fournies avec les SE UNIX, Windows, puis MacOS. Ce langage reste toujours largement utilisé pour le développement

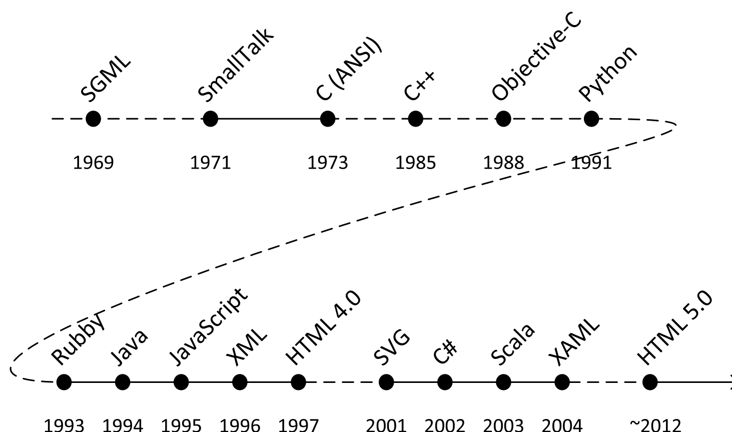

FIGURE 1.1.4 – Une chronologie des langages

d'IHM, sous Linux notamment. Le C++ a suivi en 1985, ajoutant une couche objet au langage C tout en restant entièrement compatible avec les bibliothèques écrites en C. Il a facilité l'usage des API C en y ajoutant une couche d'abstraction orientée objet. L'Objective-C est un langage objet spécifiquement créé en 1988 pour le NeXT de telle sorte que le SDK de ce SE soit directement accessible depuis un langage de haut niveau d'abstraction.

Une seconde classe regroupe les langages indépendants des SE. Les langages Python (1991) et Tcl/Tk (1993) sont des langages interprétés. Les langages SmallTalk (1971), Java (1994) puis C# (2002), utilisent un code intermédiaire qui peut s'exécuter sur un processeur virtuel (SmallTalk et Java) ou être compilé à la volée pour un processeur donné (C#).

Une troisième classe de langages concernent les langages qui offrent des constructions permettant d'étendre le langage initial en un langage dérivé. Une telle capacité autorise la création de nouveaux langages dédiés à des domaines ciblés : on parle alors de « Domain Specific Language » (DSL) interne. C'est par exemple le cas avec Ruby créé en 1993, puis Scala créé en 2003.

La quatrième et dernière classe concerne les langages du Web. Cette classe se décompose en deux sous-classes : les langages de représentation de données et les langages de programmation. Initié par IBM en 1969, SGML permet de définir des formats de documents textes ainsi que leur grammaire. Simplifié sous la forme du langage XML en 1996, d'autres dialectes de type XML vont voir le jour et être popularisés par le Web : HTML 4 en 1997, SVG en 2001, XAML en 2004 et HTML 5 en 2012. Le langage de programmation JavaScript, créé en 1995, représente aujourd'hui le langage incontournable pour rendre interactive les applications Web.

1.1.4 Conclusion

S'il y a une grande variété et une forte évolution des systèmes, il n'y a eu que peu d'évolution de leurs interfaces graphiques [10], du moins jusqu'en 2007. Entre 1984 et 2004, la vitesse des microprocesseurs a évolué d'un facteur 3 000, la capacité mémoire d'un facteur 2 000, la capacité de stockage d'un facteur 200 000, et la résolution écran d'un facteur 10. Cependant, les périphériques d'entrée sont restés cantonnés au clavier et à la souris, et les techniques d'interaction n'ont en conséquence guère évolué. Cet état de fait a cependant été remis en cause en 2007 de par l'arrivée de nouveaux dispositifs mobiles qui proposent de nouveaux périphériques d'entrée conjointement à de nouvelles techniques d'interaction. Force est de constater qu'il y a aujourd'hui une grande variété des plates-formes sur lesquelles les SI s'exécutent, aussi bien du point de vue matériel que du point de vue logiciel.

Une telle variété des plates-formes se retrouve naturellement dans la multitude des BO de développement des SI et des langages de programmation associés. Mais si cette richesse liée à la multitude

peut sembler avantageuse, comment garantir la pérennité des solutions ouvertes de type « open source » ou des solutions propriétaires des éditeurs ? La réponse à cette question tient probablement dans l'utilisation de standards : l'engouement récent pour le standard HTML5 et JavaScript illustre bien cette tendance. Cependant, si la plate-forme du Web peut être raisonnablement considérée comme un standard universel, il n'en demeure pas moins que les plates-formes matérielles sont toujours plus nombreuses et que ce standard universel reste encore insuffisant pour exprimer les différents aspects complémentaires des SI et de leurs plates-formes.

Une solution à cette problématique de la diversité consiste à utiliser les modèles de manière à prendre de la hauteur par rapport à une programmation directe qui reste « bas niveau ». Il s'agit ainsi de voir un SI comme en ensemble de modèles, lesquels ont la capacité de s'exécuter de manière coopérative sur la plate-forme logicielle et matérielle.

1.2 Modèles et ingénierie des IHM

Le domaine de l'ingénierie des IHM a généré depuis de nombreuses années différents modèles conceptuels qui visent à faciliter le développement des SI. Ces modèles décomposent les SI en différentes composantes logicielles, chacune traitant un aspect particulier du système. Les principaux modèles conceptuels dédiés aux SI sont présentés dans la section 1.2.1.

L'outillage de ces modèles, nécessaire à leur utilisation effective, est à chaque fois réalisé de manière *ad hoc*, ce qui implique un effort de développement important. Les approches issues de l'Ingénierie Dirigées par les Modèles (IDM), si elles ne sont pas spécifiquement dédiées au domaine des IHM, peuvent néanmoins être utilisées pour la réalisation d'un tel outillage. La section 1.2.2 aborde l'usage de l'IDM dans le contexte de l'ingénierie des IHM.

La conclusion présentée de la section 1.2.3 précise le cadre de l'orientation « modèle » prise par les travaux présentés dans ce mémoire, cadre associant différents modèles conceptuels de SI avec des approches issues de l'IDM.

1.2.1 Modèles conceptuels

Si les nombreuses BO permettent l'implémentation d'IHM, elles ne définissent généralement pas un cadre strict précisant leur utilisation dans le contexte du développement *complet* des SI, *i.e.* intégrant les différentes composantes d'un SI. Elles se limitent pour la plupart à la partie strictement IHM. Il est ainsi important que le développement des SI soit cadré par une approche stricte fondée sur la définition d'un modèle conceptuel qui définit explicitement les éléments du modèle ainsi que leur relation. Cette section présente les modèles conceptuels les plus connus, lesquels décomposent un SI en plusieurs composantes logicielles. Les modèles mettant l'accent sur la séparation modèle / vue sont d'abord présentés : MVC [75], MVC2 [101], PAC [52] et Arch [6]. Ceux mettant l'accent sur la séparation interaction / action sont ensuite précisés : les interacteurs [86, 85] et l'interaction instrumentale [7, 8]. Enfin, certains concepts de la POO peuvent être transposés à la programmation des interfaces utilisateur (OOUI) puisque ces interfaces présentent les objets aux utilisateurs finaux [51]. Tous ces modèles conceptuels restent indépendants des BO de développement d'IHM.

Séparation modèle / vue

Le modèle conceptuel Modèle - Vue - Contrôleur (MVC) est un modèle très répandu qui décompose les systèmes interactifs en trois éléments essentiels [75]. La figure 1.2.1 représente ces trois éléments sous la forme de trois plans orthogonaux, leurs liens étant représentés par des flèches. Le modèle M définit les *objets du domaine*, la vue V définit les *présentations* faites aux utilisateurs des objets du domaine, et le contrôleur C définit les *interactions* que les utilisateurs peuvent effectuer sur les objets du domaine *via* leurs présentations. Les liens qui unissent ces trois éléments entrent en jeu selon un ordre typique. Une vue s'enregistre en tant qu'observatrice du modèle et accède à l'état du

modèle pour un premier affichage (lien $V \rightarrow M$). Lorsqu'un utilisateur interagit avec le système, le contrôleur modifie le modèle (lien $C \rightarrow M$), lequel notifie alors les vues du changement de l'état du modèle (lien $M \rightarrow V$). Les vues se rafraichissent en fonction des changements détectés sur le modèle (lien $V \rightarrow M$ à nouveau). Certaines interactions utilisateur nécessitent la modification de la vue sans pour autant altérer le modèle associé (par exemple pour indiquer qu'une valeur saisie est incorrecte). Dans ce cas, le contrôleur modifie la vue associée (lien $C \rightarrow V$).

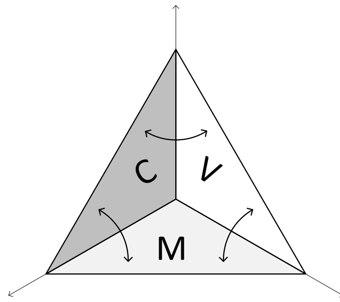

Figure 1.2.1 – MVC

Si le modèle MVC isole clairement chaque partie du système, les liens $V \rightarrow M$ et $M \rightarrow V$ deviennent difficiles à implémenter dès lors que les modèles et leurs vues deviennent complexes. Pour palier cette insuffisance, le modèle MVC a évolué vers le modèle MVC2 de la figure 1.2.2. Le contrôleur *C2* représente maintenant le lien bidirectionnel $M \leftrightarrow V$ et la vue V représente la paire vue-contrôleur de MVC [101]. Ce nouveau rôle attribué au contrôleur est analogue au contrôleur C du modèle Présentation-Abstraction-Contrôleur (PAC), dans lequel la présentation représente la vue et l'abstraction le modèle [52]. D'autres variantes de MVC ont également été proposées, telles que «Model-View-Presentation» (MVP) [97] et «Model-View-ViewModel » (MVVM) [106], pour lesquelles la variation porte là aussi sur le contrôleur de MVC.

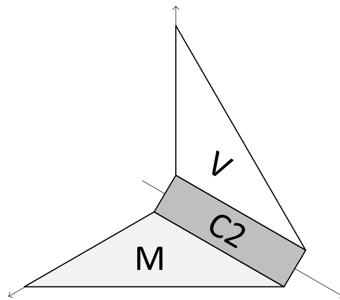

Figure 1.2.2 – MVC2

Les boîtes à outils modernes dédiées au développement des « Rich Internet Applications » (RIA), telles que WPF [99], GWT [56], Flex [74] et JavaFX [60], proposent toutes des mécanismes de liaison de données qui permettent une implémentation simplifiée des contrôleurs *C2*. Ceci illustre l'intérêt de considérer le contrôleur comme un élément de première importance.

Les modèles MV-C/C2/P/VM et PAC se focalisent finalement tous sur la séparation entre les objets du domaine et leurs présentations. Cependant, aucun de ces modèles ne se préoccupe explicitement de la séparation entre les interactions effectuées par les utilisateurs et les actions réalisées sur les objets du domaine.

Séparation interaction / action

Les interacteurs sont des éléments du SI qui séparent les interactions faites par les utilisateurs des actions qui peuvent être effectuées sur les objets du domaine, comme l'illustre la figure 1.2.3. Un

interacteur est ainsi un élément qui transforme les interactions utilisateur en actions, ces dernières modifiant alors les présentations ou/et les objets du domaine [85, 86].

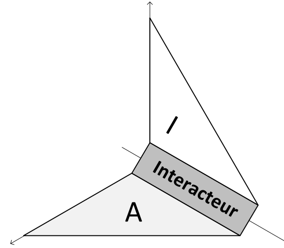

FIGURE 1.2.3 – Interacteur

La figure 1.2.4 présente l'interacteur « Menu » qui permet la sélection d'un élément parmi les n éléments proposés par le menu. Cette sélection fournit un retour visuel qui dépend de la nature de l'élément ciblé et non de l'interacteur lui-même. Ce retour visuel peut être dans un premier temps un retour intermédiaire : rectangle en "vidéo-inverse" (1) et (5), contour rectangulaire (3), enfoncement d'un bouton (disparition de l'ombre portée) (4), mise en italique de l'intitulé lui-même (6). Le retour d'information final peut compléter la perception de la sélection : contour rectangulaire (1), déplacement de l'intitulé à droite ou à gauche (2), un signe plus (3), rectangle en "vidéo-inverse" (4), mise en gras de l'intitulé lui-même (6).

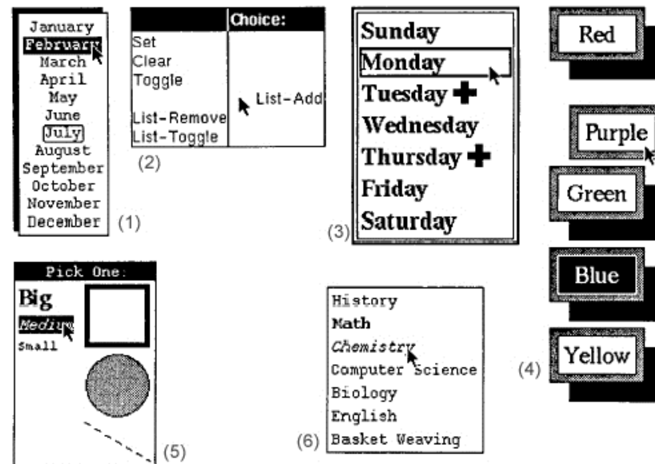

FIGURE 1.2.4 – Interacteur « Menu » – extrait de [86]

Les interacteurs permettent d'aborder l'interaction principalement d'un point de vue du *développeur*. La motivation première de l'interaction instrumentale est de proposer un modèle d'interaction centré sur l'*utilisateur* [7, 8]. Un instrument d'interaction est défini comme étant le *médiateur* entre l'utilisateur et les objets du domaine qu'il souhaite manipuler. Le principe général de fonctionnement d'un instrument est illustré par la figure 1.2.5 :

1. L'utilisateur agit sur l'instrument par le biais d'un ou plusieurs périphériques d'entrée utilisateur qui capturent les *gestes* (*i.e.* les interactions) de l'utilisateur.
2. Les gestes sont ensuite transformés en *actions* en fonction de la nature de l'instrument. Ces actions sont appliquées à l'objet du domaine ciblé.
3. L'instrument réagit aux gestes de l'utilisateur et lui fournit donc une *réaction*.
4. Il fournit de plus un *retour d'information* (ou feedback) permettant à utilisateur de percevoir l'état de l'instrument et donc de l'action en cours.

5. Quand l'action est appliquée à l'objet ciblé, ce dernier renvoie une *réponse* quand l'action est traitée.

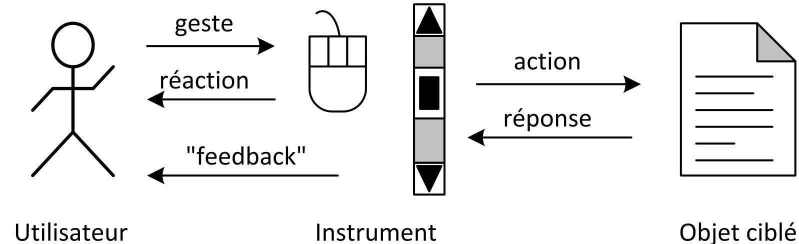

FIGURE 1.2.5 – Principe général d'un instrument – inspiré de [8]

Si les modèles d'interacteur et d'instrument d'interaction séparent clairement les interactions des actions, ils ne se préoccupent pas des relations entre les présentations ou/et les objets du domaine.

Approche orientée objet

Les premiers systèmes interactifs ont mis en avant la manipulation d'*objets* au travers d'une interface principalement graphique. Ceci a eu pour conséquence l'apparition du terme objet tant du point de vue de l'interface homme-machine que du point de vue de la programmation. Le terme «Object Oriented User Interface » (OOUI) a été défini par IBM dans les années 1990 [51]. L'approche OOUI précise les bénéfices que l'approche orientée objet peut apporter aux interfaces utilisateurs en indiquant comment certains concepts définis par l'OOP peuvent également s'appliquer aux interfaces utilisateurs :

Objet – Un objet est une entité dont l'*utilisateur* a besoin pour réaliser une tâche particulière.

Encapsulation – L'utilisateur appréhende les différents objets sans pour autant savoir comment les comportements de ces objets sont effectivement codés.

Composition – Les objets des interfaces graphiques sont naturellement des objets composés d'autres objets.

Classe – L'utilisateur perçoit de manière plus ou moins consciente que les différents objets qu'il manipule ont des comportements communs.

Héritage – Certains comportements se retrouvent d'une classe à l'autre. L'avantage pour l'utilisateur est considérable : il peut *transférer* ses connaissances acquises pour une classe aux différentes classes dérivées.

Il semble ainsi pertinent de réutiliser une approche OO pour l'Ingénierie des IHM (IIHM), d'une manière complémentaire aux modèles de type MVC. Les modèles informatiques étant typiquement des modèles OO, il devient pertinent de s'interroger sur l'usage des concepts, techniques et outils de l'Ingénierie Dirigée par les Modèles (IDM) dans le contexte de l'IIHM.

1.2.2 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) est un paradigme de l'ingénierie des logiciels, des systèmes et des données. Son idée centrale, résumée par le terme « tout est modèle » lui-même inspiré de l'expression « tout est objet », est de considérer le cycle de développement d'un logiciel comme une chaîne de transformations de modèles.

Cette section présente les différents outils et approches sous-jacentes issus de l'IDM et utilisé dans le contexte de l'IIHM. Les outils de type « Model-Based User Interface Development Environment » (MB-UIDE), apparus avant le standard « Model Driven Approach » (MDA) [91], sont tout d'abord présentés. Les outils fondés sur des standards et apparus après MDA sont ensuite mis en avant. Les apports et limites de ces outils et approches sont finalement discutés.

Outils de type MB-UIDE

Bien que développés avant l'apparition du standard MDA, les MB-UIDE présentés dans cette section respectent les principes généraux de l'IDM. En effet, chacun d'entre eux peut être considéré comme un espace technique, au sens de [31], regroupant différentes technologies dédiées à la modélisation et à la création de SI, au même titre que MDA est un espace technique fondé sur les langages MOF et UML.

L'évolution des MB-UIDE, dont les principaux sont amplement détaillés par [110] et [103], s'est effectuée en plusieurs étapes successives. Les premiers environnements, tels que COUSIN [67], UIDE [61], Mickey [89], Jade [121] et Humanoid [111], utilisaient un langage de description d'IHM pour en générer le code. Afin d'améliorer la qualité des IHM générées, de nouveaux MB-UIDE utilisant des modèles complémentaires des modèles de description d'IHM sont apparus. Parmi ces modèles, on retrouve tout d'abord le modèle de tâches, décrivant les différentes tâches qu'un utilisateur peut réaliser sur un SI. Les environnements comme DIANE+ [113], TADEUS [59], EXPOSE [65], AME [81] et JANUS [4] sont les principaux exemples mettant en œuvre les modèles de tâches. Les modèles de données (souvent appelés modèles du domaine), de présentation et d'utilisateurs sont également utilisés en plus du modèle de tâches, respectivement, pour représenter les données manipulées par les SI, définir la structure abstraite de l'IHM et spécifier les caractéristiques des utilisateurs. TRIDENT [116], FUSE [79], ADEPT [120], MECANO [94], MOBI-D [95] et TEALLACH [66] sont des exemples de tels MB-UIDE.

Un concept important qu'ont apporté certains de ces environnements est la notion d'interface abstraite et d'interface concrète. La figure 1.2.6 résume l'architecture de ces MB-UIDE dans laquelle le développement d'un SI se divise en quatre principales étapes : la définition des modèles de tâches et des données (étape ①) ; la spécification de l'interface abstraite (étape ②) ; celle de l'interface concrète (étape ③) ; la génération du code (étape ④). Les concepts définis par MDA, notamment l'indépendance de la plate-forme et la représentation d'un système sous la forme de modèles [91], sont déjà présents dans ces MB-UIDE.

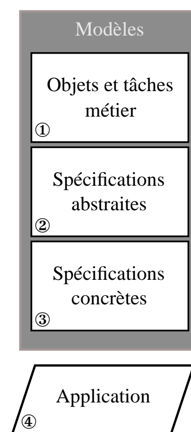

FIGURE 1.2.6 – Développement d'un SI en différentes étapes - adaptée de [110]

Les MB-UIDE présentés précédemment se fondent cependant tous sur un formalisme différent pour représenter ces modèles. Les MB-UIDE fondés sur UML ont, quant à eux, l'avantage d'utiliser un langage de modélisation largement adopté et donnant la possibilité, au travers de ses différents diagrammes, de décrire une partie des SI. Parmi ces MB-UIDE, UMLi [104] et l'extension d'UML pour les interfaces WIMP (« *Window, Icon, Menu, Pointer* ») [1] en sont les deux principaux exemples. Ils utilisent et étendent certains diagrammes UML, comme les diagrammes de classes, d'activité et de cas d'utilisation, pour décrire différentes parties d'un SI.

Outils fondés sur des standards

Les MB-UIDE post-MDA ont l'avantage de pouvoir s'appuyer sur des concepts et des outils communs et pré-existant pour modéliser un SI. De plus, ces MB-UIDE utilisent avantageusement la transformation de modèles pour passer d'un niveau d'abstraction à un autre.

L'un des environnements IDM les plus connus est certainement le « *Graphical Modeling Framework* » (GMF) développé par Eclipse [83]. GMF ne se place pas dans l'espace technique de MDA mais définit son propre espace technique, appelé « *Eclipse Modeling Framework* » (EMF) dans lequel le langage Ecore est le méta-métamodèle. GMF est dédié à la génération d'éditeurs de diagrammes à partir d'un ensemble de modèles décrivant les données sources et différentes parties de l'interface (palettes d'outils, présentations, *etc.*). Si GMF a pour principal avantage de générer intégralement le code des éditeurs de diagrammes, ces derniers ont des interfaces graphiques stéréotypées desquelles il est difficile de s'éloigner.

Une majorité des MB-UIDE post-MDA s'intéresse à de nouveaux problèmes issus de l'évolution des dispositifs électroniques (PDA, téléphone portable, *etc.*) et aux nouvelles façons d'interagir avec ces derniers (réalité mixte, interaction multimodale, *etc.*). S'ils gèrent tous les interactions multimodales, certains, comme OpenInterface [100] et ASUR [64], sont orientés pour la conception d'interactions et d'interfaces post-WIMP ; d'autres, comme CAMELEON [46], UsiXML [114] et TERESA [93], tendent à répondre au problème de l'adaptation des IHM au contexte.

Apports et limites

Un point commun à tous les MB-UIDE est l'utilisation d'un processus de conception, représenté par la figure 1.2.7, divisé en quatre niveaux :

1. Les *modèles de tâches et des données* définissent les tâches que les utilisateurs peuvent réaliser sur les données sources.
2. L'*interface abstraite* fournit une description indépendante d'une plate-forme et de toute organisation logique des éléments de l'interface.
3. L'*interface concrète*, contrairement à celle abstraite, décrit l'aspect graphique de l'interface en fonction d'une plate-forme donnée ainsi que les interacteurs utilisés.
4. L'*interface finale* correspond au code source du SI.

Une telle décomposition a pour avantages d'enrichir progressivement la spécification d'un SI et de pouvoir réutiliser les modèles définis. Une interface abstraite sert, par exemple, à la création de plusieurs interfaces concrètes, dans le cadre d'un SI multi-plateformes. Cependant, une des limites majeures à la modélisation abstraite est le pouvoir d'expression des MB-UIDE : les SI générés se restreignent généralement à un domaine d'étude [84]. Une solution à ce problème consiste à ajouter des modèles afin de décrire plus en détails les SI. Ce principe introduit un nouvel inconvénient, l'augmentation de la complexité du processus de spécification, qui réduit du même coup l'attractivité et la faisabilité de la méthode [115]. D'autres limites sont également présentes, comme la maintenance des modèles qui peut s'avérer compliquée lorsqu'ils sont liés entre eux.

Le tandem IDM-IHM ne se limite pas à la conception de SI mais concerne également la composition et l'adaptation d'IHM. La composition d'IHM consiste à fusionner différentes IHM en une unique [78, 69, 55]. L'adaptation d'IHM consiste à modifier l'IHM de manière dynamique, c'est-à-dire lors de l'exécution du SI, ou statique lors d'un changement de contexte d'usage [107, 43].

1.2.3 Conclusion

Les modèles conceptuels sont indispensables en IIHM. Ils permettent la séparation des préoccupations en modélisant les SI sous différents points de vue, chacun définissant un aspect particulier du SI. Ces points de vue mettent en évidence les éléments clés des SI que sont le modèle métier orienté donnée (D), leur présentation (P) à l'utilisateur, les interactions (I) et les actions (A). Ces

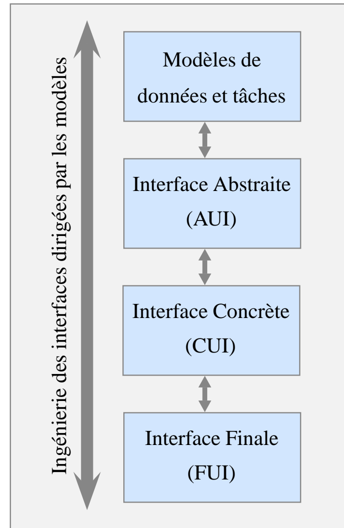

FIGURE 1.2.7 – Ingénierie des interfaces dirigée par les modèles

éléments sont liés entre eux par des mécanismes de liaison de données (lien D ↔P) ou par les interacteurs (lien I↔A).

Il doit être retenu de l'approche de l'IDM uniquement les aspects pertinents dans le cadre de l'IIHM. L'approche à transformations en cascade prônée par MDA est, pour un tel cadre, un échec : il est préférable de considérer les modèles comme directement exécutables (génération de code à un seul niveau) et coopérant entre-eux, plutôt que de faire générer l'application finale par une juxtaposition de modèles et de transformations [53]. La co-exécution d'un modèle de tâche et d'un modèle système (réseau de Pétri par exemple) illustre l'intérêt d'une telle approche : les modèles ne sont pas de simples spécifications et leur mise en correspondance garantit une meilleure intégrité du SI [5].

L'usage des modèles doit permettre de définir un SI avec un certain niveau d'indépendance de la plate-forme d'exécution, sans pour autant considérer les niveaux PIM et PSM définis par MDA. Par contre, un tel usage doit permettre une ré-utilisabilité maximale des modèles. La puissance de l'approche IDM est illustrée par les nombreux outils issus de l'IDM et permettant de construire des SI à manipulation directe, tels que les systèmes auteurs [27].

Les travaux présentés dans ce mémoire visent à *formaliser un modèle conceptuel des SI par un cadre logiciel IDM*. Ce cadre logiciel doit permettre la définition des éléments D, P, I et A d'un SI ainsi que leur mise en relation, dans une approche minimisant l'effort d'implémentation. Plutôt que d'utiliser un langage OO et une boîte à outils associée pour l'implémentation d'un modèle décrit par ailleurs, les modèles sont décrits dans un langage de syntaxe proche des langages OO habituels, et auront la capacité de s'exécuter directement sur différentes plates-formes. L'idée centrale est ainsi de décrire des modèles exécutables plutôt que d'implémenter ces mêmes modèles.

L'application décrite dans un tel langage peut donc être considérée comme une composition de différents modèles provenant de disciplines différentes : par exemple, modélisation du métier pour D, design graphique pour P, ergonomie pour I+A. Cette vision de la conception des SI est importante étant donné le caractère complexe d'une telle conception [50, 49].

L'ensemble de ces travaux ainsi que leur évolution sont présentés dans le chapitre suivant, de la description du modèle conceptuel DPI [20] jusqu'à celle du cadre logiciel Loa [29, 28].

Chapitre 2

Contribution

Ce chapitre présente l'ensemble des travaux menés depuis ma thèse jusqu'à aujourd'hui, travaux qui se concrétisent par la définition et l'outillage d'un cadre logiciel regroupant la presque totalité des réalisations antérieures. Le section 2.1 présente l'évolution chronologique de mon parcours en tant que chercheur en précisant notamment les différentes coopérations et projets qui ont été engagés avec différentes équipes et chercheurs. La section 2.2 complète cette présentation par une synthèse de la contribution en retraçant les réalisations et les publications associées. Les sections suivantes précisent plus en détail chacun des travaux liés à cette contribution : la section 2.3 présente le modèle conceptuel DPI, lequel fournit un cadre conceptuel décomposant en SI en différentes composantes qui coopèrent lors de l'exécution de l'application interactive ; la section 2.4 précise les travaux autour des transformations actives et des correspondances, travaux qui complètent ce qui n'a pas été abordé dans la thèse par la modèle DPI. La section 2.5 clôt ce chapitre en présentant le cadre logiciel Loa, lequel unifie l'ensemble des travaux menés depuis 2000.

2.1 Présentation du parcours

La chronologie de mes activités de recherche est marquée par deux phases significatives : la recherche autour de la thématique « Ingénierie des systèmes interactifs », puis son évolution vers une approche fondée sur certains principes de l'IDM. Le passage de la première phase à la seconde s'est faite par une étape pivot de transition vers l'IDM, correspondant à une coopération étroite avec l'équipe Triskell de l'IRISA. La fin de la seconde phase est marquée par la synthèse des 13 années de recherche qui se concrétisent par la proposition du cadre logiciel Loa et par les perspectives que les travaux autour de Loa ouvrent.

2.1.1 Ingénierie des systèmes interactifs

J'ai réalisé ma thèse entre 2001 et 2004 au sein l'équipe InSitu du PCRI INRIA et LRI, à l'université Paris XI (Orsay), sous la direction de M. Beaudouin-Lafon, professeur d'université et directeur du LRI. Cette thèse, intitulée « Espaces de travail interactifs et collaboratifs : vers un modèle centré sur les documents et les instruments d'interaction » [13], a fait suite au stage de DEA ayant pour titre « Paradigmes et éléments architecturaux d'une boîte à outils post-WIMP » [11]. Le partenariat établi avec l'équipe InSitu et l'équipe GRI de l'ESEO au travers de ce doctorat a permis de développer l'axe « Ingénierie des Systèmes Interactifs » au sein du GRI entre 2004 et 2008. Cet axe visait à définir des techniques et des modèles conceptuels qui permettent aux systèmes interactifs de coopérer dans des environnements distribués, dynamiques et centrés sur les documents. Cette recherche ciblait plus particulièrement la problématique des services Web, ainsi que la définition et l'implémentation de modèles d'interaction et de collaboration au niveau même

des documents structurés. J'ai participé plus spécifiquement au deuxième aspect lié au « document interactif et collaboratif ». Différentes boîtes à outils liées au modèle DPI ont été développées (eXAcT, EventPoints, Domino et DoPIdom) afin d'évaluer la contribution de différents aspects du modèle.

J'ai pris début 2006 la relève du responsable du département Informatique pour assurer la responsabilité du GRI. Les travaux de recherche du groupe abordaient les « Systèmes d'Information » selon trois axes : l'ingénierie des modèles, l'ingénierie des systèmes interactifs, la sécurité et la mobilité. En plus des charges administratives liées à cette fonction (rédaction du rapport d'activité, participation au Comité de Recherche, synthèse devant le Conseil Scientifique, etc.), la tâche principale a consisté à monter des coopérations avec d'autres laboratoires de recherche.

La majorité des doctorants de l'équipe était co-encadrée avec des enseignants-chercheurs du LERIA, laboratoire de recherche en informatique de l'université d'Angers. Conjointement à cette coopération avec le LERIA, une coopération avec le LIUM (laboratoire de recherche en informatique de l'université du Maine) a été entamée courant 2006 pour l'axe « ingénierie des systèmes interactifs ». Cette coopération s'est traduite par le montage d'un projet Région nommé Padifla (Processus d'appropriation des dispositifs de formation en ligne par les acteurs) avec l'Université du Maine (P. Leroux), l'EMN et le CNAM de Nantes. En plus de la coopération avec le LERIA et le LIUM, les deux autres axes du GRI ont entamé leurs propres coopérations dans leurs thématiques associées.

A partir du le 1er décembre 2006, ayant obtenu un financement de la part des collectivités locales (Angers agglomération), j'ai co-encadré avec le LERIA (équipe ICLN) la thèse d'Arnaud Blouin intitulée « Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données » [34]. Cette thèse, soutenue fin novembre 2009, a conduit à plusieurs publications en commun, dont une revue nationale et une conférence internationale de rang A. Elle a donné lieu à une interaction régulière avec le LERIA lors de présentations de résultats.

La réalisation de cette thèse a été l'occasion d'amorcer en 2008 une réflexion sur l'usage de certains concepts et outils de l'IDM pour l'ingénierie des IHM, et d'opérer ainsi une transition vers l'IDM.

2.1.2 Transition vers l'Ingénierie Dirigée par les Modèles

La thèse d'Arnaud Blouin était initialement intitulée « Transformation de documents dans le contexte des systèmes interactifs : une approche basée sur la correspondance entre documents sources et présentations cibles ». L'objectif principal de la thèse était de formaliser le lien entre les documents (D) et leurs présentations (P), en définissant un langage dédié par exemple. Une correspondance se différencie d'une transformation par le fait qu'une correspondance est définie relativement à un schéma source et un schéma cible alors qu'une transformation s'opère entre un document source instance du schéma source et une présentation cible instance du schéma cible. L'approche proposée se situe ainsi à un niveau au dessus des boîtes à outils pour le développement des IHM et des langages associés, à savoir qu'elle se focalise sur les schémas et non sur leurs instances. Ceci nous a conduit à aborder le problème sous l'angle des modèles au sens de l'IDM plutôt que sous l'angle des schémas des documents XML et, par conséquent, à donner un nouveau titre à la thèse : « Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données ». Conjointement à ces travaux de thèse, j'ai travaillé sur les « opérations actives » entrant en jeu dans l'exécution incrémentale des correspondances et la première implémentation de ces opérations a été réalisée sur la plate-forme IDM Kermeta [19]. L'ensemble des travaux a été présenté en 2008 à l'équipe Triskell (aujourd'hui DiverSE) de l'IRISA / INRIA Rennes, équipe à l'origine de Kermeta.

Cette année 2008 a marqué le début d'une coopération forte avec l'équipe Triskell, leader en recherche dans le domaine de l'IDM, sur le thème de l'ingénierie des IHM dans une démarche IDM. Cette période de transition vers l'IDM a été notamment marquée par le fait que le directeur de l'équipe, Jean-Marc Jézéquel, a été membre du jury de la thèse d'Arnaud (2009), Arnaud ayant ensuite fait son post-doctorat dans l'équipe Triskell (2010) pour se retrouver en tant que MDC à

l'INSA de Rennes et faisant sa recherche au sein de l'équipe Triskell (2011). Le second fait marquant de cette transition concerne le montage d'une réponse à l'AAP ANR « ContInt » notamment en relation avec l'équipe IIHM du LIG (Grenoble), équipe déjà impliquée dans l'utilisation de l'IDM pour l'ingénierie des IHM et dont un professeur, Gaëlle Calvary, était également membre du jury de la thèse d'Arnaud. Cette transition a permis de développer à partir de 2009 une approche fondée sur l'IDM en coopération étroite avec l'équipe Triskell.

2.1.3 Une approche fondée sur l'IDM

Pendant toute la période de 2008 à 2013, j'ai beaucoup œuvré dans la poursuite du travail de synthèse des travaux antérieurs sur la base des opérations actives pour proposer finalement le langage Loa. L'ensemble de ces travaux a conduit à une quinzaine de publications, dont une dizaine en commun avec l'équipe Triskell.

En 2011, j'ai intégré l'équipe TRAME de l'ESEO, le GRI devenant alors l'équipe MODESTE. Ce choix a été notamment motivé par la thématique centrale de l'équipe, à savoir la « transformation de modèles », et le côté concret des travaux menés qui implique la réalisation d'outils et la participation à des projets de type ANR. Cette même année, j'ai été jury de la thèse de Mickaël Clavreul intitulée « Model and Metamodel Composition : Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques » et réalisée à l'IRISA au sein de l'équipe Triskell.

En 2012, j'ai réalisé le projet « Outils pour la Conception de Systèmes d'Informations Interactifs » (OXII) suite à un financement MPIA (Maturation de Projets Innovants en Anjou) obtenu en 2011. Cette réalisation a consisté à consolider l'implémentation du langage Loa en tant que DSL interne du langage Scala de manière à pouvoir appliquer les opérations actives sur la plate-forme Java. Elle a également consisté à réaliser un outil graphique permettant d'avoir une vue d'ensemble des éléments manipulés par une application écrite en Loa. Ce travail a été réalisé par Mickaël Clavreul dans le cadre d'un post-doctorat financé par le MPIA. Suite à ce projet, Mickaël a été embauché à l'ESEO et a intégré l'équipe TRAME.

Cette même année, j'ai entamé le co-encadrement de la thèse CIFRE de Mengqiang Yang intitulée « Approche outillée pour la conception d'IHM dédiées à la manipulation de données complexes et contraintes ». Ce co-encadrement s'effectue avec Frédéric Saubion de l'équipe MOA du LERIA dans un intérêt réciproque : Frédéric est intéressé par les aspects IHM permettant de visualiser voire de manipuler des contraintes, et j'étais intéressé par l'adjonction de contraintes sur un modèle métier et l'impact de ces contraintes sur la correspondance entre ce modèle et ses présentations.

En 2013, je suis devenu officiellement collaborateur extérieur de l'IRISA de l'INRIA *via* l'équipe Triskell, aujourd'hui rebaptisée DiverSE (« DIVERsity-centric Software Engineering), dirigée par Benoît Baudry. Ceci est l'aboutissement concret de la coopération forte que j'ai engagée depuis 2008 avec cette équipe.

2.1.4 Projets actuels

L'année 2013 et ce début d'année 2014 est d'abord marqué par la réponse à deux appels à projet ANR, lesquels ont passé la première phase de sélection, la seconde phase étant en cours. Le premier est l'ANR JCJC pilotée par Arnaud Blouin et intitulée « NEXT-CITYGEN - Exploring and Exploiting Smart City Diversity to Build the Next Generation of Intelligent User Interfaces ». Il m'est possible de participer à cette ANR étant donné mon statut de collaborateur extérieur de l'équipe DiverSE, les appels JCJC n'étant valables que pour un unique partenaire (l'IRISA dans ce cas). Le second est l'ANR de type « Projets collaboratifs en partenariat public et privé » regroupant les universités du Maine (Le Mans et Laval), de Valencienne et de Compiègne, ainsi que l'ESSCA, l'ESEO et l'entreprise LogoSapience côté angevin. Ce projet intitulé « VIFORM - SuperVIsion de situations de FORMation médiatisée » fait revivre la coopération entamée avec le LIUM en 2006.

Bien entendu, cette année 2014 est marquée par la rédaction de mon mémoire d'HDR, mais également par la poursuite de mes travaux autour de Loa. A l'issue de ce travail de rédaction, je souhaite en effet finaliser le langage Loa et proposer un éditeur Loa et deux compilateurs permettant d'adresser les plates-formes d'exécution Java et Web. La réalisation chronophage de ces tâches rend difficile la publication d'articles pour cette année 2014. Par contre, la théorie des opérations actives présentée dans ce mémoire est une refonte complète de l'article [24] non encore publiée. Il est d'ores et déjà envisagé de publier ce travail ainsi que les différentes réalisations autour du langage Loa en 2015.

2.2 Vue d'ensemble de la contribution

La figure 2.2.1 retrace l'ensemble de la contribution en mettant en avant les différentes composantes logicielles développées et présentées dans différentes publications.

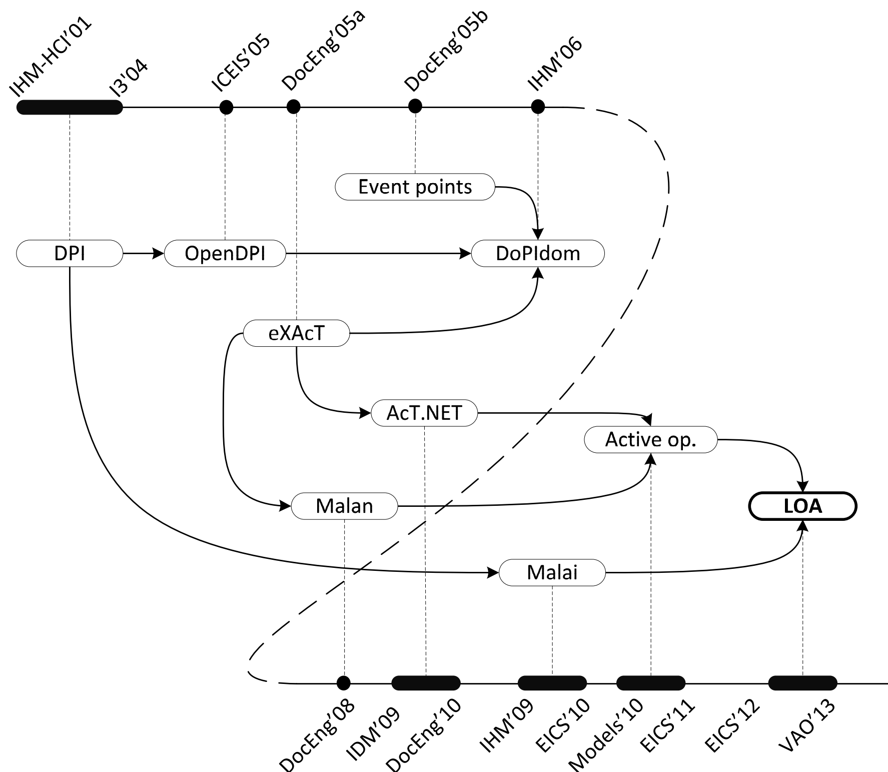

FIGURE 2.2.1 – Synthèse des travaux

Le modèle DPI (Document - Présentation - Instrument) est un modèle conceptuel mis au point de 2001 à 2004 pendant ma thèse [13, 20, 14] et visant à préciser les différentes facettes d'un système interactif (SI) manipulant des documents. Ce modèle a été évalué par différentes implémentations successives. La boîte à outils OpenDPI a d'abord été réalisée pendant la thèse afin d'illustrer les concepts de DPI par différents exemples d'implémentation [21, 22]. Le lien entre les documents D et leurs présentations P restait relativement imprécis dans le modèle conceptuel DPI, puisqu'il s'agissait d'un simple lien observable - observateur. La composante logicielle eXAcT a alors été conçue pour combler cette lacune. Il s'agit d'un processeur de transformation incrémentale (ou active) d'un document XML (modèle objet DOM) en une présentation de type graphe de scène

[16] (format SVG par exemple). Cette composante a été par la suite intégrée dans la boîte à outils DoPIdom qui propose une implémentation exhaustive du modèle DPI pour les documents XML [17]. Cette boîte à outils intègre également la composante « Event points » qui permet, par un placement judicieux de points d'évènement dans le graphe de scène des présentations, de répliquer tout ou partie du document ainsi que ses présentations [15]. Ces travaux autour de eXAcT et DoPIdom a constitué le travail d'après-thèse.

La seconde partie de la figure 2.2.1 montre l'évolution de ce travail qui s'est opérée sur la base de la thèse d'Arnaud Blouin [34], au travers du langage de correspondance Malan et du modèle d'interaction Malai. Cette évolution se traduit principalement par le passage du monde « document » (le D de DPI) au monde des « modèles » en utilisant des approches et concepts issues de l'Ingénierie Dirigée par les Modèles (IDM). La composante eXAcT a défini les principes d'implémentation d'un processeur de transformation active, mais ne propose pas un langage de haut niveau permettant de décrire les transformations actives : elles doivent être entièrement décrites en Java en respectant l'API défini par eXAcT. Malan propose donc un langage (DSL textuel) de correspondance, indépendamment de tout processeur de traitement incrémental [39]. AcT.NET est un outil permettant de définir des correspondances entre une source et une cible et de générer la transformation active correspondante [26, 23]. Cet outil propose un DSL graphique permettant de capturer la structure générale des correspondances, ainsi qu'un DSL textuel permettant d'en définir les détails. Les travaux autour de Malan et AcT.NET ont montré l'importance de proposer les algorithmes d'évaluation incrémentale des transformations en même temps que le langage de correspondance lui-même, ceci afin de garantir qu'il est possible de fournir une transformation active pour toute correspondance définie par le langage. Le concept d'opérations actives a été mis en œuvre dans l'objectif de satisfaire cette garantie : plutôt que de créer un nouveau langage de correspondance, les opérations actives sont des opérations usuelles (fonctions et opérateurs) sur les collections (telles que la sélection, la transformation ou le tri) pour lesquelles des algorithmes incrémentaux sont fournis [24] et garantissent l'exécution active des transformations. Les opérations actives sont donc à la fois des éléments pour définir des correspondances, et des éléments pour exécuter directement ces correspondances.

Il a été montré que l'usage de ces opérations actives dans le contexte du développement des systèmes interactifs permettait de préciser les liens de correspondance, des plus simples aux plus complexes, entre les données métiers et leur présentation [25]. Parallèlement à ces travaux, le modèle conceptuel DPI a évolué en un modèle formel Malai, ce dernier proposant un modèle de l'interaction utilisable conjointement avec la langage Malan pour modéliser les SI [37, 38]. Cette évolution généralise et précise les concepts du modèle DPI dans un contexte plus général que celui des documents, centré sur les modèles au sens de l'IDM. L'ensemble des travaux présentés dans la figure 2.2.1 (en excluant les « event points ») a été unifié dans un cadre logiciel appelé Loa [29, 28].

2.3 Le modèle DPI

La création du modèle Document-Présentation-Instrument (DPI) constitue la contribution essentielle de ma thèse [13]. Ce modèle forme la brique de base à partir de laquelle l'ensemble des travaux présentés dans ce mémoire a été élaboré.

Le modèle DPI est avant tout un modèle conceptuel dans ce sens qu'il ne dépend pas d'une implémentation particulière (section 2.3.1). Il vise notamment à pallier les insuffisances des modèles de type MVC en se focalisant sur l'interaction. Il intègre et formalise les principes de l'interaction instrumentale dans un modèle prenant en compte les éléments déjà définis dans les modèles de type MVC. De plus, ce modèle se veut centré sur les documents, par opposition aux approches centrées sur les applications. Le modèle DPI a ensuite été formalisé par un *modèle de composant* accompagné de sa notation graphique (section 2.3.2). Enfin, le modèle DPI a été implémenté dans la boîte à outils OpenDPI[1] à partir de laquelle une dizaine de scénarios d'interaction ont été implémentés [21, 22].

1. Étant principalement technique, OpenDPI n'est pas présenté dans ce mémoire.

2.3.1 Modèle conceptuel

Cette section présente une vue d'ensemble du modèle conceptuel DPI, avant sa formalisation sous la forme d'un modèle de composant. Dans un premier temps, les deux éléments essentiels du modèle que sont les instruments et les documents sont définis. Dans un second temps, la taxonomie des éléments du modèle est établie, en précisant comment ces éléments entrent en jeu dans une boucle action-perception. Dans un troisième et dernier temps, les principes de mise en œuvre du modèle DPI sont exposés en précisant comment les différents éléments du modèle interagissent entre-eux.

L'instrument, le document et la présentation

L'instrument et le document sont les deux métaphores constituant la base du modèle DPI. En tant que métaphores, ces deux éléments doivent apparaître naturels pour les utilisateurs. Les *documents* définissent le support des données tandis que les *instruments* correspondent aux moyens de création et de modification des documents.

La citation suivante caractérise à elle-seule le rôle des instruments : :

> « Un instrument est composé d'une *partie physique* et d'une *partie logique*. La partie physique comprend les transducteurs d'entrée-sortie utilisés par l'instrument, en entrée pour capter l'action physique de l'utilisateur et en sortie pour lui présenter un retour d'information (...). La partie logique de l'instrument comprend en entrée la méthode de transformation des actions de l'utilisateur sur l'instrument logique et en sortie la représentation de l'instrument. » [7]

Cette définition rejoint le concept d'interacteur, ce dernier représentant un objet pour le développeur là où l'instrument concerne davantage l'ergonome et le « designer ».

Un document est lui aussi composé de deux parties :

1. La partie *persistance* :

   Le document (*i.e.* le papier) est le support de la persistance puisqu'il absorbe l'encre du stylo. Quand il écrit sur un document, l'utilisateur perçoit le résultat persistant de son action (feedback). En lisant le document, l'utilisateur visualise en premier lieu la facette persistance et interprète ensuite son contenu.

2. La partie *présentation* :

   La présentation est intrinsèque au document et lui confère une présence et une apparence concrètes. Un document possède une présentation qui est le résultat direct de la persistance des informations écrites.

Si le document physique couple naturellement la persistance et la présentation, le document informatique découple *par nécessité* la persistance de la présentation. La persistance du document est gérée par le système de fichier, tandis que la présentation se situe au niveau des périphériques de sortie tel que l'écran. Ce découplage apparaît dans les systèmes usuels de par le besoin permanent de sauvegarder les documents.

Les trois éléments Document, Présentation et Instrument forment ainsi la base du modèle DPI.

Taxonomie des éléments et boucle action-perception

Le modèle DPI met en jeu ses trois éléments D, P et I selon un principe basé sur la théorie de l'action [87].

La figure 2.3.1 présente ce principe sous la forme d'un diagramme du flux d'action et de perception à trois niveaux : utilisateur, instrument et document [20].

1. Le niveau *utilisateur* précise que les utilisateurs spécifient leurs intentions par le biais d'*actions* (1, 7) et interprètent leurs résultats en utilisant leur sens (6, 10).

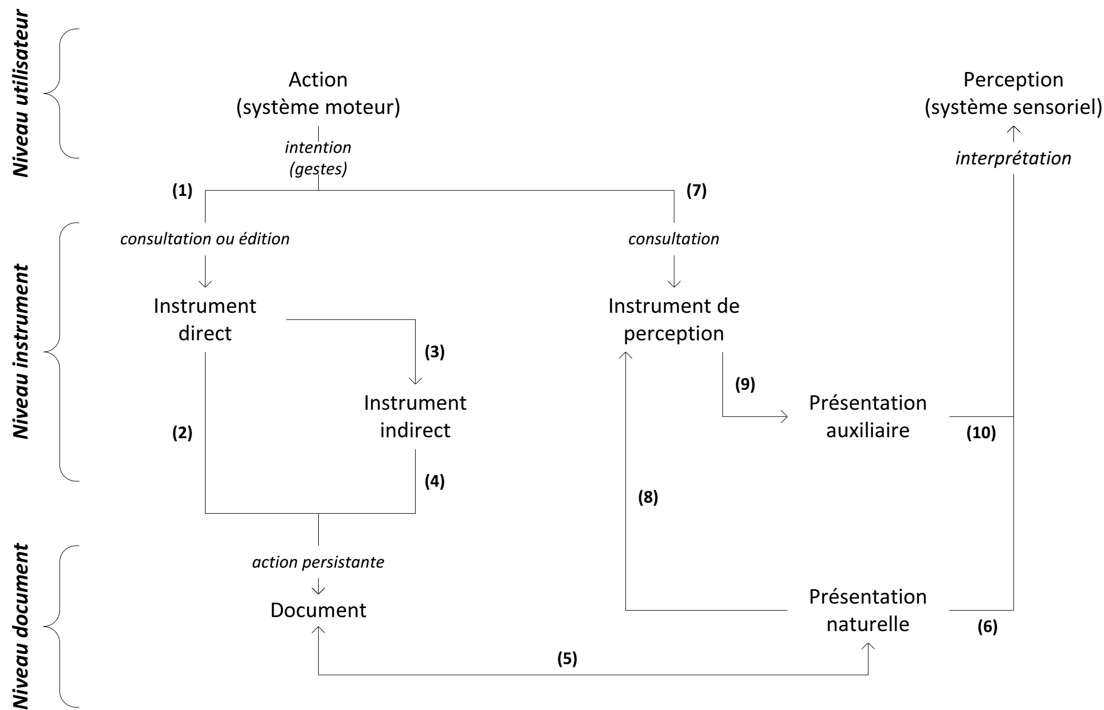

FIGURE 2.3.1 – Action et perception dans DPI

2. Le niveau *instrument* décrit comment les instruments transforment les actions des utilisateurs.

   Les instruments proposent deux types d'actions : l'édition (1) et la consultation (1, 7). La consultation peut être prise en charge par un *instrument direct* (1) tel qu'un instrument de défilement, par un *instrument indirect* (3) tel qu'un instrument de recherche, ou par un *instrument de perception* (7) tel qu'une vue radar. L'édition peut être réalisée par un instrument direct (1) tel qu'un stylo ou par un instrument indirect (3) tel qu'un correcteur orthographique.

   La perception est induite par la *présentation naturelle* d'un document (6) ou indirectement par des instruments de perception (8), telle qu'une loupe, qui fournissent des *présentations auxiliaires* (9).

3. Le niveau *document* illustre le rôle double des documents : la persistance des actions de l'utilisateur (2, 4) et la présentation de son contenu (5).

Cette figure laisse également apparaitre la taxonomie des instruments et des présentations : instruments directs (métaphore de l'outil), indirects (métaphore de l'appareil) et de perception (métaphore de la lentille optique), ainsi que les présentations naturelles et auxiliaires.

En séparant explicitement les instruments des documents, le modèle permet aux instruments d'agir sur des documents de natures variées, augmentant ainsi le nombre de combinaisons possibles. De telles combinaisons doivent simplifier l'interaction et réduire la charge cognitive des utilisateurs tout en offrant un ensemble de fonctionnalités riche. Par exemple, un même instrument peut être utilisé pour changer la couleur du titre dans un document ou la couleur d'un trait d'un graphique vectoriel. Les environnements actuels, basés sur la métaphore du bureau, n'offrent pas de telles capacités. Ce dernier point a un impact sur le principe de mise en œuvre des interactions entre les trois éléments D, P et I, principe présenté dans la section qui suit.

Principes de mise en œuvre du modèle

La communication entre un instrument et un document s'effectue *via* le vecteur appelé *action*. Une action est une grandeur qui peut être *produite* par (la partie logique de) l'instrument et *consommée* par un élément de l'une des présentations du document.

La figure 2.3.2 illustre le chaînage qui s'opère entre la production d'une action et sa consommation :

1. Lorsqu'une interaction est effectuée par l'utilisateur sur l'instrument, elle correspond à une intention particulière de l'utilisateur et donc à une action (intentionnelle) appartenant à l'ensemble des actions que l'instrument peut produire.
2. Par ailleurs, l'utilisateur précise *via* l'instrument (directement ou indirectement) quel élément est la cible de son action. Cet élément définit quant à lui les actions qu'il sait consommer.
3. Lorsque l'interaction a effectivement lieu, l'instrument effectue un test de compatibilité entre l'action qu'il doit produire et l'ensemble des actions que l'élément ciblé est en mesure de consommer. Dès lors que le test retourne au moins une action consommable compatible, le chaînage de l'interaction peut avoir lieu : l'instrument transmet l'action productible (choisie par l'utilisateur si plusieurs actions sont compatibles) vers l'élément qui la consomme.

Le couplage document - présentation assure la transmission des actions intentionnelles consommées par les présentations vers les données du domaine du document, ainsi que la synchronisation des présentations avec ces données. Il consiste à utiliser le patron de conception « observateur » [62] de manière à ce que les présentations observent tout changement d'état du document.

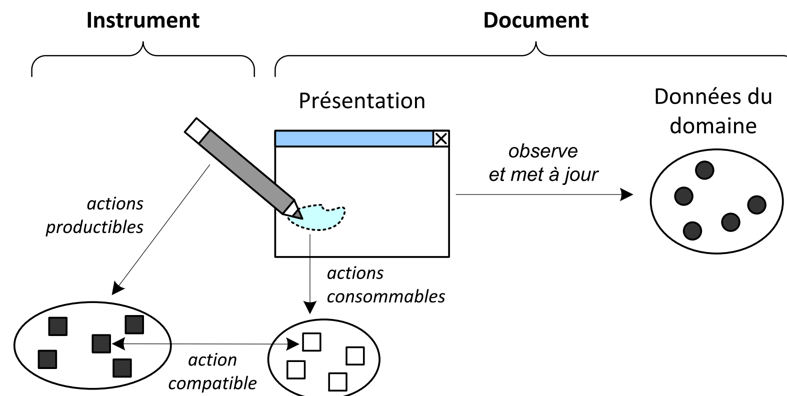

FIGURE 2.3.2 – Relation entre les éléments du modèle

La figure 2.3.3 précise la transformation des actions de l'utilisateur réalisée par les instruments :

1. L'utilisateur réalise son action intentionnelle en effectuant une *action physique* sur un ou plusieurs périphériques d'entrée.
2. Les périphériques mesurent l'action physique de l'utilisateur en fournissant l'*état physique* de leurs capteurs, tels que les déplacements X et Y et l'état des boutons d'une souris.
3. Les détecteurs de geste, définis dans la partie physique de l'instrument, analysent les états physiques des périphériques afin de détecter l'*action gestuelle* effectuée par l'utilisateur.
4. Cette action gestuelle est ensuite interprétée par l'instrument qui peut alors produire l'*action intentionnelle* sur la présentation consommatrice.

Cette figure complète la taxonomie des éléments du modèle DPI en ce qui concerne l'action : action physique, action gestuelle et action intentionnelle. Par abus de langage, le terme action désigne l'action intentionnelle (appelée « commande » dans l'interaction instrumentale), l'action physique et gestuelle correspondant à l'idée d'interaction (respectivement opérée par l'utilisateur, et détectée par le système).

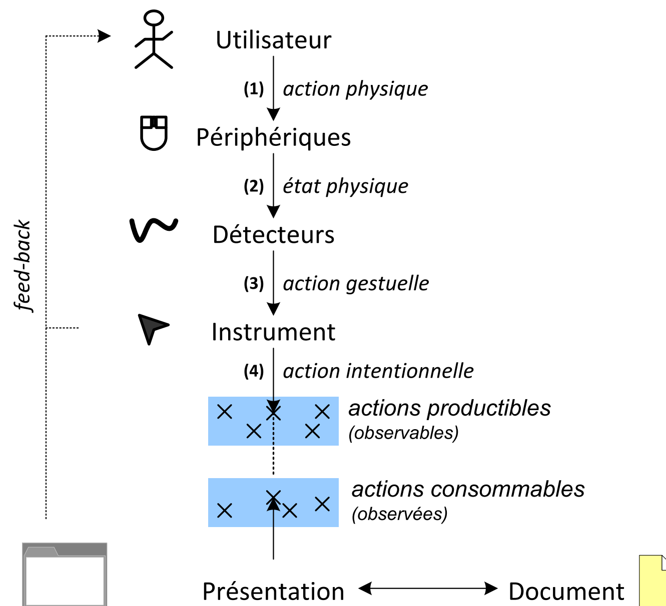

FIGURE 2.3.3 – Principe de l'interaction instrumentale

2.3.2 Modèle de composant

Cette section présente une synthèse du modèle de composant formalisant sous la forme d'un DSL graphique le modèle conceptuel DPI [14]. Ce modèle ne formalise donc pas l'implémentation du modèle DPI mais reste indépendant de la plate-forme. Une implémentation de ce modèle de composant a été réalisée sous la forme de l'API OpenDPI. Cette implémentation n'étant pas abordée dans ce mémoire, le lecteur est invité à consulter [22, 13] pour plus d'information sur le sujet.

États observables

L'état d'un composant DPI est défini par son état propre et son état structurel. L'*état propre* caractérise les valeurs des *propriétés* [109] propres au composant. Par exemple, le composant « case à cocher » définit un état propre booléen indiquant si la case est cochée ou non. L'*état structurel* qualifie la structure hiérarchique du composant en précisant ses composants fils. Par exemple, un composant « arbre » possède des composants arbres fils. L'état d'un composant est toujours observable et tout composant peut être observateur. Le mécanisme d'observation de DPI est plus précis que le patron de conception « observable / observateur » [62]. Il est fondé sur l'utilisation d'*interfaces*, au sens des langages IDL et Java [90], d'état observable et d'interfaces d'observation dont la figure 2.3.4 précise la notation graphique :

- Un disque blanc représente l'interface observable qui rappelle que tout composant est un observable.
- Un disque noir représente une interface d'observation implantée par le composant observateur :
  - Lorsqu'il est complété du symbole de composition, il représente une interface d'observation de l'état structurel (laquelle prend en compte les ajouts et retraits des composants fils).
  - Dans le cas contraire, il représente une interface d'observation de l'état propre (laquelle prend en compte les changements de l'état propre).

- La connexion entre l'interface observable et une interface d'observation indique que l'observation est engagée.

Les interfaces sont à implémenter par chaque composant pour préciser l'état observable et le résultat de l'observation.

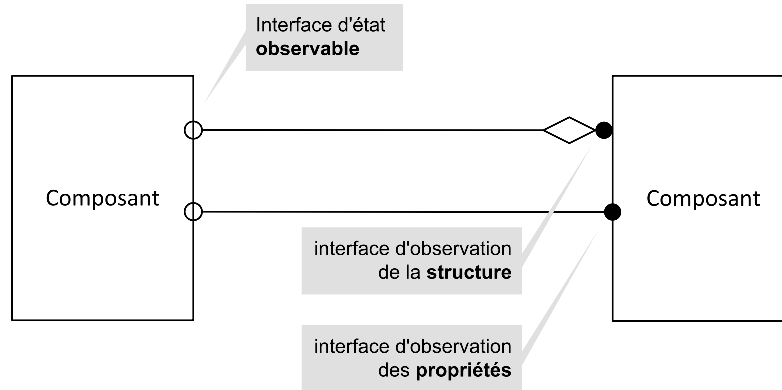

FIGURE 2.3.4 – Notation du composant observateur et observable

La figure 2.3.5 introduit l'exemple utilisé dans la suite pour illustrer le modèle de composant DPI. La détection des gestes de l'utilisateur est réalisée par des détecteurs spécialisés qui observent l'état propre de différents capteurs. Dans notre exemple, le geste de translation est détecté (*TranslateDetector*) par une observation des déplacements d'une souris qui sont capturés par les capteurs x et y (*HidSensor*) associés. De même, le geste d'appui est détecté (*PushDetector*) par une observation du capteur lié au bouton gauche de la souris. Cet exemple est complété dans la suite pour définir les composants « Hand » et « Toolglass ».

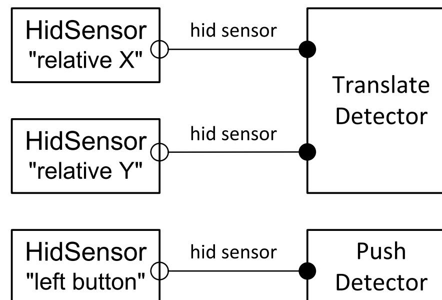

FIGURE 2.3.5 – Exemple de détecteurs de gestes

Production et consommation des actions

Si les états observables permettent de lier les différents composants afin de les synchroniser de manière réactive, ils sont insuffisants pour qualifier la chaîne complète depuis l'interaction qui se déroule au niveau présentation jusqu'à l'action résultante sur le document, action qui peut être annulée et rejouer ultérieurement. Le modèle DPI propose un modèle producteur - consommateur original qui permet d'isoler les instruments qui produisent les actions des composants qui les consomment.

Tout comme pour l'observabilité, ce modèle est fondé sur l'usage d'interfaces. À chaque classe d'action est associée une *interface de production* et une *interface de consommation*. L'*interface de production* d'une classe d'actions A est une interface généralement vide servant à indiquer que le composant est susceptible de produire des actions instances de A. L'*interface de consommation*

d'une classe d'action A définit un jeu de méthodes invoquées durant le cycle de vie de l'action (méthode de faisabilité, de début, d'exécution, de fin, d'annulation et de re-exécution). Elle doit être implantée par toute classe de composants consommateurs de l'action A.

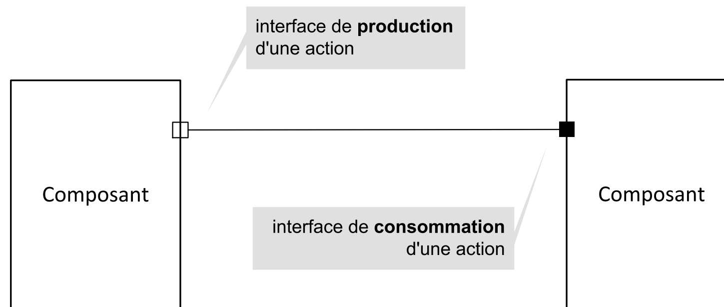

FIGURE 2.3.6 – Notation des actions pour les composants

La figure 2.3.6 précise la notation graphique utilisée pour représenter ces interfaces :

- Un carré blanc représente l'interface de production qui qualifie le composant de producteur de l'action.
- Un carré noir représente l'interface de consommation qui qualifie le composant de consommateur potentiel de l'action.
- La *possibilité* de produire l'action depuis le producteur jusqu'au consommateur est symbolisée par le lien orienté de l'interface de production vers l'interface de consommation.

La production d'une action depuis un composant producteur vers un composant consommateur suit le cycle suivant :

1. L'action est initialement associée à son producteur.
2. Lorsqu'une interaction pouvant donner lieu à la production de l'action a lieu, un test de faisabilité est effectué sur le composant cible.
3. Lorsque ce test répond positivement (par ex. lorsque les propriétés ou/et la structure concernées ne sont pas marquées), l'action démarre en marquant les propriétés ou/et la structure concernées puis s'exécute.
4. A la fin de l'interaction, l'action se termine et ôte tout marquage préalablement effectué.

La figure 2.3.7 illustre ce principe dans une chaîne instrumentale complète :

1. L'instrument main (*Hand*) est associé à la souris et observe les états propres de ses capteurs x, y et *bouton*. Cette observation est déléguée à deux composants détecteurs de geste : le détecteur de translation produit un geste de translation à chaque changement de l'état des capteurs de x ou y, et le détecteur d'appui produit un geste d'appui à chaque changement de l'état du capteur *bouton*. L'instrument consomme alors tout geste de translation et d'appui produit par les détecteurs.
2. Lorsque le geste d'appui démarre, l'instrument effectue un *piqué* qui consiste à déterminer quel composant graphique situé sous le curseur sera la cible de l'action de translation.
3. L'instrument entame la production de la translation vers la cible : la méthode de faisabilité indique que la translation peut être effectuée (nous supposons qu'il n'y a pas d'action concurrente) et la méthode de démarrage est invoquée sur la cible. La cible effectue alors un marquage des propriétés x et y.
4. Lorsque l'instrument consomme un geste de translation, il met à jour la position du curseur et invoque la méthode d'exécution de la translation sur la cible.
5. Lorsque le geste d'appui se termine, l'instrument invoque la méthode de fin sur la cible qui ôte alors le marquage précédent.

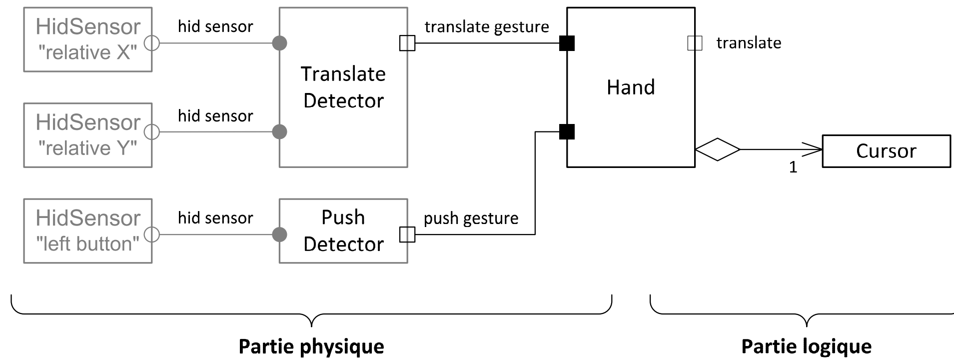

FIGURE 2.3.7 – L'instrument générique « Hand »

L'instrument *ColorToolglass*, qui permet l'application d'une couleur à partir d'un « toolglass »[33], entre un jeu dans une chaîne de composants donnée figure 2.3.8. Ce composant est disposé sur une palette, laquelle est liée à un dispositif de pointage (typiquement un « trackball ») *via* un détecteur de translation autorisant ainsi son déplacement. Le composant est constitué de boutons transparents (*ColorGlassButton*) susceptibles de produire l'action « peindre » sur l'objet ciblé par le « clic-au-travers ».

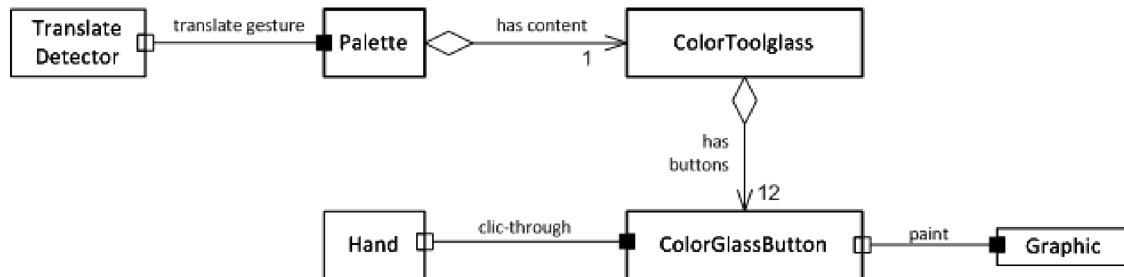

FIGURE 2.3.8 – Exemple de « Toolglass »

2.3.3 Conclusion et discussion

Le modèle DPI est un modèle conceptuel et demeure en ce sens indépendant de toute plate-forme d'exécution et de toute implémentation. Il pallie les insuffisances des modèles de type MVC en y intégrant notamment une formalisation de l'interaction instrumentale. Le modèle est centré sur les documents, mais il est généralisable à tout type de support de persistance de données : le document D deviendra dans la suite les données D du domaine de l'application visée. Un modèle de composant DPI a été proposé sous la forme d'un DSL graphique permettant de donner une vue d'ensemble des différents éléments D, P et I. Il autorise la génération d'une partie du code final du SI, les aspects à grain plus fin devant cependant être programmés manuellement dans le langage cible (par exemple en Java si l'API OpenDPI est utilisée). Ceci illustre l'intérêt de proposer conjointement un DSL graphique (tel que celui proposé dans la section 2.4.3) pour exprimer la vue d'ensemble du SI, et un DSL textuel (tel que celui introduit dans la section 2.5.2) pour les aspects plus précis.

La notation graphique de DPI suggère l'idée de « dataflow » : la figure 2.3.8 met en évidence le flot de données qui transite depuis les détecteurs jusqu'aux graphiques *via* différents instruments. On peut donc être tenté de comparer DPI avec les approches de « dataflow », telles que Whiizz [48] et ICON [57]. ICON propose un DSL entièrement graphique permettant de définir et paramétrer finement des interactions. Le modèle sous-jacent à ICON se focalise principalement sur l'interaction

et sur les objets graphiques (niveau présentation). Il n'adresse donc pas toutes les facettes d'un SI, en particulier les objets du domaine et le lien entre ces objets, les interactions et les objets graphiques. De plus, ICON est un DSL exclusivement graphique, ce qui rentre en contradiction avec l'intérêt d'un DSL mixte (graphique et textuel) souligné dans le paragraphe précédent. La finalité de DPI est donc relativement différente de celle des approches de type « dataflow ».

Cependant, si le modèle DPI est relativement précis quant à la formalisation de l'interaction instrumentale, il l'est insuffisamment en ce qui concerne le lien entre les données D et leur présentation P, ce dernier étant défini comme un simple lien observable-observateur. Les deux sections suivantes présentent les travaux réalisés pour pallier ce manque. La section 2.4 présente comment le concept de transformation active permet de définir le lien D↔P. La section 2.5 présente alors un cadre unifié qui synthétise tous les travaux faits depuis le modèle DPI jusqu'aux opérations actives. Cette unification n'est pas qu'une simple juxtaposition des travaux antérieurs : les éléments du langage Loa proposé *in fine* permettent de décrire, en plus des composantes D, P et I, les liens D↔P mais aussi les autres liens I↔P et I↔D.

2.4 Transformations actives et correspondances

Les transformations sont largement utilisées pour construire des représentations de données sources lisibles par l'humain. Le langage XSLT a popularisé cette approche : il a été largement utilisé pour fournir des représentations HTML de données au format source XML [58]. Le premier avantage de ce langage est d'être dédié à la transformation pour l'espace technique XML : il permet de décrire les transformations de documents XML sources en documents textuels cibles à partir de *règles* de transformation, là où l'usage d'un langage de programmation général (GPL) rendrait lourde l'écriture des transformations. Le second et principal avantage d'XSLT concerne la séparation des préoccupations : dans une transformation XSLT, il est possible de clairement séparer la logique de transformation (utilisation des « template match ») de la logique des éléments textuels produits en sortie (les « template name »). Cependant, XSLT définit la logique de transformation sur des expressions qui ne font pas strictement référence au schéma des données sources. La version 2 du langage a été proposée de manière à rendre possible la spécification du schéma source dans les règles de transformation [73]. Ceci ne garantit cependant pas la conformité des documents cibles produits relativement à un schéma cible. Il serait ainsi plus pertinent qu'un langage de transformation opère entre un schéma source et un schéma cible : en d'autres termes, il doit permettre de spécifier les *correspondances* de ces schémas. Par ailleurs, une transformation XSLT s'exécute selon un processus « batch », donc une fois pour toute. Un processeur incrémental incXSLT a été proposé [117], mais il possède le défaut inhérent au concept de transformation incrémentale : la *granularité* de la réévaluation incrémentale reste relativement élevée puisqu'elle correspond à la totalité du modèle produit par la règle devant être réévaluée.

La section 2.4.1 présente le processeur de transformation eXAcT qui permet d'exécuter des transformations actives, c'est-à-dire des transformation incrémentales de granularité fine [16]. Une telle granularité est possible en s'affranchissant de la notion de règle présente dans les langages dédiés à la transformations et en se focalisant alors sur le concept de correspondance. S'agissant d'un processeur et non d'un langage, une transformation active eXAcT est un programme Java relativement complexe à écrire.

Les sections 2.4.2 et 2.4.3 présentent chacune un DSL dédié à spécification de correspondances, lesquelles peuvent être traduites en transformations actives pour un processeur tel que eXAcT. Le premier langage, Malan, est un DSL textuel [39] complet, permettant de définir des correspondances complexes. Le second, AcT.NET [26] [23], est un DSL mixte (textuel et graphique), la partie graphique permettant de donner une vue d'ensemble des concepts mis en correspondance, et la partie textuelle permettant de décrire les relations précises entre les données des concepts mis en relation.

2.4.1 eXAcT : un processeur de transformations actives

Bien que les approches fondées sur la transformation de documents soient considérées comme très pertinentes pour la génération de présentations [96], les boîtes à outils récentes telles que Flex [112] ou WPF [112] ne proposent aucun mécanisme de transformation. Elles proposent plutôt des mécanismes de liaison de données («data binding ») qui sont d'un usage simple pour les liaisons usuelles, mais deviennent inutilisables seuls pour les liaisons plus complexes. De plus, les liaisons sont définies à l'intérieur même des présentations, là où les transformations séparent très clairement les données du domaine de leurs présentations.

Un chalenge important est ainsi de pourvoir tirer bénéfice de l'intérêt de la transformation de documents dans le contexte plus spécifique de la construction de systèmes interactifs. Ceci implique de repenser le processus de transformation d'un document source en présentations interactives cibles. Les transformations incrémentales qui étendent les processus usuels de transformation [92][117] et la méthode Ajax qui permet une modification partielle des pages Web côté client [63], illustrent ce besoin.

Les transformations XML actives eXAcT ont été proposées afin de redéfinir le concept de transformation dans les systèmes interactifs centrés sur les documents [16]. Une transformation eXAcT est un objet écrit en Java qui permet la transformation active de n'importe quel document XML en présentations, les modèles de présentation pouvant être SVG, X3D ou Draw2d.

Le modèle eXAcT inclut principalement un modèle de présentation fondé sur les fragments et les ancres, ainsi qu'un modèle de transformation qui définit le processus de transformation.

Modèle de présentation : les fragments et les ancres

Toute présentation cible d'une transformation eXAcT est une combinaison de fragments qui se composent les uns les autres *via* les ancres [16]. Un fragment définit l'apparence d'une partie de la présentation, l'ensemble des fragments formant alors la présentation finale. Les ancres, situées à l'intérieur des fragments, définissent les points de variations des fragments, ces variations pouvant concerner du texte simple ou une imbrication complexe d'autres fragments.

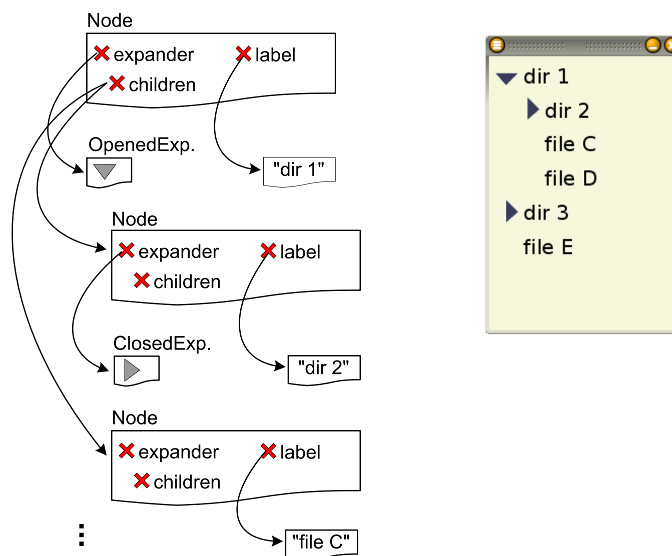

FIGURE 2.4.1 – Fragments et ancres par l'exemple

La figure 2.4.1 illustre le principe des fragments et des ancres sur un exemple simple. La partie droite montre la présentation finale d'une arborescence de fichiers et répertoires, la partie gauche représentant l'imbrication des fragments et des ancres relatifs aux trois premiers nœuds visibles

de l'arbre (« dir 1 », « dir 2 » et « file C »). Les différents fragments *Node*, *OpenedExpander* et *ClosedExpander* sont ici utilisés pour représenter une arborescence de fichiers, mais ils peuvent bien entendu être utilisés dans tout autre contexte. Le fragment *Node* définit l'apparence propre d'un nœud ainsi que les trois ancres *expander*, *label* et *children*. L'ancre *expander* définit l'emplacement du fragment *Node* susceptible d'accueillir soit un fragment *OpenedExpander* représentant un nœud dont le contenu est développé, soit un fragment *ClosedExpander* représentant un nœud non développé, soit aucun fragment dans le cas d'une feuille. L'ancre *label* définit l'emplacement du fragment *Node* qui accueillera un fragment textuel simple représentant son nom. L'ancre *children* définit l'emplacement du fragment *Node* susceptible d'accueillir d'autres fragments *Node* représentant les nœuds fils.

Tous les fragments sont définis dans un *unique* document XML, au SVG pour notre exemple :

```
<svg>
   <g exact:fragment='Node'>
      <exact:anchor name='expander' path='polygon'/>
      <exact:anchor name='label' path='text/text()'/>
      <exact:anchor name='children' path='g/g'/>
      <polygon exact:fragment='OpenedExpander' points='...' />
      <text>dir 1</text>
      <g transform='translate (...) '>
         <g transform='translate (...) '>
            <polygon exact:fragment='ClosedExpander' points='... ' />
            <text>dir 2</text>
         </g>
         <!-- etc. -->
      </g>
      <!-- etc. -->
   </g>
   <!-- etc. -->
</svg>
```

Le document fournit une scène affichant l'*ensemble* des représentations graphiques des fragments. Il est entièrement conçu par un « designer » graphique. Les fragments et leurs ancres sont ensuite définis en annotant le document *via* l'espace de noms *exact*. Les ancres permettent la construction active de la présentation, cette dernière étant pilotée par la transformation elle-même. Le processus de transformation est abordée dans la section suivante.

Modèle de transformation

La figure 2.4.2 donne une vue synthétique du processus de transformation active d'un document source en une présentation cible. Le document source est constitué d'un document XML gérant sa persistance, lié à un document DOM fournissant sa représentation objet. La présentation cible est constituée de sa partie logique, de sa partie physique et de son affichage : la partie logique définit l'imbrication des fragments *via* leurs ancres, la partie physique complète cette imbrication en fournissant les représentations concrètes des fragments, cette partie physique pouvant alors être affichée. Le lien entre les parties physique et logique est établi lors de l'instanciation des fragments, mécanisme qui n'est pas détaillé ici [2].

La transformation construit et met à jour la présentation logique en fonction des évènements DOM [118] produits par le document source lorsque ce dernier évolue [3] :

1. La construction initiale consiste à instancier différentes définitions de fragment et à assembler les fragments résultant en fonction de l'état initial du document source.

2. Pour plus de détails, se référer à [16, 18].
3. Les modifications du document DOM sont réalisées par des interacteurs eXAcT qui ne sont ici pas abordés pour des raisons de clarté.

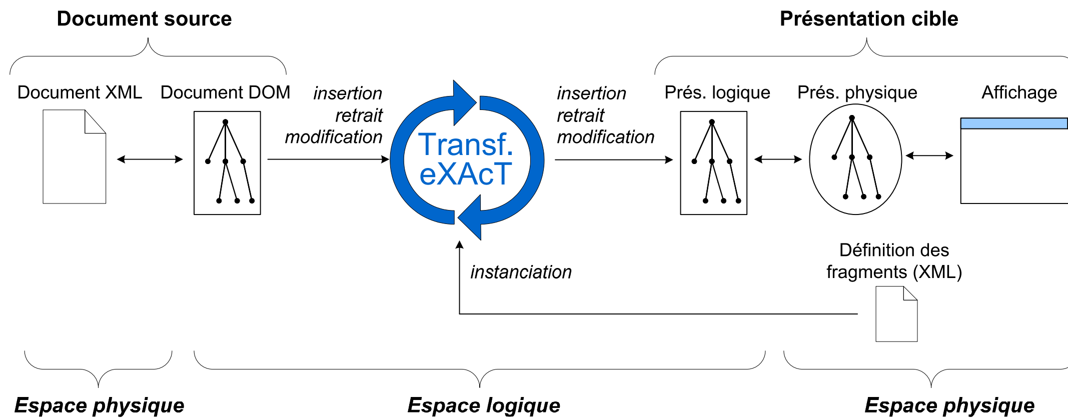

FIGURE 2.4.2 – Principe général d'une transformation active

2. L'insertion d'un nouveau nœud dans le document source implique la création d'un nouveau fragment le représentant, suivi de l'insertion de ce fragment dans la présentation (partie logique *et* partie physique).
3. Le retrait d'un nœud du document source implique le retrait du fragment qui a été associé à ce nœud lors du point 1 ou 2.
4. Finalement, tout changement de valeur dans un nœud du document implique la mise à jour du fragment associé.

Conclusion

La boîte à outils OpenDPI se focalise sur l'implémentation du modèle de l'interaction instrumentale formalisée par DPI, sans lien aucun avec un modèle standard de document. Son successeur, DoPIdom, implémente le modèle DPI sur la base du modèle objet DOM pour décrire les éléments D et P. Il intègre le processeur eXAcT pour définir le lien D↔P [12, 17]. Un éditeur SVG complet a été réalisé afin d'illustrer l'usage de DoPIdom.

L'intégration d'eXAcT dans DoPIdom permet la séparation claire entre les espaces logiques et les espaces physiques, et facilite la réutilisation des fragments dans une approche « boîte blanche ». Cependant, l'élaboration d'une transformation eXAcT nécessité l'écriture d'une classe Java à coder manuellement en utilisant l'API eXAcT qui implémente le processus de transformation. De plus, une transformation eXAcT ne prend pas en compte les schémas des documents sources et des présentations cibles. Les deux sections suivantes proposent chacune un langage dédié qui permet de décrire des correspondances entre schémas sources et schémas cibles.

2.4.2 Malan : Un langage de correspondances

Malan est un langage déclaratif permettant de mettre en correspondance un schéma source et un schéma cible. Il a été élaboré durant la thèse d'Arnaud Blouin [34]. Il est complémentaire d'eXAcT dans ce sens où il ne définit pas comment la mise en correspondance peut être réalisée par une transformation active : il se focalise sur la spécification de correspondances.

Un état de l'art exhaustif des langages de correspondance a été réalisé pendant cette thèse [40]. Le langage Malan a alors été construit afin de pallier aux insuffisances des langages de correspondance, ces derniers ne permettant pas de satisfaire les contraintes imposées par l'établissement et la gestion des liens entre les données D et leurs présentations P [39].

Les sections qui suivent présentent le concept de correspondance en le comparant à celui de transformation, puis comment Malan permet la modélisation des données des schémas sources et cibles ainsi que leur mise en correspondance.

Transformation *versus* correspondance

La figure 2.4.3 synthétise la différence essentielle entre correspondance et transformation. Une transformation opère directement au niveau modèle, ce qui offre l'avantage de la simplicité. Par exemple, construire un programme XSLT transformant un document XML en un document HTML reste une tâche simple tant que la grammaire du document XML n'est pas trop complexe. Cette approche a cependant deux inconvénients. Premièrement, le programmeur doit lui-même s'assurer que la transformation fonctionne pour tous les documents XML sources conformes à la grammaire source, et qu'elle génère des documents HTML cibles conformes à la grammaire HTML. Deuxièmement, le programme XSLT reste dépendant de la plate-forme des données (XML) et de la plate-forme graphique (HTML), le rendant inutilisable dans d'autres contextes.

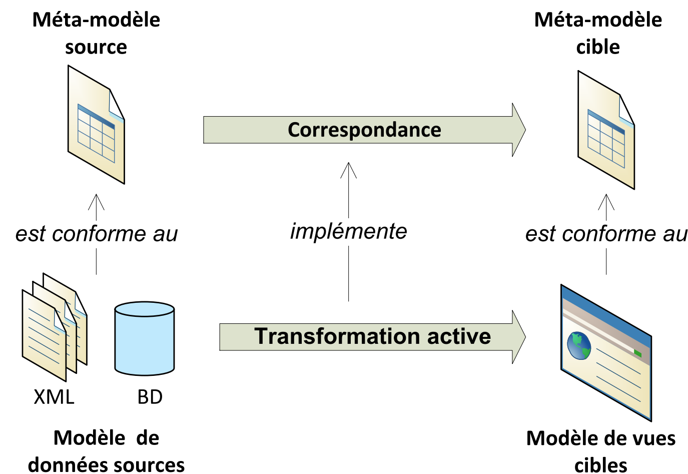

Figure 2.4.3 – Correspondance et transformation active

L'utilisation d'une correspondance plutôt qu'une transformation permet de travailler au niveau méta-modèle, supprimant alors les deux inconvénients précédents.

Modèle de données

Afin d'assurer la validité des données manipulées, le langage Malan travaille au niveau des méta-modèles : une correspondance de schémas s'établit entre les méta-modèles des données sources et celui des données cibles, tous deux décrits par des diagrammes de classes UML. En contre-partie, la passerelle entre les diagrammes de classes et les plates-formes de données est à la charge de l'implémentation de Malan.

Le choix des diagrammes de classes UML pour représenter les données se justifie pour les raisons suivantes :

1. Le langage UML, et en particulier les diagrammes de classes, est dédié à la modélisation de systèmes et est devenu l'outil principal de l'IDM pour définir les modèles et les méta-modèles.
2. Dans notre contexte, les données sources correspondent à des documents XML ou des données relationnelles, tandis que les données cibles représentent des graphes de scènes. L'utilisation d'UML dans le cadre d'XML et des données relationnelles a été largement étudiée sans présenter de limite majeure [108].
3. Les profils UML permettent d'étendre UML à la modélisation de domaines initialement non prévus.

L'exemple de la figure 2.4.4 décrit un *Blog* qui définit un *nom* et se compose d'un ensemble de *billets*. A chaque *Billet* correspond une *date*, un *numéro*, un *titre*, un *contenu* et un *auteur*, lequel

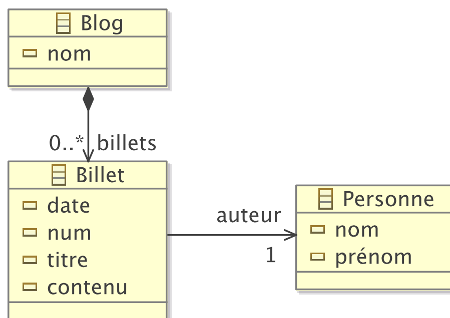

FIGURE 2.4.4 – Diagramme de classes d'un blog

possède un *nom* et un *prénom*. L'exemple du blog sera utilisé pour illustrer comment Malan permet de présenter à l'utilisateur les données d'un blog *via* une présentation décrite en SVG.

La manière dont Malan traite un méta-modèle exprimé par un diagramme de classe est détaillé dans la thèse [34]. Malan formalise un sous-ensemble des éléments entrant en jeu dans les diagrammes de classe UML (notion de classe, d'attribut, de relation, de cardinalité, d'héritage et de fonction), formalisation inspirée de [30].

Modèle de correspondance

Un modèle de *correspondance de schémas* définit comment les éléments des schémas source et cible peuvent être mis en relation. Une correspondance de schémas se décompose en différentes *correspondances de classes*, selon la définition formelle suivante :

> Soient $\mathcal{S}$ et $\mathcal{T}$ deux diagrammes de classes UML. Une correspondance de schémas est un prédicat $\mathcal{CS}(\mathcal{S}, \mathcal{T}, \Sigma)$ où $\mathcal{S}$ est appelé le schéma source et $\mathcal{T}$ le schéma cible. Σ est un ensemble des correspondances de classes décrites dans une logique $\mathcal{L}$ sur $\langle \mathcal{S}, \mathcal{T} \rangle$ et dont le but est de créer un lien entre les schémas source et cible.

Le modèle de données a introduit le prédicat $C(c)$ qui précise si c est une classe ou non, prédicat utilisé dans la définition formelle suivante de la correspondance de classe :

> Soient $\mathcal{S}$ et $\mathcal{T}$ deux diagrammes de classes UML. Une correspondance de classes est un prédicat $\mathcal{CC}(c_1, \ldots, c_n, t, \Omega)$ tel que $(c_1, \ldots, c_n) \in \mathcal{S}$ et $C(c_i)$, avec i de 1 à n, $t \in \mathcal{T}$ et $C(t)$. Ω est un ensemble de prédicats, appelés « instructions », décrits dans une logique $\mathcal{L}$ sur $\langle \mathcal{S}^n, \mathcal{T} \rangle$, établissant des liens entre les classes sources $(s_1, \ldots, s_n)$ et la classe cible t.

Les correspondances sont définies de manière déclarative, choix motivé par le fait que l'ordre des fonctions et des correspondances de classes à l'intérieur d'une correspondance de schémas n'a aucune incidence sur la sémantique de cette dernière. Il en est de même pour la définition des instructions d'une correspondance de classes.

Syntaxe concrète par l'exemple

Afin d'illustrer la syntaxe concrète du langage Malan, l'exemple du blog, dont le modèle a été introduit précédemment, est utilisé dans la perspective d'afficher son contenu en SVG. La vue d'ensemble de la correspondance entre un blog et une représentation SVG est donnée figure 2.4.5.

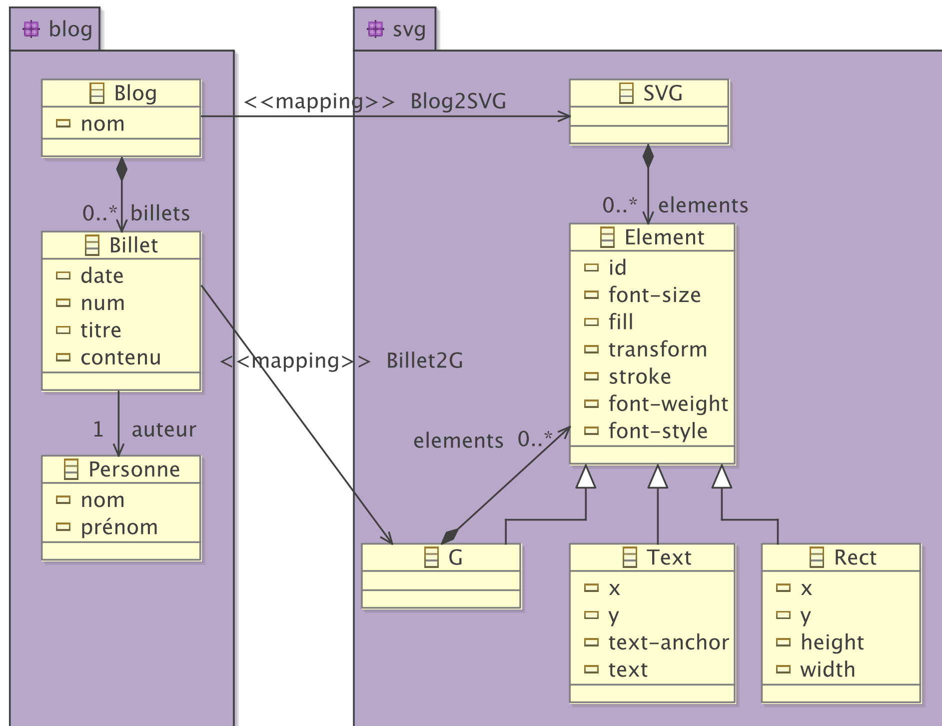

FIGURE 2.4.5 – Vue d'ensemble de la correspondance de schéma pour le blog

Le code Malan fourni à la page suivante précise le détail de la correspondance de schémas associée. La correspondance de schémas commence par préciser les *schémas* UML *source* et *cible* (ligne 1), ainsi que le *paramètre* de couleur d'affichage de chaque billet (ligne 2). Cette correspondance de compose de deux correspondances de classes, *Blog2SVG* (lignes 4 à 22) et *Billet2G* (lignes 24 à 49). La première correspondance met en relation (symbole ->) la classe *Blog* avec la classe *SVG* représentant l'ensemble du dessin vectoriel (ligne 4). Elle commence par déclarer cinq *alias* (lignes 5 à 9) qui permettent une meilleure lisibilité du code. Les différents attributs UML sont ensuite mis en correspondance (symbole ->) : des valeurs littérales sont d'abord assignées (lignes 11 à 15), puis une mise en correspondance directe d'attributs est définie (ligne 16), suivie de mises en correspondance utilisant différents calculs (lignes 17 à 19). La correspondance *Blog2SVG* se termine par la mise en correspondance de relations à cardinalité multiple (lignes 20 à 21) : la relation *gg* possède la même cardinalité que la relation *billets*, et chaque *billet[i]* est mis en correspondance avec chaque groupe *gg[i]*. Cette dernière mise en correspondance sera alors établie par un nombre d'applications de la correspondance *Billet2G* égal à *|billets|*. La correspondance de classes *Billet2G* suit le même principe : définition des alias (lignes 25 à 28), correspondances avec des constantes (lignes 30 à 39), correspondances simples d'attributs (lignes 40 à 43), et correspondances nécessitant des calculs (lignes 44 à 48).

2.4.3 AcT.NET : Un environnement centré sur les correspondances

Le langage de correspondance Malan permet la définition de correspondances de schémas représentés par des diagrammes de classe. Il reste donc indépendant des plates-formes et des domaines d'application. Dans le contexte des SI, il est cependant nécessaire de considérer la construction incrémentale des présentations, comme cela a été réalisé avec eXAcT. De plus, il est nécessaire de garantir que toutes les correspondances pouvant être définies avec Malan aient une implémentation

```
1  "exemple/blog.uml/blog" −> "exemple/blog.uml/svg" {
2      param couleurBillet
3
4      Blog2SVG : Blog −> SVG {
5          alias g             (G) elements[1]
6          alias titre         (Text) g.elements[1]
7          alias listeAuteurs  (Text) g.elements[2]
8          alias dateDernier   (Text) g.elements[3]
9          alias gg            (G) g.elements[4]
10
11         24             −> g.font−size
12         '#000000'      −> g. fill
13         20             −> titre.y
14         'middle'       −> titre. text−anchor
15         ' rotate(−90)' −> titre.transform
16         nom            −> titre. text
17         −| billets |∗140 −> titre.x
18         max(Blog. billets .date)                    −> dateDernier.text
19          billets .auteur.( nom + ' ' + prénom + ', ') −> listeAuteurs
20         | billets |     −> |gg|
21          billets [ i ]  −> gg[i], i =[1..| billets |]
22     }
23
24     Billet2G : Billet billet −> G {
25         alias rect  (Rect) elements[1]
26         alias text1 (Text) elements[2]
27         alias text2 (Text) elements[3]
28         alias text3 (Text) elements[4]
29
30         '#000000' −> rect.stroke
31         180       −> rect.height
32         630       −> rect.width
33         40        −> rect.x
34         60        −> text1.x
35         55        −> text2.x
36         'bold'    −> text1.font−weight
37         ' italic ' −> text3.font−style
38         'end'     −> text3.text−anchor
39         620       −> text3.x
40         num       −> id
41          couleurBillet −> rect. fill
42          titre     −> text1.text
43         contenu   −> text2.text
44         auteur.prénom + ' ' + auteur.nom + ' − ' +date −> text3.text
45         positionY( billet , 0)   −> rect.y
46         positionY( billet , 30)  −> text1.y
47         positionY( billet , 95)  −> text2.y
48         positionY( billet , 165)−> text3.y
49     }
50
51     function int positionY( Billet b, int supp) {
52         return ( position (b) − 1) ∗ 130 + 30 + supp;
53     }
54 }
```

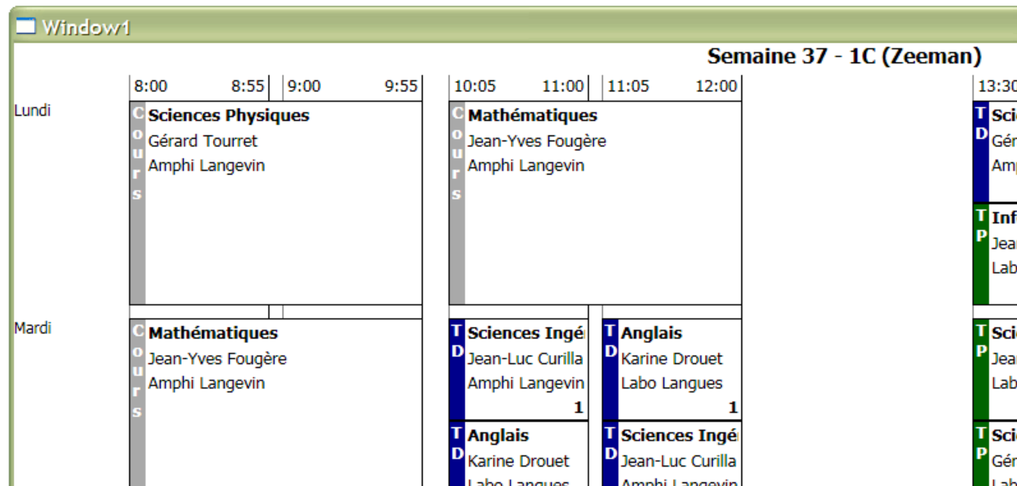

FIGURE 2.4.6 – Application de gestion d'un planning scolaire

incrémentale à grain fin (transformation active). L'objectif de l'environnement AcT.NET est ainsi de permettre [26] [23] :

1. La définition de correspondances un utilisant un DSL mixte, la partie graphique du DSL permettant de fournir une vue d'ensemble des classes et de leurs correspondances, et la partie textuelle du DSL d'un préciser les détails ;
2. La fourniture d'une transformation active pour chaque correspondance ;
3. La définition de nouveaux composants graphiques liés aux modèles cibles des correspondances.

Les sections qui suivent fournissent une synthèse de l'environnement AcT.NET en précisant comment sont définis les modèles des données sources, les modèles des présentations cibles, et les correspondances entre ces modèles. La figure 2.4.6 est une capture d'écran de l'application illustrative de gestion d'un planning scolaire, laquelle est utilisée tout au long de cette section afin d'illustrer les principes essentiels de AcT.NET.

Modèle des données sources

La figure 2.4.7 page suivante présente la correspondance réalisée avec l'application *AcTeditor*.

La partie gauche définit le modèle des données du domaine pour notre application d'édition de plannings scolaires. Une *Semaine* d'enseignement est décomposée en 5 *Jour*s d'enseignement, lesquels accueillent différents *Enseignements*. Chaque enseignement concerne une *Matiere*, est assuré par un *Enseignant*, a lieu dans une *Salle*, et se déroule sur une ou deux *PlageHoraire*s consécutives.

Modèle des présentations cibles

La partie droite de la figure 2.4.7 représente le modèle cible de la présentation, lequel est basé sur trois composants graphiques : le *CanvasAgenda* qui comprend des *LigneHeure*s ainsi que des *VignetteEvenement*s, chaque vignette pouvant accueillir différentes *Description*s textuelles.

Ce modèle est indépendant de la plate-forme d'exécution (ici .NET). Il doit cependant être complété par un modèle dépendant de la plate-forme .NET, lequel définit l'apparence concrète des différents composants utilisés dans l'application finale (représentée figure 2.4.6). La définition d'un composant pour l'environnement AcT.NET s'effectue en deux étapes :

1. Définition en XAML de la représentation graphique du composant ;

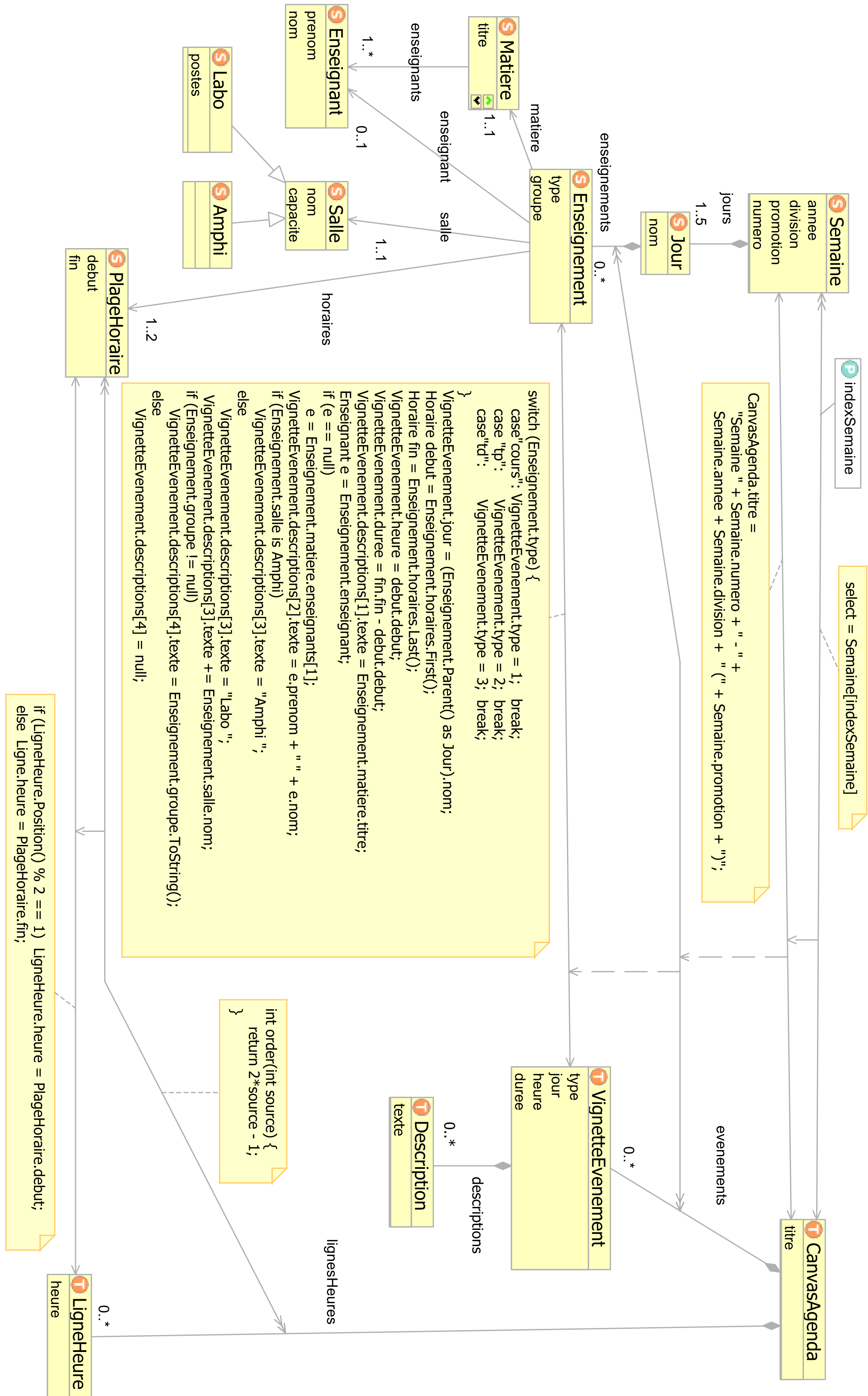

Figure 2.4.7 – Un exemple de correspondance

2. Définition en C# du lien entre les données du modèle du composant et sa description en XAML.

Le code suivant donne le modèle XAML du composant *LigneHeure* :

```
<UserControl>
    <Grid Name="ligneHeure" Canvas.Top="0" Height="800">
        <Line X1="0" Y1="0" X2="0" Y2="10000"/>
        <TextBlock Name="label" Text="" />
    </Grid>
</UserControl>
```

Ce composant est un contrôle utilisateur qui regroupe dans une grille nommée *ligneHeure* une ligne verticale ainsi qu'un *label*. Cette description en XAML doit être complétée par un code C# définissant la propriété *heure* de type *Time* et liant la valeur de cette propriété au contenu XAML précédent. Ce lien s'exprime donc dans le langage de la plate-forme d'exécution .NET.

Définition des correspondances

La structure générale de la correspondance est définie par les trois opérateurs *ForA*, *ForEach* et *Map*, représentés respectivement par les flèches $\twoheadleftarrow\!\!\rightarrow$, $\twoheadleftarrow\!\!\twoheadrightarrow$ et $\leftrightarrow$ (figure 2.4.7). Une flèche $\rightarrow$ dénote une instance unique et une flèche $\twoheadrightarrow$ dénote un ensemble d'instances. Les opérateurs sont chaînés entre-eux, chaînage représenté par la flèche $\dashrightarrow$. Le code associé aux opérateurs est donné dans la figure 2.4.7 *via* des notes pour plus de lisibilité ; dans l'application *AcTeditor*, ce code est éditable dans une boîte de dialogue.

La correspondance débute par l'opérateur *ForA* qui établit la correspondance entre la semaine (classe source *Semaine*) sélectionnée (mot clé *select*) *via* le paramètre *indexSemaine* et l'agenda graphique (classe cible *CanvasAgenda*). L'opérateur est ensuite chaîné à l'opérateur *Map* qui établit la correspondance entre la semaine sélectionnée et l'agenda graphique : le *titre* est formé à partir des différentes propriétés de la semaine. L'opérateur *Map* est chaîné à l'opérateur *ForEach* qui établit une correspondance de relations entre les enseignements de la semaine et les *evenements* du *CanvasAgenda* : il définit le fait que chaque enseignement de la semaine est associé à une vignette représentant un évènement de l'agenda. Enfin, chaque enseignement (classe *Enseignement*) est mis en correspondance avec une vignette évènement (classe *VignetteEvenement*) par l'opérateur *Map*. Les spécifications de l'enseignant, de la matière, de la salle et du groupe sont effectuées en faisant correspondre les propriétés de l'enseignement à 4 *descriptions* de la vignette évènement.

La correspondance comporte également l'opérateur *ForEach* entre les classes *PlageHoraire* et *LigneHoraire*. Il fait correspondre chaque instance de *PlageHoraire* avec deux instances de *LignesHeure*, l'une correspondant au début de la plage horaire, l'autre à la fin. Cette mise en correspondance utilise la relation d'ordre entre deux ensembles d'instances *via* la fonction *int order(int s)* qui fournit, pour chaque position des instances de *PlageHoraire* la position des instances de *LigneHeure*. Au niveau de la transformation active qui implémente cette correspondance, cette relation d'ordre induit la création de 2 instances de *LigneHeure* pour chaque nouvelle instance *PlageHoraire*, ainsi que leur insertion dans la relation *lignesHeures*. L'opérateur *Map* est finalement chaîné à ce dernier *ForEach* pour établir la correspondance entre les propriétés de *PlageHoraire* et de *LigneHeure*. Il utilise un test de parité de la position de *LigneHeure* (fonction *Position*) de manière à faire correspondre à *LigneHeure* soit le début de la plage horaire, soit la fin.

2.4.4 Conclusion

Les différentes travaux présentés dans cette section ont permis de montrer l'importance de considérer le lien entre les données du domaine et leurs présentations sous l'angle des correspondances

(langages Malan et AcT), tout en considérant le passage de ces correspondances à des transformations actives qui les implémentent (processeur de transformation eXAcT). De plus, les correspondances restent indépendantes de la plate-forme d'exécution, mais les fragments des présentations doivent être définis dans un ou plusieurs langages de cette plate-forme (par exemple en XAML et C# pour la plate-forme .NET).

Il subsiste quatre problèmes qui ne sont pas tous couverts par chacun de ces travaux :

Problème 1 - *Comment garantir qu'un langage de correspondance permette de décrire toute situation de mise en correspondance entre un modèle de données du domaine et un modèle de présentation ?* Malan répond à ce premier problème mais pas AcT.NET.

Problème 2 - *Comment garantir que toute correspondance soit transposable en une transformation active qui implémente son exécution incrémentale ?* AcT.NET résout ce problème mais pas Malan.

Problème 3 - *Comment définir les fragments des présentations cibles dans un formalisme non dépendant de la plate-forme d'exécution ?* Malan ne permet pas de définir de fragments (indépendance complète de la plate-forme) et les fragments dans AcT.NET sont décrits en XAML et en C# (donc en dehors du DSL AcT).

Problème 4 - *Comment définir des interacteurs dans un langage indépendant de la plate-forme d'exécution ?* Une approche minimaliste des interacteurs est intégrée dans eXAcT et AcT.NET, mais il reste à mener une réflexion plus aboutie sur cette question. En effet, il semble pertinent de considérer la notion de correspondance au niveau même de l'interacteur puisque ce dernier établit des correspondances ente les interactions, les actions et leurs effets sur les présentations ou/et les données du domaine.

La section suivante présente l'aboutissement des travaux qui, précisément, répond à l'ensemble de ces quatre problèmes en proposant un cadre unifié pour la conception de SI. Ce cadre est centré sur le langage Loa qui formalise l'usage des opérations actives dans le contexte des SI.

2.5 Vers un cadre logiciel unifié

L'aboutissement des travaux de recherche présenté dans ce mémoire est un cadre logiciel, appelé Loa, qui unifie l'ensemble des travaux menés depuis 13 ans (voir la figure 2.2.1 page 20). Cette unification s'est faite sur la base de deux travaux préalables : d'une part, le langage Malai défini dans la thèse d'Arnaud Blouin [34] et, d'autre part, le concept d'opération active défini dans la continuité des transformations actives (eXAcT et AcT.NET) et du langage Malan [19, 24, 25].

La section 2.5.1 donne une vue d'ensemble de cadre logiciel Malan/Malai en se focalisant sur les différents modèles de Malai qui permettent de caractériser les composantes des SI. La section 2.5.2 présente le concept d'opération active du point de vue théorique. L'unification par le cadre logiciel Loa est finalement présentée dans la section 2.5.3.

2.5.1 Le cadre logiciel Malan/Malai

Le langage Malan a pallié au premier manque du modèle DPI en ce qui concerne la caractérisation du lien D-P. Le langage Malai complète l'approche de Malan en caractérisant finement le modèle DPI dans une approche centrée sur les modèles, en mettant l'accent sur une formalisation de l'interaction instrumentale [37, 42, 38]. Contrairement à Malan, le langage Malai n'est pas entièrement formalisé : il est défini sous la forme d'un langage neutre de type « pseudo-code » donnant les principes du DSL Malai. L'usage conjoint de Malan et de Malai définit un cadre logiciel complet permettant de construire des SI.

Principe général

La figure 2.5.1 présente l'organisation d'un SI en différentes composantes définies par le cadre logiciel Malan/Malai. Le DSL Malan permet la définition du lien entre les *données* sources et leurs différentes *présentations,* alors que le DSL Malai permet la définition de la facette interactive.

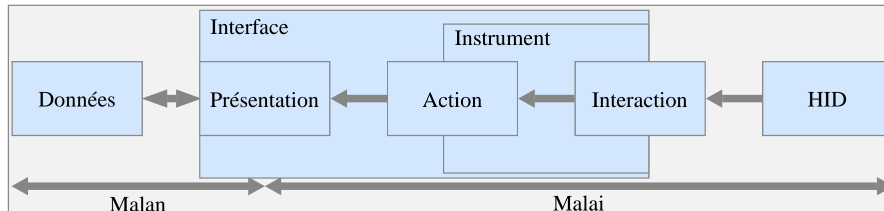

Figure 2.5.1 – Principe général de Malan/Malai

Un *instrument* sert de médiateur entre l'utilisateur et le système : l'utilisateur manipule des instruments en effectuant des *interactions* à l'aide de périphériques d'entrée (HID) afin de réaliser des *actions* qui modifient une *présentation* ou les données sources du SI.

Exemple

Afin d'illustrer le modèle des composantes de Malai, cette section utilise l'exemple d'une application permettant d'éditer un fichier XML par le biais d'un « widget » de type arbre.

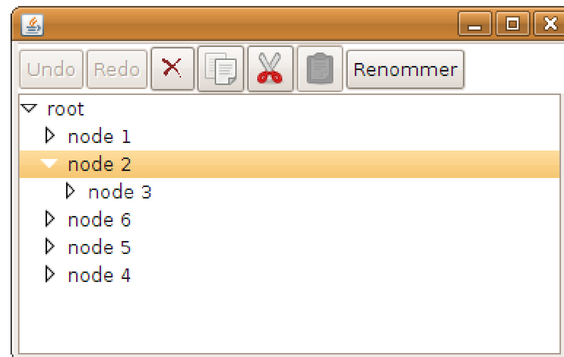

Figure 2.5.2 – Exemple d'application

La figure 2.5.2 fournit une présentation concrète *possible* pour une telle application. La figure 2.5.3 précise quant à elle les cinq composantes centrales de l'application que sont les données, les présentations abstraite et concrète, les actions, les instruments et les interactions, ainsi que les liens entre ces différentes composantes. Les classes *Noeud*, *NoeudRacine*, *NoeudTexte* et *Attribut* modélisent les données du document XML à éditer, et les classes *Arbre* et *Noeud* modélisent la présentation abstraite du « widget » arbre. Les classes *ArbreUI*, *NoeudUI* et *DeplieurUI* définissent la présentation concrète du « widget » arbre. Les classes *DéplacerNoeud*, *SupprimerNoeud* et *SélectionnerNoeud* représentent les actions pouvant être produites par les instruments *Main* et *Supprimer*, lesquels transforment les interactions *Glisser-Deposer* et *Touches*.

La figure 2.5.3 distingue clairement la *partie abstraite* de la partie concrète. Malai étend ainsi le concept de partie abstraite et de partie concrète à toutes les composantes du cadre logiciel. La partie concrète inclut les composants directement perceptibles par l'utilisateur, et la partie abstraite ceux qui ne le sont pas. Ainsi, les données du domaine, la partie abstraite de la présentation, mais également les actions, forment la partie abstraite en Malai. Les interactions et les présentations

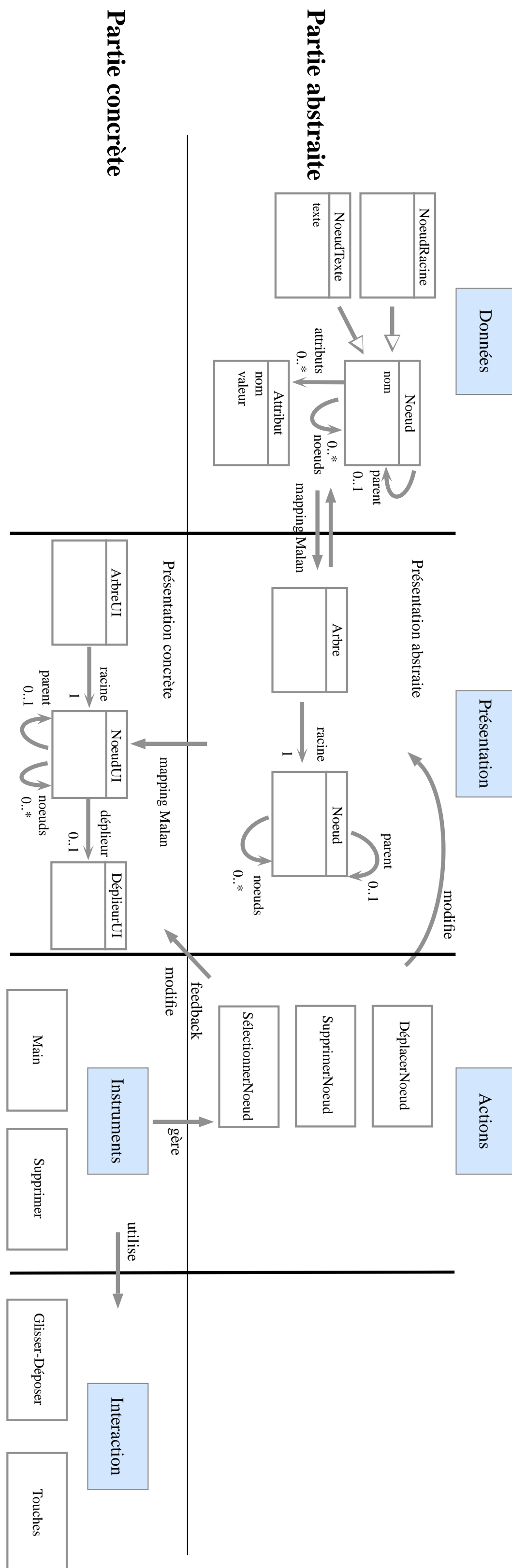

FIGURE 2.5.3 – Vue d'ensemble du modèle

concrètes sont naturellement présentes dans la partie concrète. C'est également le cas des instruments puisqu'ils sont dépendants, d'une part, des interactions et, d'autre part, des présentations concrètes sur lesquelles ils peuvent fournir un feedback intérimaire de l'action en cours.

Les sections qui suivent présentent les grandes lignes du DSL Malai en décrivant les modèles des composantes interaction, action et instrument. Pour chacun de ces modèles sont présentées les *parties statiques* qui définissent les données caractérisant l'état des composantes, puis les *parties dynamiques* qui précisent comment ces états évoluent dans le temps. Les modèles pour les composantes données et présentation ont été préalablement décrits dans la section 2.4.2 sur le langage Malan.

Modèle d'interaction

Une interaction est décrite par une machine déterministe à nombre fini d'états prenant en entrée des évènements HID pour en déduire l'*état* de sortie. La caractérisation de cet état forme la partie statique de l'interaction. Par exemple, la partie statique de l'interaction *Glisser-Déposer* de la figure 2.5.3 s'écrit en Malai comme suit :

```
Interaction Glisser −Déposer {
    // partie statique
    Point pointInitial
    Point pointFinal
    Objet objetCible
    Touche touche
}
```

Les données *pointInitial*, *pointFinal*, *objetCible* et *touche* précisent l'état courant de l'interaction. Cet état est mis à jour par la partie dynamique qui formalise une machine à états. La figure 2.5.4 représente la machine à états de l'interaction *Glisser-Déposer*. Cette interaction peut démarrer avec un évènement *pression* du dispositif de pointage, s'exécute tant que des évènements *glissement* surviennent, et se termine avec un évènement *relâchement*. Elle peut être *avortée* si l'utilisateur appuie sur la *touche* 'échap', ou si un évènement *relâchement* apparaît alors que l'état courant est *pressé*.

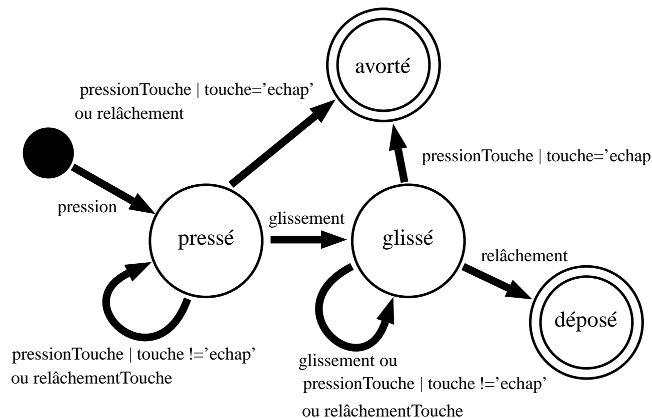

FIGURE 2.5.4 – Machine à état du « glisser-déposer »

Le pseudo-code Malai suivant est un extrait relatif à l'état *pressé* de l'interaction *Glisser-Déposer*. Il précise comment sont modifiées les données de l'interaction lors des transitions d'états caractérisées par les évènements HID :

```
Etat pressé {
    Transition glissement -> glissé {
        pointFinal = getPoint()
        objetCible = getObjet(pointFinal)
        notifier que l' interaction évolue
    }

    Transition pressionTouche='échap' -> pressé {
        touche = getTouche()
        notifier que l' interaction évolue
    }
}
```

Modèle d'action

Une action résulte du besoin pour un utilisateur ou un système de réaliser tout ou partie une tâche soit sur des données sources *via* une présentation, soit directement sur une présentation sans modifier les données. La spécification d'une action s'effectue indépendamment des instruments qui peuvent la produire.

La partie statique d'une action définit l'état courant de l'action. Dans l'exemple de la figure 2.5.3, l'action *SélectionnerNoeud* définit le nœud sélectionné comme suit :

```
Action SelectionnerNoeud {
    // partie statique
    Noeud sélection
}
```

L'exécution d'une action peut nécessiter l'exécution *préalable* d'autres actions. Dans notre exemple, le nœud de l'arbre doit d'abord être sélectionné avant d'être déplacé, couper, copié ou supprimé, et il doit être copié avant d'être collé. La partie statique de l'action *SupprimerNoeud* est définie comme suit :

```
Action SupprimerNoeud necessite SelectionnerNoeud est Annulable {
    // partie statique
    Noeud parent
    Entier position
}
```

L'état d'une action est mis à jour par une machine à état qui implémente le cycle de vie générique des actions représenté figure 2.5.5. Ce cycle de vie étend celui défini par le patron de conception *Commande* dans lequel les actions sont des objets paramétrables et dont les effets peuvent être annulés [62]. Cette extension consiste à considérer que le cycle de vie de l'action se déroule *en parallèle* de celui de l'interaction associée, ce qui permet de fournir en permanence à l'utilisateur un feedback sur l'état de l'action. Ce point constitue une contribution significative de Malai, les actions étant considérées dans la littérature comme des objets entrant en jeu une fois l'interaction terminée. Une fois *créée*, une *action en cours* peut être *exécutée* et *mise à jour* plusieurs fois. Une action en cours peut également être *avortée* pour être éventuellement *recyclée* en une autre action qui devient la nouvelle action en cours. Une fois l'*action terminée*, si l'exécution de l'action a eu des effets sur les données ou sur une présentation, elle est *sauvegardée* pour être éventuellement *défaite* et *refaite*. Une action *sauvegardée* peut être *supprimée* du système si la taille de la mémoire dédiée aux opérations annulables est limitée.

La partie dynamique des actions précise ce que l'action doit réaliser pour passer d'un état à l'autre. Elle doit fournir l'implémentation des fonctions *faire* et *peutFaire*, ainsi que celle des fonctions *defaire* et *refaire* dans le cas d'une action annulable. De plus, la fonction *interimFeedback* peut

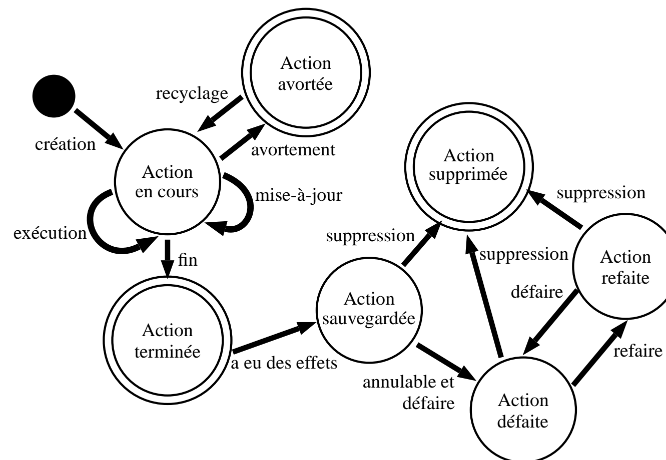

FIGURE 2.5.5 – Cycle de vie d'une action

être implémentée pour fournir à l'utilisateur un feedback intérimaire. Ce feedback consiste à fournir à l'utilisateur une réponse continue aux actions qu'il effectue sur le système [102] en présentant à l'utilisateur le résultat que provoquerait l'exécution de l'action [86]. La machine à état interne à l'action invoque ces différentes fonctions dès lors que les transitions concernées sont actives.

Le code Malai suivant précise comment l'action *SupprimerNoeud* supprime un nœud du document source XML :

```
Action SupprimerNoeud necessite SelectionnerNoeud est Annulable {
   // partie statique
   Noeud parent
   Entier position

   // partie dynamique
   peutFaire() { retourner sélection !=null }

   faire () {
      parent = sélection . parent
      position = parent. position (noeud)
      #supprimer(parent.noeuds, sélection )
   }

   defaire () { #ajouter(parent.noeuds, sélection , position ) }
   refaire () { #supprimer(parent.noeuds, sélection ) }
}
```

L'exécution de cette action (fonction *faire*) sauvegarde le nœud à supprimer, puis l'efface des données sources. Cette action est à même d'être défaite (fonction *defaire*) et refaite (fonction *refaire*). On notera que la variable sélection provient de l'action *SélectionnerNoeud*, nécessaire à l'action *SupprimerNoeud*.

Modèle d'instrument

Un instrument transforme l'interaction réalisée par l'utilisateur en actions s'appliquant sur les données sources ou sur l'une de leur présentation. En fonction de l'état de l'interaction en cours, l'instrument crée, met à jour, exécute, termine, avorte ou recycle l'action correspondante. De plus, il peut fournir un feedback intérimaire permettant à l'utilisateur de disposer d'informations sur

l'état de l'instrument et celui des actions que l'instrument peut exécuter, complété d'un éventuel feedback intérimaire défini par l'action elle-même. L'instrument établit ainsi le lien entre les cycles de vie de l'interaction et celui de l'action associée [4].

La partie statique de l'instrument précise l'interaction et l'action que l'instrument lie, ainsi que les paramètres de l'instrument que l'utilisateur peut modifier pour l'adapter à ses besoins. La partie dynamique de l'instrument précise les changements d'état de l'action associée en fonction des changements d'états de l'interaction. Ces changements concernent le démarrage, la mise à jour, la fin et l'avortement d'une interaction. Pour chacun de ces changements, l'instrument peut fournir du feedback intérimaire ou gérer l'action correspondante (créer, avorter, mettre à jour, exécuter, recycler ou terminer l'action).

Le code suivant est un extrait du modèle Malai pour l'instrument *Main* qui gère les actions *SélectionnerNoeud* et *DeplacerNoeud* en fonction des interactions *PressionBouton* et *GlisserDeposer* :

```
Instrument Main {
    // partie statique
    Action action
    Interaction inter

    // partie dynamique
    interactionDémarre( Interaction i ) {
        si (instrument déjà actif ) fin
        si ( i est PressionBouton et cible de i est NoeudUI ou ArbreUI) {
            créer une action SélectionnerNoeud
            activer l'instrument
        }
        sinon si ( i est GlisserDeposer et existe une action SelectionnerNoeud){
            si (touche de i==null et source est NoeudUI et cible==instrSupp.boutSupp)
                créer une action DeplacerNoeud
            activer l'instrument
        }
    }

    interactionEvolue ( Interaction i ) { ... }
    interactionTermine ( Interaction i ){... }
    interactionAvorte ( Interaction i ) { ... }
    recyclageActions ( Interaction i ) { ... }
    feedbackInterimaire (){ ... }
}
```

Les différentes fonctions présentées dans cet exemple caractérisent l'ensemble des définitions qu'un instrument doit fournir pour transformer des interactions en actions tout en informant l'utilisateur de l'état de cette transformation.

2.5.2 Théorie des opérations actives

Le langage Malan permet d'exprimer toute correspondance entre schémas sources et schémas cible (réponse au problème 1 mentionné dans la conclusion 2.4.4 page 39). Il ne garantit cependant pas qu'il existe une transformation active pour toute correspondance (problème 2).

L'approche des opérations actives prend un biais différent des approches qui visent à fournir un processeur de transformation incrémental pour des langages de transformation ou de correspondance *préexistants*. Les opérations actives définissent *en premier lieu* les algorithmes incrémentaux

4. Voir le chapitre 8 de la thèse pour la formalisation de ce lien [34]

qui rendent actives ces opérations [24]. Ces opérations peuvent dans un second temps être utilisées au travers d'un DSL spécifique.

Une opération active est une fonction ou un opérateur qui possède la capacité d'être réévaluée dynamiquement dès lors qu'un de ses arguments change de valeur : on parle alors de *mutation*. Considérons l'opérateur + actif dans le pseudo-code suivant :

```
1 a = b+c
2 inspecte(a)
3 b = 1
4 c = 2
```

La variable a est *définie* comme étant la somme des variables b et c (ligne 1). La fonction *inspecte* affiche dans la console la valeur initiale de a mais aussi toute valeur ultérieure (ligne 2). Lorsque les valeurs de b puis c sont modifiées (lignes 3 et 4), la console affiche alors les valeurs 1 puis 3. Cette façon de programmer définit un paradigme de programmation appelé « programmation réactive » [54].

La théorie des opérations actives, dénotée par le symbole $\mathfrak{T}$ dans la suite, peut être considérée comme une application de la programmation réactive sur les collections, les collections définissant l'objet de première classe pour spécifier des modèles exécutables. Le lecteur est averti que cette section est assez longue et que son contenu n'a pas encore été publié. La théorie $\mathfrak{T}$ proposée est une extension de la théorie des opérations actives sur les collections publiée dans [24].

Exemple illustratif

La figure 2.5.6 donne le modèle d'une application simple de gestion de répertoires téléphoniques. Un répertoire est un ensemble de contacts (classes *Directory* et *Contact*), chaque contact étant affiché à l'intérieur d'une liste (classe *List* et *ListItem*). Un contact est défini par son nom et son numéro de téléphone, lesquels sont éditables *via* deux champs textuels (classe *TextField*). Quatre correspondances *D2L*, *C2I*, *L2NF* et *L2PF* définissent les liens entre les différents éléments de l'application. Ces liens peuvent tous être modélisés et exécutés par des opérations actives. Cet exemple sera utilisé dans toute cette section pour illustrer la théorie $\mathfrak{T}$.

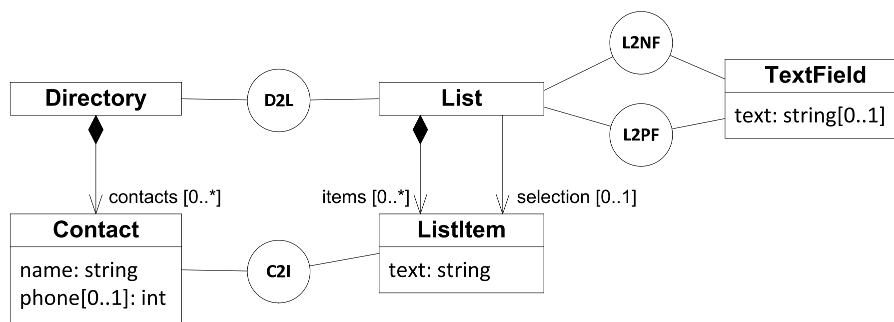

FIGURE 2.5.6 – L'exemple du « repertoire »

Modèles, contenus et contenants

Le modèle du « répertoire » de la figure 2.5.6 peut être intégralement abordé sous l'angle des *contenants*. Un *contenant* peut accueillir différents éléments qui forment alors son contenu. Les éléments peuvent être insérés, retirés et remplacés de leur contenant. Les contenants caractérisent la partie *modifiable* d'un modèle. L'élément d'un contenant est soit une *valeur*, soit une *référence*. Une *valeur* est par nature *immuable* alors qu'une *référence* désigne un objet constitué de contenants ; l'objet est donc modifiable puisque ses contenants le sont par nature.

La notation formelle de $\mathfrak{T}$, présentée dans cette section, est utilisée comme suit pour définir les types du modèle du « répertoire ». Une date est triplet d'entiers :

$$Date = int^3$$

Un répertoire est une référence sur un ensemble non ordonné de contacts, chaque contact étant une référence sur un triplet (nom, date naissance, numéros de téléphone) :

$$Directory =\uparrow set_{Contact} \text{ avec } Contact =\uparrow one_{string} \times one_{Date} \times seq_{string}$$

Une liste d'affichage est une référence sur un couple (items, item sélectionné), chaque item étant une référence sur un texte à afficher :

$$List =\uparrow oset_{ListItem} \times opt_{ListItem} \text{ avec } ListItem =\uparrow one_{string}$$

Enfin, un champs texte est une référence sur un texte optionnel à éditer ; si le contenant est vide, le champ texte n'est pas éditable :

$$TextField =\uparrow opt_{string}$$

L'ensemble $\mathbb{E} = \{int, float, bool, string\}$ définit les *types élémentaires* usuels de $\mathfrak{T}$. Les opérations (opérateurs et fonctions) sur ces types sont celles habituellement rencontrées dans les langages de programmation. La théorie $\mathfrak{T}$ n'étant pas un DSL, ces opérations n'y sont pas explicitées. Les produits cartésiens $\mathbb{E}^n$ permettent de définir des n-uplets de valeurs, comme dans l'exemple du type $Date$. Tout type $T \in \mathbb{E}^n$ avec $n \geq 1$ défini un type de valeur, ce qui est dénoté par le prédicat $\mathcal{V}$ défini par :

$$\mathcal{V}(T) \Leftrightarrow T \in \mathbb{E}^n \text{ avec } n \geq 1$$

L'ensemble $\mathbb{C} = \{opt, one, set, oset, seq, bag\}$ définit les six des types de contenant : les types singletons *opt* et *one*, et les quatre types collections *set*, *oset*, *seq* et *bag* définis par OCL [119]. Le prédicat $\mathcal{C}$ précise si un type représente une collection ou non :

$$\mathcal{C}(T) \Leftrightarrow T \in \mathbb{C}$$

Un produit cartésien $T \in \mathbb{C}^n$ de types de contenant n'est intéressant à considérer qu'au travers de références. Un type référence se compose du type T lui-même et du type T_O représentant l'objet référencé, selon la notation $T =\uparrow T_O \Leftrightarrow T_O = T\uparrow$. Le prédicat $\mathcal{R}$ précise si un type représente une référence ou non :

$$\mathcal{R}(T) \Leftrightarrow \exists T_O,\, T =\uparrow T_O$$

Les trois prédicats $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{R}$ caractérisent les trois grandes classes des types manipulés par la théorie. Ils sont qualifiés de *restrictions* puisque les opérations actives ont en général une sémantique restreinte à certaines de ces trois classes. Le prédicat $\mathcal{I}$ précise si un type représente une entité immuable ou non. A l'instar des langages fonctionnels, tout type représentant une valeur représente une entité immuable :

$$\mathcal{V}(T) \Rightarrow \mathcal{I}(T)$$

Un contenant joue alors le rôle de pivot entre les valeurs immuables et les modèles dont l'état est par nature modifiable :

$$\mathcal{C}(T) \Rightarrow \neg\mathcal{I}(T)$$

Une référence est établie une fois pour toute et est donc immuable, mais l'objet référencé reste quant à lui modifiable :

$$\mathcal{R}(T) \Rightarrow \mathcal{I}(T) \wedge \neg\mathcal{I}(T\uparrow)$$

A ces quatre prédicats sont ajoutés le prédicat d'unicité $\mathcal{U}$, le prédicat d'ordre $\mathcal{O}$ et la cardinalité $\mathcal{N}$, lesquels définissent les *contraintes* caractérisant les six types de contenant :

<table>
<tr><th></th><th>$\mathcal{U}$</th><th>$\mathcal{O}$</th><th>$\mathcal{N}_{min}$</th><th>$\mathcal{N}_{max}$</th></tr>
<tr><td>one</td><td rowspan="4">$\mathcal{U}$</td><td rowspan="3">$\mathcal{O}$</td><td>$\mathcal{N}_{min} = 1$</td><td rowspan="2">$\mathcal{N}_{max} = 1$</td></tr>
<tr><td>opt</td><td rowspan="5">$\mathcal{N}_{min} = 0$</td></tr>
<tr><td>oset</td><td rowspan="4">$\mathcal{N}_{max} = \star$</td></tr>
<tr><td>set</td><td rowspan="2">$\neg\mathcal{O}$</td></tr>
<tr><td>bag</td><td rowspan="2">$\neg\mathcal{U}$</td></tr>
<tr><td>seq</td><td>$\mathcal{O}$</td></tr>
</table>

La cardinalité $\mathcal{N}_{min} = 1$ d'un contenant de type *one* impose qu'un élément soit toujours présent dans le contenant. Ceci est rendu possible par définition de la fonction $\mathcal{D}$ qui retourne une instance par défaut pour n'importe quel type considéré. Les valeurs par défaut des types élémentaires représentent leurs « zéros » :

$$T \in \mathbb{E} \Rightarrow \mathcal{D}(T) \in \{0, 0.0, non,''\}$$

Un contenant vérifiant $\mathcal{N}_{min} = 0$ est par défaut vide, sinon il contient par défaut l'élément $\mathcal{D}(A)$:

$$T \in \mathbb{C} \Rightarrow (\mathcal{N}_{min}(T) = 0 \wedge \mathcal{D}(T) = \{\}) \vee (\mathcal{N}_{min}(T_A) = 1 \wedge \mathcal{D}(T_A) = \{\mathcal{D}(A)\})$$

Un n-uplet contient par défaut les instances par défaut de chaque type T_k du produit cartésien associé :

$$T \in \mathbb{E}^n \vee T \in \mathbb{C}^n \Rightarrow \mathcal{D}(T) = (\mathcal{D}(T_1), ..., \mathcal{D}(T_n)) \text{ avec } n \geq 2$$

Enfin, une référence par défaut référence l'instance par défaut du type référencé :

$$\mathcal{R}(T) \Rightarrow \mathcal{D}(T) =\uparrow \mathcal{D}(T\uparrow)$$

Ce dernier point évite la définition de la référence « nulle » qui conduirait à des évaluations erronées d'une chaîne d'opérations actives (erreur de type « null reference exception »). La référence « nulle » de la POO correspond dans $\mathfrak{T}$ à contenant vide de type *opt*.

Cet ensemble de définitions permet la spécification formelle de modèles orientés objet, comme l'exemple du « répertoire » le montre. Un type d'objet, appelé *classe*, se compose de quatre parties :

1. Les *attributs* représentant des valeurs modifiables ;
2. Les *relations* représentant des références à des instances modifiables ;
3. Les *paramètres* représentant des valeurs immuables ;
4. Les *objets composés* représentant les références immuables d'objets immuables.

Les parties 1 et 2 suffisent à caractériser la plupart des modèles en utilisant les six types de contenant. Les parties 3 et 4 ont des usages spécifiques au domaine des IHM [5] : les symboles graphiques peuvent vouloir définir des paramètres à valeur immuables, et les interacteurs composent des classes représentant des interactions avec des classes représentant des actions.

Les différentes restrictions $\mathcal{V}$, $\mathcal{C}$, $\mathcal{R}$ et $\mathcal{I}$, et des contraintes $\mathcal{U}$, $\mathcal{O}$ et $\mathcal{N}$ entrent en jeu dans la caractérisation de ces quatre parties. Un attribut de type A est représenté par un contenant de

5. L'application des points 3 ou/et 4 à d'autres domaines n'a pas été identifiée à ce jour.

type $C_A \in \mathbb{C}$ vérifiant toujours $\mathcal{V}(A)$, le plus souvent $\neg\mathcal{U}(C_A)$, et généralement $\mathcal{N}_{max}(C_A) = 1$. Une relation vers des objets instances du type A est représentée par un contenant de type $C_A \in \mathbb{C}$ vérifiant toujours $\mathcal{R}(A)$, toujours $\mathcal{U}(C_A)$, généralement $\neg\mathcal{O}(C_A)$ pour les objets métiers, et généralement $\mathcal{O}(C_A)$ pour les objets d'IHM (l'ordre a son importance pour l'affichage). L'usage des paramètres et des objets composés n'est pas détaillée ici étant donné leur spécificité [6].

Les différents types de $\mathbb{C}$ définissent un jeu commun d'opérations correspondant aux fonctions habituelles sur les collections. Cet ensemble d'opérations actives comporte actuellement plus d'une centaine d'opérations, lesquelles sont largement inspirés des opérations définies dans les langages OCL [119] et Scala [88]. Parmi ces opérations, une vingtaine sont des opérations de base fournissant leur propre implémentation active ; une cinquantaine sont des opérations définies par équivalence à partir des opérations de base ; une quarantaine correspondent aux opérations usuelles définies pour les types de $\mathbb{E}$.

Taxonomie des opérations

La taxonomie des opérations actives est composée de cinq grandes classes d'opérations, chacune divisée en deux sous-classes. L'arbre suivant fournit les opérations les plus significatives de cette taxonomie [7], les opérations marquées d'une $\star$ étant des opérations de base :

1. Opérations mathématiques
   (a) sur les ensembles : *union*, *intersection**, *difference**, *concat**, *zip**
   (b) sur les vecteurs (prédicat $\mathcal{O}$) : *indices**, *indicesOf*, *indexOf*, *slice**, *at*, *first*, *last*, *size*
2. Opérations de transformation
   (a) sur le type des éléments : *apply**, *uapply*, *map**, *mappedBy*, *path**
   (b) sur la cardinalité (singleton $\leftrightarrow$ collection) : *times**, *merge**, *split**
3. Opérations sur les vues
   (a) Sélection : *select**, *reject*, *when**, *count*, *contains*, *ifElse*, *ifTrue*, *ifFalse*, *is*
   (b) Tri : *sort**, *order**, *revert*
4. Opérations de conversion
   (a) des collections (contraintes $\mathcal{C}_{min}$, $\mathcal{C}_{max}$, $\mathcal{O}$ et $\mathcal{U}$) : *distinct** , *asOne*, *asOpt*, *asBag*, *asSeq*, *asSet*, *asOSet*
   (b) des éléments contenus dans les collections (forçage de type) : *as*
5. Opérations usuelles sur les types élémentaires *int*, *float*, *bool* et *string* :
   (a) Opérateurs : + , − , ∗ , / , % , ! , && , || , & , | , *etc.*
   (b) Fonctions : *sin*, *cos*, *min*, *max*, *intToString*, *substring*, *toUpperCase*, *toLowerCase*, *etc.*

Les fonctionnalités de chacune de ses opérations ne sont pas détaillées dans ce mémoire. L'usage des opérations actives essentielles que sont *apply*,*zip*, *map*, *mappedBy* et *path* pour les opérations de transformation, ainsi que *select*, *order*, *sort* pour les opérations sur les vues, est abordé dans la section suivante au travers de l'exemple du « répertoire ».

6. Certains exemples seront néanmoins fournis dans la section 2.5.3 page 58
7. Les surcharges de fonctions ne sont par exemple pas incluses dans cet arbre.

Usage des opérations

L'expression $b_B = a_A \to apply(f)$ applique la fonction $f : A \to B$ à chaque élément du contenant a_A et retourne le contenant b_B. Par exemple, soit b un one_{Date} avec $Date = int^3$ (date de naissance d'un contact) ; l'expression suivante retourne un one_{string} représentant la date convertie en une chaîne de caractères (affichable dans un champ texte) :

$$b \to apply(dateToString) \text{ avec}$$
$$dateToString(y, m, d) = intToString(d) +' /' + intToString(m) +' /' + intToString(y)$$

L'expression $ab_{A\times B} = a_B \to zip(b_B)$ regroupe les contenus respectifs des contenants a_A et b_B dans un unique contenant $ab_{A\times B}$. Utilisée conjointement à *apply*, l'opération *zip* permet l'application d'une fonction à plusieurs arguments. Par exemple, soit n un one_{string} représentant le nom d'un contact et b un one_{Date} représentant la date de naissance du même contact ; l'expression suivante retourne un one_{string} permettant l'affichage textuel du couple (n, b) :

$$n \to zip(b) \to apply(s, d \mid s +' \; (' + dateToString(d) + ')')$$

L'expression $b_B = a_A \to map(f)$ met en correspondance des éléments du contenant source a_A avec des éléments du contenant cible b_B par application de la fonction $f : A \to B$. Contrairement à *apply*, l'opération *map* concerne des contenants de références ; les liens entre les éléments sources et cibles sont mémorisés pour la fonction *mappedBy*. Par exemple, soit *contacts* un répertoire de type $set_{Contact}$ avec $Contact =\uparrow one_{string} \times one_{Date} \times seq_{string}$; l'expression suivante retourne un set_{Item} avec $Item =\uparrow one_{string}$ affichant de manière textuelle le contenu du répertoire :

$$contacts \to map(C2I) \text{ avec}$$
$$C2I(\uparrow (n, b, p)) =\uparrow n \to zip(b) \to apply(s, d \mid s +' \; (' + dateToString(d) + ')')$$

L'expression $a'_A = b'_B \to mappedBy(f)$ permet alors de retrouver dans a'_A des éléments sources mis en correspondance avec des éléments cibles b'_B *via* une opération *map* préalable. Par exemple, soit *sel* un opt_{Item} contenant l'item sélectionné dans la liste d'affichage des contacts ; l'expression $sel \to mappedBy(C2I)$ retourne alors un $opt_{Contact}$ contenant le contact qui a été mis en correspondance avec l'item sélectionné.

La navigation dans un graphe d'objets s'effectue en utilisant l'opération *path* qui, comme pour *apply* et *map*, applique une fonction aux éléments d'un contenant. L'expression $c_C = a_A \to path(f)$ avec $f : A \to B_C$ permet de sélectionner, pour chaque élément e de a_A où A vérifie $\mathcal{R}(A)$, une propriété de e. Cette propriété étant un contenant, le contenant résultat c_C est constitué de la concaténation des contenants $b_{i,C}$ de type B_C fournis par $f(e)$. Par exemple, l'expression suivante retourne un bag_{string} contenant les noms des contacts :

$$names = contacts \to path(\uparrow (n, b, p) \mid n)$$

L'expression $b_A = a_A \to select(f)$ sélectionne les éléments de a_A qui satisfont le prédicat $f : A \to bool$. Par exemple, soit un ensemble *dates* de dates ; la sélection des dates avant le XXIème siècle s'écrit comme suit :

$$dates \to select(y, m, d \mid y < 2000)$$

L'opération *select* est restreinte aux types A vérifiant $\mathcal{V}(A)$. Dans le cas contraire, la seule observation de a_A ne suffirait pas pour l'évaluation active de *select*. Afin d'étendre la sélection aux types A vérifiant $\mathcal{R}(A)$, l'opération *select* est surchargée en utilisant un « contenant prédicat » b_{bool} qui *réifie* en un contenant la « fonction prédicat » $f : A \to bool$. L'expression $c_A = a_A \to select(b_{bool})$ sélectionne les éléments de a_A qui satisfont le prédicat b_{bool}, ce qui implique d'utiliser l'ordre interne à tout contenant (qu'il soit ordonné ou non). Par exemple, la sélection des contacts nés avant le XXIème siècle s'écrit comme suit :

$$contacts \rightarrow select(pred) \text{ avec}$$
$$pred = contacts \rightarrow path(\uparrow (n, b, p) \mid b) \rightarrow apply(y, m, d \mid y < 2000)$$

L'opération *apply* réifie la « fonction prédicat » $f(y, m, d) = y < 2000$ en « contenant prédicat » *pred*.

L'expression $b_A = a_A \rightarrow sort(f)$ avec $f : A^2 \rightarrow bool$ retourne le contenant b_B ayant le même contenu que a_A mais trié selon la relation d'ordre définie par f. Par exemple, l'ensemble des noms des contacts peut être simplement trié par l'expression suivante :

$$names \rightarrow sort(f) \text{ avec}$$
$$names = contacts \rightarrow path(\uparrow (n, b, p) \mid n) \text{ et } f(s_1, s_2) = s_1 < s_2$$

Le tri multicritère peut s'effectuer en combinant les opérations *zip* et *sort*. Par exemple, l'ensemble *namesDates* des couples (nom, date de naissance) des *contacts* peut être trié par leurs noms dans l'ordre alphabétique puis par leur date de naissance du moins âgé au plus âgé comme suit :

$$namesDates \rightarrow sort(f) \text{ avec}$$
$$namesDates = contacts \rightarrow path(\uparrow (n, b, p) \mid n \rightarrow zip(b)) \text{ et}$$
$$f(s_1, d_1, s_2, d_2) = (s_1 < s_2) \vee ((s_1 = s_2) \wedge d_1 > d_2)$$

L'opération *sort* est, pour les mêmes raisons que *select*, restreint aux types A vérifiant $\mathcal{V}(A)$. Afin d'étendre le tri aux types A vérifiant $\mathcal{R}(A)$, l'opération *sort* est surchargée en utilisant un « contenant d'ordre » b_{int} qui *réifie* en un contenant la « fonction d'ordre » $f : A^2 \rightarrow bool$. L'expression $c_A = a_A \rightarrow sort(b_{int})$ trie les éléments de a_A qui satisfont l'ordre b_{int}, lequel précise l'indice j dans c_A de l'élément e situé à la position i dans a_A par la relation $j = b_{int}[i]$. Par exemple, le tri des contacts par leur nom puis leur date de naissance s'effectue comme suit :

$$contacts \rightarrow sort(order) \text{ avec}$$
$$order = namesDates \rightarrow order(f)$$

L'opération *order* réifie la « relation d'ordre » f en un « contenant d'ordre » *order*.

Liaisons inter-contenant

Les opérations actives permettent de définir :

1. Des transformations actives qui créent de *nouveaux* contenants ;
2. Des contraintes actives entre des contenants *préexistants*.

La section précédente illustre comment les opérations actives permettent de construire des transformations actives. Cette section présente comment il est possible d'établir des liaisons entre des contenants *préexistants*.

L'expression$b_A = a_A$ précise que le contenant b_A représente le *même* contenant que a_A : il s'agit d'un alias. L'opérateur $=$ possède donc la même sémantique qu'en mathématique. L'opérateur $::=$ permet quant à lui de définir une *liaison unidirectionnelle* entre un contenant source à un contenant cible. L'expression $b_B ::= a_A$ signifie que les deux contenants a_A et b_B sont *distincts* mais ont le même contenu, le sens de la copie se faisant de a_A vers b_B. Pour que la copie ait une sémantique correcte, il est nécessaire de restreindre les types A et B de sorte qu'ils vérifient $A \subseteq B$, c'est-à-dire soit $A = B$, soit A est un type dérivé de B (noté $A \subset B$).

Dans une liaison $b_B ::= a_A$, le contenant a_A est généralement calculé par une chaîne d'opérations actives $a_A = a'_{A'} \rightarrow op_1 \rightarrow ... \rightarrow op_2$. Par exemple, dans notre exemple du « répertoire », la liste d'items textuels est un contenant cible lié comme suit au contenant source définissant les contacts :

$$list ::= contacts \rightarrow sort(order) \rightarrow map(C2I) \text{ avec}$$
$$order = contacts \rightarrow path(\uparrow (n, b, p) \mid n \rightarrow zip(b)) \rightarrow order()$$

Il est possible d'établir une liaison dans le sens inverse, d'un contenant cible vers un contenant source, soit $b_B ::= a_A$ avec $b_B = b'_{B'} \rightarrow op_1 \rightarrow ... \rightarrow op_n$. Les opérations op_k ne doivent par contre pas transformer $b'_{B'}$, sinon la liaison n'aurait aucune sémantique : ces opérations doivent nécessairement retourner un sous-ensemble de $b'_{B'}$ et peuvent donc être des *sélections* et des *chemins*. De plus, le contenant retourné par l'avant-dernière opération op_{n-1} est restreint aux singletons ($\mathcal{N}_{max} = 1$), sinon la liaison n'aurait là aussi aucune sémantique. Par exemple, l'édition du nom du contact correspondant à l'item *sel* sélectionné dans la liste d'affichage des contacts *via* le champ texte $textField$ de type opt_{string} peut se définir par la liaison suivante :

$$sel \rightarrow mappedBy(C2I) \rightarrow path(\uparrow (n, b, p) \mid n) ::= textField$$

Le fait d'autoriser des liaisons dans les deux sens (source vers cible et cible vers source) permet d'envisager les *liaisons bidirectionnelles*. Ainsi les expressions $b_B ::= a_A$ et $a_A ::= b_B$ sont équivalentes à l'expression $b_B :=: a_A$ où :=: est l'opérateur de liaison bidirectionnelle, et où $A \subseteq B \wedge B \subseteq A$ soit $A = B$ nécessairement. De plus, la contrainte $\mathcal{U}(a_A) \wedge \mathcal{U}(b_A)$ doit être respectée afin d'éviter tout cycle. L'exemple précédent relatif à l'édition du nom du contact est réalisé par la liaison bidirectionnelle suivante :

$$sel \rightarrow mappedBy(C2I) \rightarrow path(\uparrow (n, b, p) \mid n) :=: textField$$

Règles de typage

Les opérations actives imposent presque toujours des restrictions, exprimées par les prédicats $\mathcal{V}$, $\mathcal{C}$, $\mathcal{R}$ et $\mathcal{I}$, sur les contenants qu'elles manipulent. Les contraintes $\mathcal{U}$, $\mathcal{O}$ et $\mathcal{N}$ sont quant à elles utilisées pour caractériser le type des contenants retournés par les opérations. L'ensemble des définitions de ces restrictions et contraintes constitue les règles de typage des opérations actives.

La transformation $b_B = a_A \rightarrow apply(f)$ impose la restriction $\mathcal{V}(A)$ permettant de garantir que f n'introduise pas de mutations non observables. Le contenant b_B vérifie : $\mathcal{U}(b_B) = \mathcal{N}_{max}(a_A) \leq 1$ puisque la fonction f est susceptible engendrer des doublons sauf si a_A est un singleton ; $\mathcal{O}(b_B) = \mathcal{O}(a_A)$ puisque l'ordre est conservé ; $\mathcal{N}(b_B) = \mathcal{N}(a_A)$ puisqu'il s'agit d'une opération de transformation. L'expression $b_B = a_A \rightarrow map(f)$ impose au contraire de *apply* la restriction $\mathcal{R}(A) \wedge \mathcal{R}(B)$. Le contenant b_B est exactement de même type que a_A : $\mathcal{U}(b_B) = \mathcal{U}(a_A)$, $\mathcal{O}(b_B) = \mathcal{O}(a_A)$ et $\mathcal{N}(b_B) = \mathcal{N}(a_A)$.

La sélection $b_A = a_A \rightarrow select(f)$ impose la même restriction $\mathcal{V}(A)$ que *apply*, et pour les mêmes raisons. Le contenant b_A est presque de même type que a_A puisque $\mathcal{U}(b_B) = \mathcal{U}(a_A)$ et $\mathcal{O}(b_B) = \mathcal{O}(a_A)$, mais la cardinalité minimale est 0 puisqu'il est possible de n'avoir aucun élément sélectionné, soit $\mathcal{N}(b_A) = (0, \mathcal{N}_{max}(a_A))$.

Les opérations ensemblistes (union, intersection, etc.) imposent des restrictions de dérivation sur le type des éléments des contenants, au même titre que la liaison $b_B ::= a_A$ impose $A \subseteq B$. La concaténation de deux contenants $c_C = a_A \rightarrow concat(b_B)$ impose d'avoir $A = B = C$ si $\mathcal{V}(A) \vee \mathcal{V}(B)$, et $A \subseteq C \wedge B \subseteq C$ si $\mathcal{R}(A) \vee \mathcal{R}(B)$. Le contenant c_C vérifie alors : $\neg\mathcal{U}(c_C)$ puisque d'éventuels doublons peuvent apparaître entre entre a_A et b_B ; $\mathcal{O}(c_C) = \mathcal{O}(a_A) \wedge \mathcal{O}(b_B)$ puisque la concaténation conserve l'ordre ; la cardinalité est la cardinalité maximale de a_A et b_B définie par $\mathcal{N}_{min}(c_C) = max(\mathcal{N}_{min}(a_A), \mathcal{N}_{min}(b_B))$ et $\mathcal{N}_{max}(c_C) = max(\mathcal{N}_{max}(a_A), \mathcal{N}_{max}(b_B))$.

Le regroupement $ab_{A\times B} = a_B \rightarrow zip(b_B)$ n'impose pas de restriction particulière sur A et B. Le contenant $ab_{A\times B}$ vérifie $\mathcal{U}(ab_{A\times B}) = \mathcal{U}(a_A) \vee \mathcal{U}(b_B)$ et $\mathcal{O}(ab_{A\times B}) = \mathcal{O}(a_A) \wedge \mathcal{O}(b_B)$, sa cardinalité étant la plus étroite soit : $\mathcal{N}_{min}(ab_{A\times B}) = max(\mathcal{N}_{min}(a_A), \mathcal{N}_{min}(b_B))$ et $\mathcal{N}_{max}(ab_{A\times B}) = min(\mathcal{N}_{max}(a_A), \mathcal{N}_{max}(b_B))$.

Mutation et observabilité

L'observation des *mutations* opérées sur les contenants est à la base de l'implémentation des opérations actives. Pour un contenant a_A donné, le flux des mutations de a_A est noté $\mathcal{M}(a_A)$,

l'*observation* des mutations s'écrivant alors $\forall m \in \mathcal{M}(a_A)$, ... Tous les contenants, qu'ils vérifient $\mathcal{O}$ ou $\neg\mathcal{O}$, utilisent en *interne* un tableau. Par conséquent, toute mutation $m \in \mathcal{M}(a_A)$ précise en plus de l'élément e de type A concerné par la mutation, la position i de cet élément dans le tableau interne du contenant. Une mutation peut être :

- Une insertion notée $(i, e)^+$ où e représente l'élément inséré à la position i ;
- Un retrait noté $(i, p)^-$ où p représente l'élément retiré de la position i ;
- Un remplacement noté $(i, p, e)^=$ où e représente l'élément remplaçant l'élément p situé à la position i.

Une opération active ayant en argument un contenant a_A observe les mutations de a_A selon la forme canonique suivante :

$$\begin{array}{c} \forall i \in [0, |a_A|[,\ t_o(i, a_A[i]) \\ \forall (i,e)^+ \in \mathcal{M}(a_A),\ t_1(i,e) \\ \forall (i,p)^- \in \mathcal{M}(a_A),\ t_2(i,p) \\ \forall (i,p,e)^= \in \mathcal{M}(a_A),\ t_3(i,p,e) \end{array}$$

Les t_k représentent les différents traitements réalisés par l'opération active. Le traitement t_0 correspond au traitement initial permettant de définir le contenu initial du contenant résultat. Les traitements t_1, t_2 et t_3 correspondent à la mise à jour du contenant résultat en fonction des mutations observées. Pour la plupart des opérations actives unaires, on peut utiliser le traitement $t_0(i, e) = t_1(i, e)$ pour initialiser le contenu du contenant résultat. De plus, un remplacement $(i, p, e)^=$ est équivalent au retrait $(i, p)^-$ suivi de l'insertion$(i, e)^+$. Il est alors possible d'utiliser le traitement $t_3(i, p, e) = t_2(i, p), t_1(i, e)$ pour gérer toute mutation $(i, p, e)^=$. Cependant, t_1 ou/et t_2 induisent généralement des décalages dans le contenant résultat, alors que t_3 est la plupart du temps un traitement optimisé évitant de tels décalages.

Chaque traitement t_k consiste en la mise à jour du contenant résultat *via* différentes mutations à opérer sur ce contenant. Une mutation m est *opérée* sur un contenant b_B en injectant m dans le flux des mutations $\mathcal{M}(b_B)$ *via* l'opérateur $\leftarrow$, soit $\mathcal{M}(b_B) \leftarrow m$. La théorie $\mathfrak{T}$ ne formalise pas comment les mutations sont appliquées aux contenants, ceci étant du ressort du langage de la plate-forme d'exécution. On a cependant la correspondance typique suivante entre la notation formelle de la théorie et une implémentation dans un langage cible :

Notation formelle	Langage cible
$\mathcal{M}(b_B) \leftarrow (i,e)^+$	b.insert(i,e)
$\mathcal{M}(b_B) \leftarrow (i,p)^-$	b.removeAt(i)
$\mathcal{M}(b_B) \leftarrow (i,p,e)^=$	b.set(i, e)

Charge au langage cible de gérer la lecture et l'écriture des flux de mutation, de manière synchrone ou asynchrone (utilisation de file d'attente).

Implémentation par observation

Les 17 opérations de base ainsi que les opérateurs de liaison fournissent leur implémentation active à partir de l'observation des mutations qui surviennent sur les contenants passés en argument.

Une liaison simple $b_B ::= a_A$ définit un traitement trivial commun à toutes les mutations puisqu'il s'agit de reproduire les mutations de a_A sur b_B :

$$\forall m \in \mathcal{M}(a_A),\ \mathcal{M}(b_B) \leftarrow m$$

Une transformation de valeurs $b_B = a_A \rightarrow apply(f)$ définit un traitement simple des mutations consistant à appliquer f sur les valeurs insérées, retirées ou remplacées :

$$\forall (i,e)^+ \in \mathcal{M}(a_A),\ \mathcal{M}(b_B) \leftarrow (i, f(e))^+$$

$$\forall (i,p)^- \in \mathcal{M}(a_A),\, \mathcal{M}(b_B) \leftarrow (i, f(p))^-$$

$$\forall (i,p,e)^= \in \mathcal{M}(a_A),\, \mathcal{M}(b_B) \leftarrow (i, f(p), f(e))^=$$

Une mise en correspondance de références $b_B = a_A \rightarrow map(f)$ est implémentée *via* les traitements identiques à *apply*, à ceci près que les liens entre la référence source e et la référence cible $f(e)$ sont sauvegardés pour être utilisés par l'opération *mappedBy*.

Une sélection d'éléments $b_A = a_A \rightarrow select(f)$ est implémentée en répercutant sur b_B les mutations de a_A pour les éléments satisfaisant le prédicat f. Ceci nécessite le calcul d'un indice j puisque la sélection implique $|b_B| \leq |a_A|$:

$$\forall (i,e)^+ \in \mathcal{M}(a_A) \mid f(e),\, \mathcal{M}(b_A) \leftarrow (j, e)^+$$

$$\forall (i,p)^- \in \mathcal{M}(a_A) \mid f(p),\, \mathcal{M}(b_A) \leftarrow (j, p)^-$$

$$\forall (i,p,e)^= \in \mathcal{M}(a_A) \mid f(p) \wedge f(e),\, \mathcal{M}(b_A) \leftarrow (j, p, e)^=$$

$$\text{avec } j = a_A \rightarrow array() \rightarrow slice(0,i) \rightarrow count(f)$$

La notation $\forall m \in \mathcal{M}(a_A) \mid f(m)$ est un raccourci de $\forall m \in \{m \in \mathcal{M}(a_A) \mid f(m)\}$. L'expression $a_A \rightarrow array()$ retourne le tableau interne du contenant a_A à partir duquel différentes fonctions peuvent être invoquées.

La sélection de références $c_A = a_A \rightarrow select(b_{bool})$ satisfaisant un contentant-prédicat b_{bool} est une opération à deux arguments. Elle est donc mise en équivalence avec l'expression$c_A = ab_{A\times bool} \rightarrow select()$ avec $ab_{A\times bool} = a_A \rightarrow zip(b_{bool})$, où *select* sélectionne les références e de a_A pour chaque couple de $ab_{A\times bool}$ valant $(e, vrai)$. L'implémentation de cette surcharge de *select* est alors réalisée ainsi :

$$\forall (i,(e,s_e))^+ \in \mathcal{M}(ab_{A\times bool}) \mid s_e,\, \mathcal{M}(c_A) \leftarrow (j, e)^+$$

$$\forall (i,(p,s_p))^- \in \mathcal{M}(ab_{A\times bool}) \mid s_p,\, \mathcal{M}(c_A) \leftarrow (j, p)^-$$

$$\forall (i,(p,s_p),(e,s_e))^= \in \mathcal{M}(ab_{A\times bool}) \mid s_p \wedge s_e,\, \mathcal{M}(c_A) \leftarrow (j, p, e)^=$$

$$\forall (i,(p,s_p),(e,s_e))^= \in \mathcal{M}(ab_{A\times bool}) \mid \neg s_p \wedge s_e,\, \mathcal{M}(c_A) \leftarrow (j, e)^+$$

$$\forall (i,(p,s_p),(e,s_e))^= \in \mathcal{M}(ab_{A\times bool}) \mid s_p \wedge \neg s_e,\, \mathcal{M}(c_A) \leftarrow (j, e)^-$$

$$\text{avec } j = a_A \rightarrow array() \rightarrow slice(0,i) \rightarrow count(e, s|s)$$

Implémentation par équivalence

Les opérations qui ne font pas partie des 17 opérations de base sont des opérations dérivées implémentées par équivalence avec une ou plusieurs opérations. Par exemple, l'opération $union(b_B)$ est équivalente à la séquence d'opérations $concat(b_B) \rightarrow asSet()$, l'opération *asSet* étant elle-même implémentée sur la base de l'opération de base *distinct*.

L'application d'une équivalence peut impliquer de relâcher les restrictions imposées sur les opérations utilisées dans l'équivalence. Par exemple, l'opération $mappedBy(f)$ est équivalente à l'opération $apply(getSource)$ où $getSource(e)$ renvoie la référence source mise en correspondance avec la référence cible e *via* une opération $map(f)$ préalable. Dans l'exemple de *mappedBy*, les éléments manipulés sont des références, alors que *apply* n'est sensée manipuler que des valeurs.

Performances

Les complexités asymptotiques des opérations actives dépendent en premier lieu de la complexité liée à la gestion des écritures dans les flux de mutation implémentée dans le langage cible. Les contenants utilisant tous un tableau en interne pour stocker leur contenu, on peut considérer que ces complexités en moyenne et au pire des cas sont les suivantes :

<table>
<tr><th>Mutation à opérer</th><th>$\neg\mathcal{U}(b_B)$</th><th>$\mathcal{U}(b_B)$</th></tr>
<tr><td>$\mathcal{M}(b_B) \leftarrow (i,e)^+$</td><td colspan="2">$n$</td></tr>
<tr><td>$\mathcal{M}(b_B) \leftarrow (i,e)^-$</td><td colspan="2">$n$</td></tr>
<tr><td>$\mathcal{M}(b_B) \leftarrow (i,p,e)^=$</td><td>1</td><td>$n$</td></tr>
</table>

Les insertions et les retraits induisent des décalages dans le tableau interne. Les remplacements n'induisent pas de tels décalages, mais nécessitent une vérification préalable pour la gestion des doublons dans le cas des contenants b_B vérifiant $\mathcal{U}(b_B)$. Les complexités moyennes des traitements t_k effectués par les implémentations des opérations peuvent alors être inférées :

<table>
<tr><th></th><th colspan="4">$\neg\mathcal{U}$</th><th colspan="4">$\mathcal{U}$</th></tr>
<tr><th>Opérations</th><th>$n \times t_0$</th><th>t_1</th><th>t_2</th><th>t_3</th><th>$n \times t_0$</th><th>t_1</th><th>t_2</th><th>t_3</th></tr>
<tr><td>when</td><td>1</td><td>n</td><td colspan="2">1</td><td>1</td><td>n</td><td colspan="2">1</td></tr>
<tr><td>concat</td><td colspan="3">n</td><td>1</td><td colspan="4">-</td></tr>
<tr><td>indices</td><td colspan="4">-</td><td colspan="3">n</td><td>1</td></tr>
<tr><td>apply, map,zip, slice, distinct</td><td colspan="3">n</td><td>1</td><td colspan="4">n</td></tr>
<tr><td>intersection, difference, merge</td><td colspan="4">-</td><td colspan="4">n</td></tr>
<tr><td>times, split</td><td colspan="4">n</td><td colspan="4">-</td></tr>
<tr><td>select, sort</td><td colspan="4">n</td><td colspan="4">n</td></tr>
<tr><td>order</td><td>$n \times \log(n)$</td><td colspan="3">n</td><td colspan="4">-</td></tr>
<tr><td>path</td><td>n^2</td><td colspan="3">n</td><td>n^2</td><td colspan="3">n</td></tr>
</table>

Les colonnes $\neg\mathcal{U}$ et $\mathcal{U}$ précisent les complexités pour les opérations devant respectivement gérer et ne pas gérer l'unicité du ou des contenants associés. La colonne $n \times t_0$ donne la complexité de la construction initiale du contenant résultat, les colonnes suivantes donnant les complexités des traitements $t_{k \geq 1}$ de mise à jour du contenant. Ces différentes complexités sont à comparer aux complexités associées aux mutations observées : par exemple, dans l'expression $b_{string} = a_{int} \rightarrow apply(intToString)$, l'insertion de l'entier 123 dans a_{int} est de complexité en n et engendre l'insertion de la chaîne $'123'$ dans b_{string} qui est aussi de complexité n.

Les complexités de $t_{k \geq 1}$ sont souvent du même ordre que la complexité $n \times t_0$, ce qui provient des décalages imposés par les insertions et les retraits. La performance intéressante de l'incrémental se situe principalement dans la modification d'un élément (traitement t_3) : la complexité de t_3 est souvent 1, celle de $n \times t_0$ étant souvent n. Un gain significatif concerne l'opération *path*, très fréquemment utilisée dans une mise en correspondance de modèles, dont la complexité passe de n^2 pour le traitement initial t_0 à n pour les traitements incrémentaux $t_{k \geq 1}$.

Il est possible d'optimiser les performances précédentes en utilisant, pour le tableau interne des contenants, une structure de données différente du simple tableau. Par exemple, l'utilisation d'arbres binaires de segments permet de diminuer significativement les complexités moyennes [80] : le contenu d'un contenant comportant au total n éléments est segmenté en t tableaux hiérarchisés dans un arbre binaire. L'usage des tables de hachage n'est *a priori* pas possible car les indices liés à la représentation par tableau sont utilisées pour la réification respective des fonctions-prédicats et des fonctions-ordres en contenants-prédicats et contenants-ordres.

Si l'évaluation incrémentale des opérations possède de bonnes performances par rapport à un calcul complet, l'emprunte mémoire des contenants entrant en jeu dans une séquence d'opérations n'est pas négligeable. En effet, une séquence d'opérations $op_1 \rightarrow ... \rightarrow op_n$ produit n contenants a_k qui persistent tant que les opérations restent actives, donc pendant toute la durée de vie de l'application.

Passage à un DSL

Cette section donne quelques préconisations à considérer quant à l'élaboration d'un DSL implémentant la théorie $\mathfrak{T}$.

En premier lieu, le DSL doit définir clairement les types permettant de représenter des objets *immuables* et ceux représentant des objets *modifiables*. Les types immuables seront des types élémentaires (au minimum ceux de $\mathbb{E}$) ainsi que les classes représentant des objets immuables. Par exemple, le type immuable *Date* pourrait être présenté par une classe définie comme suit :

```
class Date {
    int annee
    int mois
    int jour
}
```

Les types non immuables définissent des objets dont l'état peut varier dans le temps. Par exemple, la version suivante de classe *Date* représente une date modifiable :

```
class Date {
    one<int> annee
    one<int> mois
    one<int> jour
}
```

Comme il est d'usage en POO, un objet ne devrait être accessible que *via* une référence. Il est ainsi préconisé de ne pas différencier les types valeurs des types références [8]. Par conséquent, aucune notation de type « pointeur » ne devrait être introduite dans le DSL. Les références nulles sont interdites (et donc le mot clé *null* ne devrait pas être défini) car elles doivent être représentées par un contenant de type *opt*.

Les opérateurs d'affectation $=$ et de comparaison $==$ doivent avoir une sémantique précise. Quelque soit le type T, l'opérateur $=$, quant il est utilisé *lors de la déclaration* d'une variable, établit un lien de référence. L'opérateur $=$ peut par contre être utilisé *après* la déclaration d'une variable si T est un type de contenant : dans ce cas, l'affectation $b = a$ s'effectue par valeur, *i.e.* en copiant le contenu de a dans b. L'opérateur $==$ compare quant à lui les valeurs pour T vérifiant $\mathcal{I}(T) \vee \mathcal{C}(T)$, et par référence pour les autres.

En second lieu, les fonctions $b_A = a_A \rightarrow select(f)$ et $b_A = a_A \rightarrow sort(f)$ devraient utiliser les contenants respectivement prédicats et ordres afin de relâcher la contrainte $\mathcal{I}(A)$. Par exemple, le code suivant doit être correct quelle que soit la version de la classe *Date* utilisée parmi les deux proposées plus haut :

```
bag<Date> dates
seq<Date> datesTriees = dates−>sort(d1, d2 | d1.annee < d2.annee)
```

En troisième et dernier lieu, la notation pointée habituelle de la POO devrait être utilisée conjointement à la notation fléchée des opérations actives. La flèche concernera ainsi l'espace de noms du DSL et le point l'espace de noms du modèle défini en utilisant le DSL. La notation pointée correspondra alors à un raccourci de la notation fléchée. Par exemple, le code suivant :

```
seq<int> a
seq<string> b = a−>apply(i | i.toString())
```

devrait pouvoir s'écrire :

```
seq<int> a
seq<string> b = a.toString()
```

8. La théorie $\mathfrak{T}$ introduit cependant une telle différence afin d'éviter toute ambigüité entre l'objet et ses références.

ce qui évitera toute confusion si la fonction *toString* est également définie pour le type *seq*. Ceci se généralise aux fonctions à n arguments. Par exemple, le code suivant qui effectue la conversion des entiers de a en utilisant la base définie dans b :

```
seq<int> a
one<int> b
seq<string> c = a−>zip(b)−>apply(i, j | i.toString(j))
```

devrait pouvoir s'écrire :

```
seq<int> a
one<int> b
seq<string> c = a.toString(b)
```

De même, la notation pointée devrait pouvoir se substituer à la notation fléchée pour l'opération *path*. Par exemple, le chemin suivant qui fournit les noms des enfants d'un ensemble de personnes :

```
oset<Person> a
seq<string> b = a−>path(p | p.children)−>path(p | p.name)
```

devrait pouvoir s'écrire :

```
oset<Person> a
seq<string> b = a.children.name
```

Les mêmes raccourcis devraient être possibles pour les opérateurs unaires et binaires.

L'ensemble de ces préconisations ainsi que la syntaxe utilisée dans les différents exemples précédents correspondent à un sous ensemble du DSL Loa, langage central du cadre logiciel Loa.

2.5.3 Le cadre logiciel Loa

Cette section présente le cadre logiciel Loa qui synthétise la quasi totalité des travaux menés depuis ma thèse. Ce cadre permet de répondre aux quatre problèmes mentionnés dans la conclusion 2.4.4 page 39. Le jeu des 17 opérations actives de base permet de construire toute mise en correspondance de modèles (problème 1) tout en garantissant l'exécution incrémentale de ces correspondances (problème 2). La section qui suit montre comment l'utilisation des opérations actives permet de résoudre des problèmes simples à complexes, problèmes qui ne peuvent être résolus en utilisant les langages de « data binding » proposés par les boîtes à outils récentes. Les sections suivantes illustrent comment le langage Loa peut être utilisé pour définir les différentes composantes d'une application interactive, en permettant notamment la définition des symboles des présentations cibles (problème 3) ainsi que les interacteurs (problème 4).

Opérations actives et IHM

La théorie des opérations actives a été élaborée de manière à garantir que toute situation de mise en correspondance de modèles puisse s'exprimer et s'exécuter *via* des opérations actives. Cette section vise à montrer que l'usage des opérations actives permet de repousser les limites actuelles des mécanismes et langages de « data binding » proposés par les boîtes à outils « RIA » récentes [25].

Trois boîtes à outils, jugées les plus représentatives vis-à-vis des mécanismes de liaison de données qu'elles proposent, ont été sélectionnées. Il s'agit de BO spécifiques au développement des RIA : Flex [74, 112], WPF [99] et JavaFX 1.0 [60]. Le tableau suivant précise quelles sont les fonctionnalités proposées par ces trois BO :

	Flex		**WPF**		**JavaFX**	
	Collection	Singleton	Collection	Singleton	Collection	Singleton
Opérations math.						
select	(√)		√			
sort	(√)	n/a	√	n/a		n/a
Vues multiples			√	√	√	√
apply		√		√	√	√
path		√		√		√
map	√	√	√	√	√	√
Liaison bidir.	n/a	√	n/a	√	n/a	√
+ conversion	n/a		n/a	√	n/a	

Les colonnes « Collection » et « Singleton » différencient les fonctionnalités respectivement présentes pour les collections et les singletons. Les opérations mathématiques ne sont présentes dans aucune boîte à outils. Les opérations pour les vues (*select*, *sort* et possibilité d'avoir plusieurs vues pour une même collection) sont partiellement disponibles ; par exemple les opérations *select* et *sort* sont possibles en Flex, mais elles altèrent le modèle source et, par conséquent, ne permettent pas la définition de vues multiples. L'opération *apply* n'est généralement pas considérée pour les collections. Enfin, la liaison bidirectionnelle incluant une conversion n'est possible qu'avec WPF.

Afin de comparer l'usage de ces trois BO avec une BO intégrant les opérations actives, l'API Flex a été augmentée par une implémentation en ActionScript des opérations actives qui permet leur utilisation conjointe avec les « widgets » fournis par l'API Flex initiale. Trois exemples ont été spécifiquement construits dans le but de montrer les limites des mécanismes de liaison de données des trois BO par rapport à cette BO Flex augmentée.

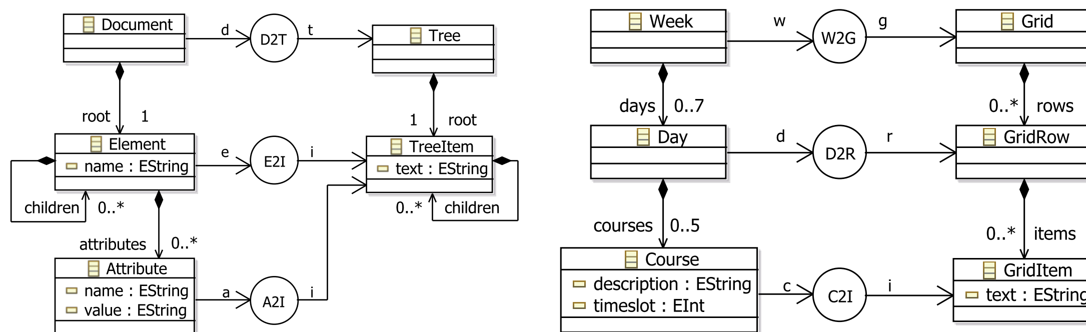

FIGURE 2.5.7 – Exemples

L'exemple 1, dont le modèle est donné sur la partie gauche de la figure 2.5.7, est un éditeur XML simple. Le document XML courant s'affiche dans un *via* la correspondance *D2T*, l'arbre affichant de manière hiérarchique les éléments et les attributs dans des items *via* les correspondances respectives *E2I* et *A2I*. L'exemple 2 est l'exemple du « répertoire » dont le modèle a été présenté figure 2.5.6 mais augmenté par un « widget » champ texte *tf* permettant le filtrage de la liste des contacts basé sur l'initiale de leur nom. L'exemple 3, dont le modèle est donné sur la partie droite de la figure 2.5.7, est une visionneuse de plannings scolaires simples dans une grille dont le modèle est analogue à celui des tables HTML. Les « widgets » utilisés dans ces trois exemples et représentés respectivement par les classes *Tree*, *List*, *TextFileld* et *Grid* sont tous fournis par l'API Flex.

Sur la base du tableau de fonctionnalités précédent, il est possible d'étudier dans quelle mesure les trois exemples peuvent être réalisés avec les BO Flex, WPF et JavaFX. Le tableau suivant synthétise le résultat de cette étude :

		Flex + op. act.	**Flex**	**WPF**	**JavaFX**
Éditeur XML	*sort*	√	(√)	√	
Répertoire	*select/sort*	√	(√)	√	
	select à 2 args.	√			
	liaison bidir. + conv.	√		√	
Planning	*concat*	√			

L'éditeur XML nécessite un tri pour l'affichage des attributs des éléments. Cette fonctionnalité est présente dans WPF, a des capacités limitées avec Flex (modification directe du modèle) et est absente de JavaFX. Le fait d'avoir ajouté un champ texte permettant d'appliquer un filtrage sur la liste des contacts du répertoire implique l'usage de l'opération *select* à deux arguments (utilisation de l'opération *zip*) : aucune des trois boîtes à outils n'offre cette fonctionnalité. Enfin, le modèle Flex des tableaux basé sur la classe *Grid* impose qu'une cellule vide du tableau soit représenté par une instance de la classe *GridItem* (à l'instar des tables HTML). Il est par conséquent nécessaire de lier les plages horaires disponibles avec des *GridItems*, et donc de calculer ces plages. Ce calcul fait notamment appel à l'opération *concat*, absente des trois boîtes à outils. Le détail des correspondances relatives aux trois exemples proposés peut être trouvé dans [25].

Vues orthogonales et liaisons

Cette section et les suivantes illustrent les principes et la syntaxe du langage Loa sur la base d'un exemple d'application appelée « Graf ». Cette application, présentée sur la partie gauche de la figure 2.5.8, permet l'édition de graphes dans un navigateur Web. L'outil « Node » permet la création, le déplacement et le renommage d'un nœud du graphe, l'outil « Edge » permet la création de sommets reliant deux nœuds, et l'outil « Eraser » permet la suppression d'un nœud et des arcs attenants.

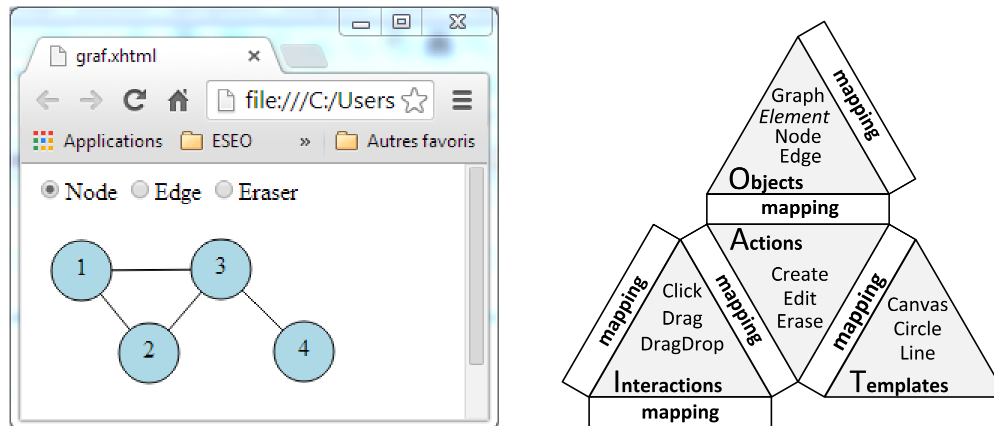

FIGURE 2.5.8 – L'application « Graf » et sa représentation selon le tétraèdre de Loa

Comme le précise l'état de l'art, les modèles conceptuels sont généralement fondés soit sur MVC ou sa variante MVC2, soit sur le concept d'interacteur. Le modèle conceptuel de Loa considère les liens entre modèles et vues (MVC2) ainsi que les liens entre interactions et actions (interacteurs).

La partie droite de la figure 2.5.8 précise comment le *tétraèdre* de Loa décompose toute application en quatre composantes essentielles, chaque face qualifiant un aspect précis de l'application [28]. Les *objets* du domaine sont représentés par les classes *Graph*, *Element*, *Node* et *Egde* ; leurs *présentations* utilisent les symboles représentant le dessin dans son ensemble (classe *Canvas*) et

les différents éléments qui le constitue (classes *Circle* et *Line*) ; les *interactions* entrant en jeu sont le double-clic (classe *Click*), le glisser et le glisser-déposer (classes *Drag* et *DragDrop*) ; ces interactions sont transformées en *actions* sur les objets : création (classe *Create*), édition du nom de nœud (classe *Edit*), et suppression d'un nœud (classe *Erase*).

Chacune des faces du tétraèdre est liée à ses trois faces adjacentes par des relations de correspondance (ou « mapping »). Par exemple, une correspondance entre un objet et un symbole précise comment l'objet est présenté à l'utilisateur *via* un symbole. Une correspondance entre une interaction et une action définit un interacteur qui transforme l'interaction en action. A son tour, une action est liée à un objet du domaine ou à un symbole par une correspondance qui précise comment l'objet ou le symbole est modifié par la correspondance. Le DSL Loa permet de décrire le contenu de chacune des quatre faces du tétraèdre de Loa ainsi que leurs correspondances.

Processus de conception

L'utilisation du cadre Loa suit un processus, représenté figure 2.5.9, qui guide le développeur lors de la conception des différentes composantes de l'application représentées par les quatre faces du tétraèdre ainsi que leurs correspondances [29].

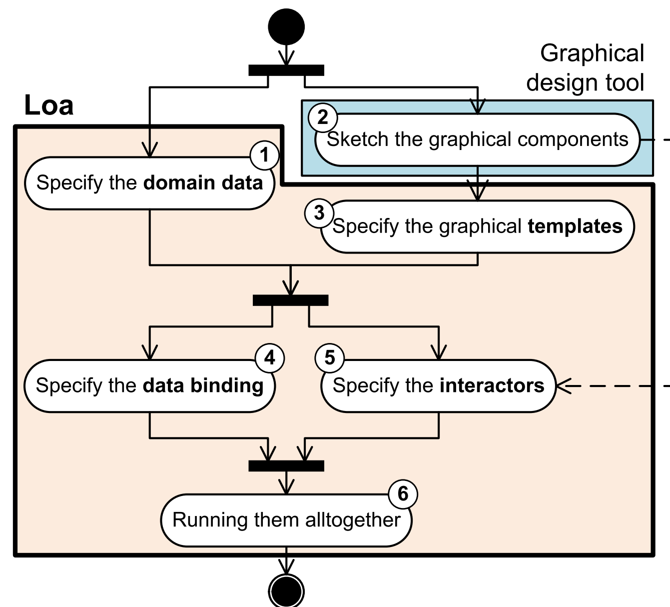

FIGURE 2.5.9 – Processus de conception avec Loa

Le processus commence une première phase constituée des étapes ①, ② et ③ relatives aux faces du tétraèdre. L'étape ① vise à construire le modèle des objets du domaine tandis que l'étape ② consiste à réaliser un prototype de l'interface graphique. Ces deux premières étapes peuvent être menées en parallèle, bien qu'il soit préférable d'identifier les objets du domaine avant de réaliser un prototype de l'interface graphique. L'étape ③ formalise les différents éléments du prototype de l'interface en symboles graphiques (ou « templates »). La modélisation des interactions et des actions est déjà réalisée par les classes fournies avec l'API Loa : elle ne constitue donc pas une étape de cette phase.

La seconde phase modélise les liens de correspondance entre les différentes faces du tétraèdre. L'étape ④ définit les correspondances entre les objets, les symboles et les actions. L'étape ⑤ précise les interacteurs proposées par l'application, sur la base du prototype proposé par l'étape ②. Le processus se termine par l'étape ⑥ qui fournit le programme principal de l'application.

Modèle des objets du domaine

Le code suivant définit le modèle des objets du domaine de l'application « Graf » (étape ①) :

```
class Graph { set<Element> elements }

abstract class Element { }

class Edge extends Element {
   opt<Node> start :: Node.outgoing
   opt<Node> end   :: Node.incoming
}

class Node extends Element {
   one<string> label
   one<int> x
   one<int> y
   set<Edge> outgoing :: Edge.start
   set<Edge> incoming :: Edge.end
}
```

Les contenants de type *set*, *opt* et *one* sont utilisés pour représenter les propriétés (attributs et relations) modifiables des classes *Graph*, *Node* et *Edge*. Le symbole :: permet de déclarer les relations opposées. Par exemple, la relation *Edget.start* est la relation opposée de *Node.outgoing* (et vice versa), si bien que l'ajout d'un nœud n dans la relation $e.start$ d'un sommet e induit l'ajout automatique de e dans la relation $n.outgoing$.

Modèle des présentations

L'interface graphique de l'application est définie par la classe racine *Canvas* suivante (étape ③) :

```
class Canvas {
   one<bool> nodeTool = true
   one<bool> edgeTool
   one<bool> eraserTool
   set<Shape> topLayer
   set<Shape> bottomLayer
   <html>
      <body>
         <input type='radio' name='tool' checked='{{nodeTool}}'/>Node
         <input type='radio' name='tool' checked='{{edgeTool}}'/>Edge
         <input type='radio' name='tool' checked='{{eraserTool}}'/>Eraser
         <svg xmlns='www.w3.org/2000/svg'>
            {bottomLayer}
            {topLayer}
         </svg>
      </body>
   </html>
}
```

Les trois premières propriétés sont des attributs booléens représentant l'état des sélecteurs des trois outils « Node », « Edge » et « Eraser ». Les deux propriétés suivantes représentent des relations définissant le contenu « avant » (*topLayer*) et « arrière » (*bottomLayer*) utilisés respectivement pour l'affichage des nœuds et des sommets.

Le code HTML qui suit la définition des propriétés définit le symbole graphique représentant l'application dans son ensemble. Ce symbole est constitué de trois boutons radio HTML suivis d'une zone de dessin SVG. Il utilise la syntaxe étendue du langage XML qui permet de contrôler le contenu du symbole une fois ce dernier instancié en un fragment DOM. Les accolades simples délimitent un fragment DOM qui contiendra ce qui est spécifié dans l'expression entre accolades. Par exemple, l'expression *{bottomLayer}* précise que le contenu du contenant *bottomLayer* sera placé à l'endroit même des accolades. Lorsque des accolades doubles sont utilisées, le lien entre le contenant Loa et le texte DOM associé est bidirectionnel. Par exemple, l'expression *{{nodeTool}}*

précise que la propriété booléenne *nodeTool* définit l'état du premier bouton radio, cet état pouvant être fixé soit par le canvas lui-même afin de préciser quel bouton est sélectionné (lien Loa→DOM), soit par l'utilisateur lorsqu'il appuie sur un bouton (lien DOM→Loa).

La classe de base *Shape* définit les données communes aux formes géométriques considérées :

```
abstract class Shape {
   one<int> x0
   one<int> y0
   one<string> stroke = 'black'
}
```

Elle est utilisée par la classe *Circle* comme suit :

```
class Circle extends Shape {
   opt<string> text
   <g transform={'translate(' + x0 + ',' + y0 + ')'} >
      <circle r='20' stroke={stroke} fill ='lightblue'/>
      <text y='3' text-anchor='middle'>
         {text}
      </text>
   </g>
}
```

La classe *Line* est définie de manière analogue à la classe *Circle*. L'ensemble de ces quatre classes *Canvas*, *Shape*, *Circle* et *Line* formalise ainsi le prototype de l'interface réalisée en HTML par le « designer » lors de l'étape ②, en extrayant de ce prototype les éléments graphiques de base de manière à en faire des symboles.

Modèle des actions et des interactions

La définition des interactions et des actions s'effectue également *via* des classes en Loa. Cependant, ces classes ne sont pas à définir par le développeur car elles sont fournies par l'API standard de Loa. Par exemple, les classes *Click* et *Create* sont ainsi prédéfinies :

```
class Click {
   one<int> button = 1
   one<int> count = 1
   one<class> type
   opt<int> x
   opt<int> y
   opt<object> target
}
```

```
class Create {
   one<class> type
   opt<object> parent
   opt<property> relation
   opt<object> instance
}
```

La propriété *Click.type* précise sur quel type d'objet (côté présentation ou objet métier) le clic doit être considéré. Si un tel clic survient, la propriété *Click.target* précise alors l'objet sur lequel le clic est survenu. La propriété *Create.type* précise quelle classe doit être instanciée pour créer l'objet en question, cet objet étant alors placé dans le contenant *Create.instance*. La propriété *Create.parent* précise alors, conjointement à la propriété *Create.relation*, dans quel contenant l'objet créé doit être placé. Ces deux classes *Click* et *Create* sont utilisées pour définir un interacteur permettant la création d'un nœud. Cet interacteur est défini dans la section suivante.

Correspondances entre les modèles

Les objets du domaine et leurs présentations *via* des symboles sont mis en correspondance en utilisant l'opération active *map* (étape ④), comme l'illustre l'exemple suivant :

```
Canvas G2C(Graph g) {
    Canvas c = new Canvas()
    c.topLayer ::= g.elements->as(Node)->map(N2C)
    c.bottomLayer ::= g.elements->as(Edge)->map(E2L)
    return c
}
```

```
Line E2L(Edge e) {
    Line l = new Line()
    l.x0 ::= e.start.x
    l.y0 ::= e.start.y
    l.x1 ::= e.end.x
    l.y1 ::= e.end.y
    return l
}
```

La fonction *G2C* crée un canvas *c* lié au graphe *g* en y accueillant les nœuds et sommets de *g* dans les couches respectives *topLayer* et *bottomLayer*. Les sommets sont représentés par des lignes mises en correspondance *via* la fonction *E2L*. La mise en correspondance des nœuds avec des cercles est opérée de manière analogue à *E2C* par la fonction *N2C*.

L'interacteur de création de nœud est construit en utilisant la classe *ClickCreate* prédéfinie par Loa, laquelle mixe l'interaction *Click* avec l'action *Create* (étape ⑤), comme suit :

```
1  ClickCreate C2NC(Canvas canvas) {
2      ClickCreate i = new ClickCreate()
3      i.enabled ::= canvas.nodeTool
4      i.target ::= canvas
5      i.click.count = 2
6      i.click.type = Graph
7      i.create.type = Node
8      i.create.parent = i.click.target
9      i.create.relation = Graph.elements
10     opt<Node> createdNode = i.create.instance->as(Node)
11     createdNode.x ::= i.click.x
12     createdNode.y ::= i.click.y
13     return ClickCreate
14 }
```

L'interacteur n'est actif que si l'outil « Node » du canvas est sélectionné (ligne 3). Son interaction *Click* concerne un double clic (ligne 5) sur une instance de la classe *Graph* (ligne 6). Lorsque ce double clic a lieu, une instance de *Node* est créée (ligne 7), est insérée dans le contenant *elements* (ligne 9) du graphe « cliqué » (ligne 8), pour être stockée dans un contenant (ligne 10). Le lien entre la position de l'interaction et celle du nœud créé est finalement établi (lignes 11 et 12).

Programme principal

Le programme principal définit les éléments de départ de l'application, à savoir le graphe à éditer et le canvas permettant son édition :

```
1  void main() {
2      opt<Graph> g = sampleGraph()
3      opt<Canvas> c = g->map(G2C)
4      set<Interactor> i = c->map(C2NC) + ...
5      display(c, i)
6  }
```

Le graphe *g* est ici un graphe exemple construit par la fonction *sampleGraph* (ligne 2). Il est mis en correspondance avec le canvas *c via* la fonction *G2C* (ligne 3). Les interacteurs sont construits et mis en correspondance avec le canvas *via* la fonction *C2NC* et les suivantes (ligne 4). La fonction *display* de l'API Loa permet l'affichage de *c* par la plate-forme Web en prenant en considération les interacteurs associés (ligne 5).

Implémentation sur différentes plates-formes

Les premières implémentations des opérations actives ont été réalisées sur la plate-forme Kermeta pour de premiers essais [19], puis sur la plate-forme Flex pour illustrer l'intérêt des opérations actives dans le contexte du développement des interfaces graphiques [25]. Ces implémentations étaient cependant dépourvues de DSL. Deux implémentations ont alors suivi afin de combler cette lacune. Elles permettent à une application écrite en Loa de s'exécuter sur la plate-forme JVM ou sur la plate-forme Web. La figure 2.5.10 donne l'architecture commune à ces deux implémentations.

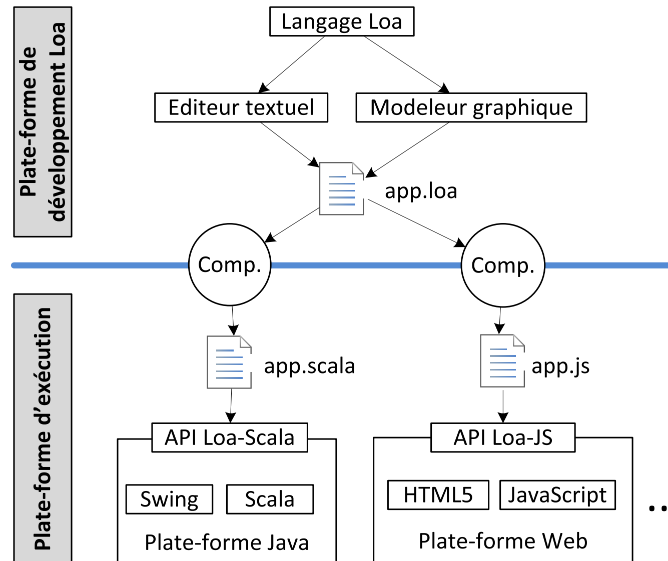

FIGURE 2.5.10 – Implémentation sur plates-formes Java et HTML5

L'implémentation en Scala définit le DSL sous la forme d'un DSL *interne*, c'est-à-dire construit comme une *extension* du langage Scala [29]. Ce DSL est ainsi sensiblement différent du DSL Loa ciblé puisqu'il est contraint par la syntaxe du langage Scala. De plus, l'implémentation du DSL interne est alors mixée à celle de l'API Loa-Scala, rendant très complexe sa compréhension. Cette implémentation a cependant permis de tester l'ensemble des opérations actives de la théorie $\mathfrak{T}$, ainsi que la compatibilité de Loa avec les « widgets » proposées par la bibliothèque Swing de Java.

Le langage Loa est aujourd'hui en cours de finalisation. Il est défini sous la forme d'un DSL *externe* et d'un compilateur qui transforme le code Loa en code JavaScript. Ce code fait appel à l'API Loa-JS qui implémente la théorie $\mathfrak{T}$, le mécanisme d'instanciation des symboles Loa, ainsi qu'un jeu d'interacteurs, permettant aux modèles Loa de pouvoir s'exécuter sur la plate-forme du Web [28].

Chapitre 3

Conclusion et perspectives

Ce chapitre clôt le mémoire par une synthèse des travaux qui regroupe le parcours chronologique et les contributions qui l'ont jalonné (section 3.1), puis par les différentes perspectives envisagées autour du cadre logiciel Loa en lien avec différents partenaires (section 3.2).

3.1 Synthèse des travaux

La contribution scientifique des travaux présentés dans ce mémoire a été réalisée en deux phases chronologiques : une phase appliquée au domaine de l'ingénierie des systèmes interactifs centrée sur les notions de modèles conceptuels et de boîtes à outils, suivie d'une phase abordant cette problématique de l'ingénierie des SI sous l'angle des modèles au sens de l'IDM .

La première phase, de 2001 à 2008, a débuté par ma thèse « Espaces de travail interactifs et collaboratifs : vers un modèle centré sur les documents et les instruments d'interaction » [13], laquelle propose le modèle conceptuel DPI pour la construction de SI [20, 14] ainsi qu'une boîte à outils OpenDPI qui a permis d'évaluer la pertinence du modèle proposé [21, 22]. Les travaux de cette thèse ont été poursuivis à l'ESEO au sein du GRI dans l'axe « Ingénierie des Systèmes Interactifs ». Le modèle conceptuel DPI a été enrichi sur deux volets. Le premier concerne le lien de type « observable-observateur » proposé dans DPI pour lier les composantes D et P : ce lien a été enrichi par le concept de transformation active implémenté dans eXAcT [16]. Le second volet aborde le partage des documents et de leurs présentations sous l'angle des « EventPoints », lesquels permettent la réplication de tout ou partie des documents structurés ou/et des graphes de scène définissant les présentations de ces documents [15]. Ces deux volets ont été intégrés dans la boîte à outil DoPIdom, laquelle synthèse les implémentations OpenDPI, eXAcT et EventPoints [17].

La thèse d'Arnaud Blouin, dont le titre initial était « Transformation de documents dans le contexte des systèmes interactifs : une approche basée sur la correspondance entre documents sources et présentations cibles », avait pour objectif de de formaliser l'idée de transformation active dans le contexte du modèle DPI. Arnaud a dans un premier temps proposé un langage de correspondance, Malan, qui permet de décrire une transformation sous la forme d'une correspondance de schémas [39]. Dans un second temps a été proposé un deuxième DSL, Malai, qui permet de décrire les composantes interactives d'un SI [37]. L'usage conjoint des deux langages Malan et Malai permet la modélisation de toutes les composantes d'un SI [38]. La thèse d'Arnaud, co-encadrée avec l'équipe ICLN du LERIA, a été l'occasion d'entamer une coopération avec l'Université d'Angers. Le concept de correspondance mis en avant dans cette thèse nous a fait réaliser qu'il était pertinent de proposer une approche de l'ingénierie des IHM centrée sur les schémas, au sens des bases de données et des documents XML, c'est-à-dire sur la notion centrale de *modèle*. La thèse a ainsi été renommée en « Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données », caractérisant l'entrée dans la seconde phase, à savoir « l'approche fondée sur certains principes de l'IDM ».

La seconde phase, de 2008 à aujourd'hui, s'est concrétisée par une coopération très étroite avec l'équipe DiverSE (ex Triskell) de l'IRISA (INRIA Rennes), cette équipe apportant son expertise en IDM et l'ESEO son expertise en IHM. Le responsable de cette équipe a fait partie du jury de la thèse d'Arnaud en tant que rapporteur et je suis devenu officiellement collaborateur extérieur de l'IRISA et INRIA en 2013. Conjointement aux travaux d'Arnaud, le concept d'opération active a émergé au début de cette seconde phase, une opération active étant, au sens de la programmation réactive, une opération usuelle sur les collections pour laquelle une implémentation incrémentale est garantie [24]. Ce concept permet de définir des opérations permettant l'exécution directe de correspondances entre les différentes composantes d'un SI, en dépassant ce qu'il est possible d'exprimer avec les langages et mécanismes de «data binding » existants [25]. Le cadre logiciel Loa, articulé autour du DSL Loa, unifie les travaux autour des opérations actives avec ceux réalisés pendant la première phase [29], et aborde le concept d'interacteur et d'instrument sur la même base des opérations actives [28]. La réalisation d'un projet financé MPIA 2011, intitulé « Outils pour la Conception de Systèmes d'Informations Interactifs », a permis de consolider la première implémentation de ce cadre logiciel.

Cette seconde phase se conclut par le montage de deux réponses à l'appel à projets « ANR générique 2014 ». Le premier projet, NEXT-CITYGEN, est un projet JCJC porté par l'équipe DiverSE. Je suis engagé dans ce projet de part mon statut de collaborateur extérieur IRISA. Le second projet, VIFORM, est un projet collaboratif en partenariat public et privé qui fait suite à une coopération engagée en 2005 avec le LIUM au travers d'une réponse à un appel à projet régional dans le domaine des TICE.

3.2 Perspectives

Les perspectives à court terme des travaux autour de Loa concernent la rédaction d'articles et la réalisation d'outils. Les différentes contributions non encore publiées du langage Loa sont les suivantes : approche des interacteurs centrée sur le seul usage des opérations actives ; la théorie complète des opérations actives, indépendante de tout DSL ; le langage Loa en tant que DSL centré sur les modèles exécutables et les opérations actives ; l'implémentation de Loa sur la plate-forme du Web. Cet ensemble de publications accompagnera la finalisation du langage Loa ainsi que son outillage.

Ce travail à court terme constituera la base des cinq perspectives à moyen et long termes présentées dans les sections qui suivent, de la plus prioritaire à la moins prioritaire.

3.2.1 Passerelles entre les modèles

L'équipe de recherche TRAME à laquelle j'appartiens travaille autour du thème précis des passerelles entre modèles. Le terme de passerelle englobe différentes approches et langages développés par l'équipe qui permettent la mise en relation de modèles : le langage de synchronisation de modèles Loa, le langage de communication entre modèles Protocola [1], et le langage de transformation de modèles ATL [72, 71]. Ces trois langages adressent la thématique générale de la correspondance et de la composition de modèles sous des angles complémentaires. Ils visent tous à faciliter le développement des différentes parties des systèmes, qu'ils soient embarqués, mobiles ou d'information, et donc à en réduire le coût.

La figure 3.2.1 montre comment les trois langages peuvent coopérer pour modéliser et implémenter un système matériel et logiciel complet. Loa assure la mise en correspondance des interacteurs, des objets métiers et des présentations. Protocola permet d'alimenter les interacteurs en transformant le flot des évènements des différents périphériques d'entrée en interactions. Il permet également l'affichage des présentations représentées par des graphes qui peuvent être transformés en arbres de type SVG ou en un flot d'appels de fonctions de dessin. Il réalise enfin la transformation des

1. Non encore publié à ce jour.

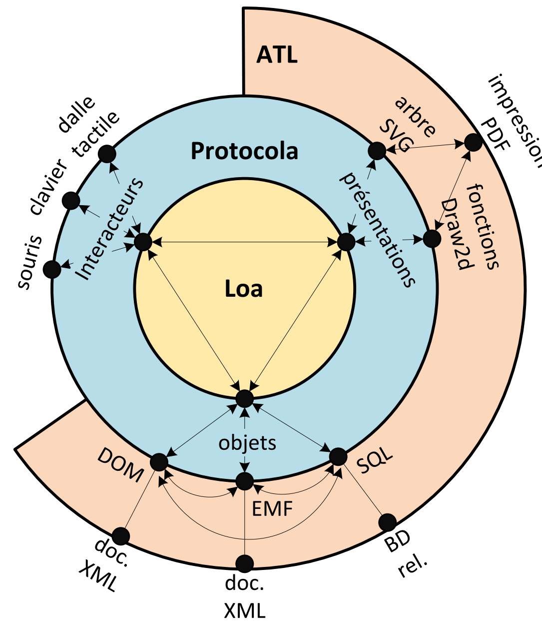

FIGURE 3.2.1 – Utilisation conjointe des 3 langages ATL, Loa et Protocola

objets métiers dans des formats tels que DOM, EMF ou SQL liés à des supports de persistance différents (documents XML ou bases de données relationnelles). Le langage ATL réalise, quant à lui, des transformations permettant de passer d'un espace technique à l'autre, par exemple de DOM à EMF ou SQL, ou d'un arbre SVG à un document PDF.

La première perspective de mes travaux est de participer à la réalisation d'une telle infrastructure pour un démonstrateur à définir. Ceci constituerait une vitrine du savoir-faire de l'équipe TRAME. Ce travail pourrait être amorcé dès la rentrée prochaine par un ensemble de PFE définissant un projet commun entre les deux départements « Informatique » et « Informatique Industrielle ».

3.2.2 Diversité des plates-formes d'exécution

Le langage Loa permet la construction des modèles pour les différentes composantes entrant en jeu dans la conception des SI. Ces modèles ont la capacité d'être directement exécutables sur la plate-forme ciblée et de pouvoir se synchroniser. Cette approche bénéficie de l'avantage inhérent à l'IDM, à savoir la possibilité d'adresser différentes plates-formes et donc de factoriser les coûts inhérents aux développements multi plates-formes.

Le cadre logiciel Loa est constitué d'un éditeur de codes sources conformes au langage Loa, ainsi que d'un ensemble de compilateurs permettant de transformer le code Loa dans les langages des plates-formes d'exécution ciblées. Actuellement, deux plates-formes d'exécution sont adressées : la plate-forme Web et la plate-forme Java. Le développement d'autres compilateurs pour adresser de nouvelles plates-formes d'exécution (par ex. Android, iOS, Eclipse EMF, ou .NET) est un travail d'ingénierie qui pourrait être réalisé par le biais de projets industriels. A l'heure actuelle, un projet avec le CEA est en cours de construction autour du logiciel Papyrus et de la plate-forme Eclipse.

La problématique de la diversité doit être également abordée sous un angle plus théorique afin de définir des mécanismes qui permettraient de gérer la diversité *a priori* (modèles statiques) mais également *a posteriori* (modèles adaptables, réutilisation de modèles, etc.). Ce travail se fera avec l'équipe de recherche DiverSE de l'IRISA dont la thématique centrale est la gestion de la diversité. L'adaptabilité des IHM à la plate-forme d'exécution a déjà fait l'objet d'un travail en commun entre l'ESEO et l'IRISA [41]. Ce travail se poursuit actuellement au travers du projet ANR NEXT-CITYGEN piloté par l'équipe DiverSE.

3.2.3 Application de contraintes

Le modèle des objets métier peut être complété par un système de contraintes pouvant s'appliquer à certaines propriétés de ces objets. De telles contraintes définissent également un modèle, ce modèle étant orthogonal à celui des objets.

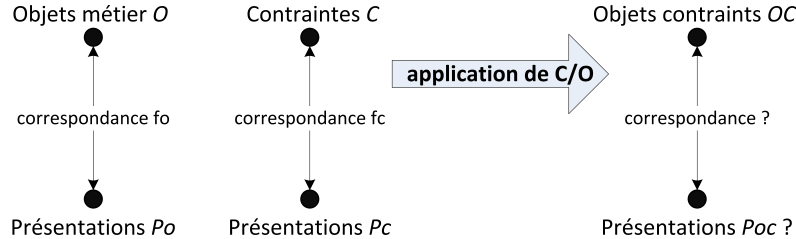

FIGURE 3.2.2 – Projection de contraintes

La figure 3.2.2 présente le problème posé par l'application de contraintes sur des objets métier. Avant une telle application, les objets métier O sont mis en correspondance avec leur présentation P_O *via* la fonction f_O définie par $f_O(O) = P_O$. Selon ce même principe, les contraintes C sont mises en correspondance avec leur présentation P_C *via* la fonction f_C définie par $f_C(C) = P_C$, indépendamment des objets O. Le fait d'appliquer les contraintes C sur les objets O permet à l'utilisateur, *via* des présentations P_{OC}, de constater dans quelle mesure les objets contraints OC respectent ou non les contraintes en question. Le problème soulevé ici est donc un cas particulier de la composition de modèle : la nouvelle correspondance f_{OC} est la composition des correspondances f_O et f_C, et la présentation P_{OC} est la composition des présentations P_C et P_O notée $P_{OC} = P_C \setminus P_O$. L'opération de composition $\setminus$ est appelée projection puisqu'il s'agit de projeter la présentation P_C au dessus d'une présentation existante P_O devant être globalement préservée.

Une coopération entamée depuis un an avec l'équipe MOA du LERIA s'inscrit précisément dans le thème de la projection de contraintes sur un modèle métier. Elle s'est concrétisée par la thèse CIFRE de Mengqiang Yang intitulée « Approche outillée pour la conception d'IHM dédiées à la manipulation de données complexes et contraintes ». Le modèle des objets métiers concernait la description des produits définis par les flux entrant en jeu dans leur fabrication, et les contraintes concernaient des normes environnementales à respecter lors de cette fabrication. La projection consistait donc à vérifier dans quelle mesure les produits respectent de telles normes. Cette coopération se poursuivra à la rentrée prochaine par la réalisation d'un projet conjoint entre les étudiants de l'ESEO et ceux du master 2 de l'université d'Angers. Ce projet s'inscrira probablement dans une coopération régionale entre différentes équipes travaillant sur le thème des contraintes.

Par ailleurs, l'idée de projection d'un modèle orthogonal au modèle métier est également à l'œuvre dans le cadre du projet ANR VIFORM construit avec Le Mans (LIUM et ENSIM). Dans ce projet, il est en effet question de projeter l'activité d'apprenants réalisée *via* une plate-forme TICE, sur le scénario pédagogique construit par les encadrants en amont de cette activité. Le résultat d'une telle projection donne une vue précise qui permet d'évaluer pour les encadrants et les apprenants l'avancée *in situ* du scénario pédagogique, et ainsi de pouvoir faire évoluer la formation.

3.2.4 Documents composites et partage

La plate-forme du Web a la particularité d'utiliser le langage standard HTML5 qui permet de mixer textes et graphiques, là où les bibliothèques pour les IHM se focalisent sur la fourniture de composants graphiques, les « widgets ». Mes différents travaux ont jusqu'ici ignoré le versant texte du contenu des présentations, bien que ma thèse ait porté sur les documents au sens général, *i.e.* les documents composites. Il me semble ainsi pertinent d'analyser dans quelle mesure le cadre logiciel Loa permettrait l'édition de documents composites et donc la construction de systèmes auteurs.

Le partage des objets métier est une question qui a été abordée de manière assez exhaustive dans ma thèse. Le concept de points d'évènement a été proposé afin de permettre le partage de tout ou partie d'un arbre d'objets ou/et d'un graphe de scène fournissant la présentation de ces objets aux utilisateurs [15]. Ceci autorise l'édition collaborative, en temps réel ou en temps différé, de documents, à l'instar de ce que l'on peut faire avec les applications Web telles que celles proposées par Google docs. Les « EventPoints » ont été intégrés dans la boîte à outils DoPIdom mais sont absents du cadre logiciel Loa.

Aucun projet n'a été engagé après la thèse de 2004 sur ce domaine. L'équipe MINT de l'INRIA Lilles pourrait être un partenaire pour ce projet, notamment pour la facette partage de document. De manière complémentaire, l'équipe IntuiDoc de l'IRISA pourrait être un partenaire complémentaire pour la facette document composite.

3.2.5 Visualisation d'information

L'utilisation de modèles « actifs » (au sens des opérations actives qui les mettent en correspondance) a un impact non négligeable sur la quantité d'information devant être conservée en mémoire pendant toute la durée de vie de l'application. Cette emprunte mémoire peut devenir élevée dans le contexte des corpus volumineux entrant en jeu dans le domaine de la « visualisation d'information ». Cependant, l'approche définie par Loa me semble pertinente pour ce type d'application puisqu'elle permet, d'une part, de définir des présentations graphiques complexes et, d'autres part, de définir des liens complexes entre les objets métier et de telles présentations.

Une première réflexion a été menée avec l'équipe DiverSE sur le sujet autour du « slicing » de modèle [35, 36]. Le « slicing » de modèle vise à définir une approche permettant de fournir différents points de vue d'un modèle central, ce modèle ne pouvant être humainement compris dans sa globalité. Dans le cadre de PFE menés en amont du projet VIFORM, les étudiants ont mis en œuvre un « zoom sémantique » en utilisant Loa, le zoom sémantique pouvant être considéré comme un cas particulier de « slicing ».

Une étude plus approfondie de l'application de l'approche des opérations actives au domaine de la visualisation d'information pourrait être envisagée par une coopération avec l'équipe Aviz de l'INRIA (Paris Orsay) dirigée par Jean-Daniel Fekete, membre du conseil scientifique de l'ESEO.

Bibliographie

[1] J.M. Almendros-Jiménez and L. Iribarne. An extension of UML for the modeling of WIMP user interfaces. *Journal of Visual Languages & Computing*, 19(6) :695–720, 2008.

[2] Apple. Opendoc technical summary. Technical documentation, Apple Computer Inc., 1994.

[3] B. E. Backlund. Ooe : A compound document framework. *ACM SIGCHI Bulletin*, 29(1) :68–75, Jan. 1997.

[4] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann. The JANUS application development environment – generating more than the user interface. In *Computer-Aided Design of User Interfaces*, 1996.

[5] E. Barboni, J.-F. Ladry, D. Navarre, P. Palanque, and M. Winckler. Beyond modelling : An integrated environment supporting co-execution of tasks and systems models. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*, pages 165–174. ACM, 2010.

[6] L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, . Seacord, S. Sheppard, and M. R. Szczur. A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1) :32–37, 1992.

[7] M. Beaudouin-Lafon. Interaction instrumentale : de la manipulation directe à la réalité augmentée. In *Actes de la Conférence Francophone sur l'Interaction Homme-Machine (IHM'97)*. Cépaduès Éditions, 1997.

[8] M. Beaudouin-Lafon. Instrumental interaction : An interaction model for designing post-wimp interfaces. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'00)*, pages 446–453. ACM Press, 2000.

[9] M. Beaudouin-Lafon. Novel interaction techniques for overlapping windows. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'01)*, pages 153–154. ACM Press, 2001.

[10] M. Beaudouin-Lafon. Designing interaction, not interfaces. In *Proceedings of Conference on Advanced Visual Interfaces (AVI' 20'04)*, pages 15–22. ACM, 2004.

[11] O. Beaudoux. Paradigmes et éléments architecturaux d'une boîte à outils post-wimp. Rapport technique, ESEO-LRI, 2000.

[12] O. Beaudoux. DoPIdom : Une boîte à outils pour la conception d'interfaces centrées sur les documents XML. In *Actes de la conférence francophone sur l'Interaction Homme-Machine (IHM'04)*, pages 187–190. ACM Press, 2004.

[13] O. Beaudoux. *Espaces de travail interactifs et collaboratifs : vers un modèle centré sur les documents et les instruments d'interaction*. Thèse de doctorat, LRI, 2004.

[14] O. Beaudoux. Un modèle de composants (inter)actifs centré sur les documents. *Revue Information, Interaction, Intelligence (RI3)*, 4(1) :41–58, 2004.

[15] O. Beaudoux. Event points : Annotating XML documents for remote sharing. In *Proceedings of the 2005 ACM Symposium on Document Engineering (DocEng'05)*, pages 159–161. ACM Press, 2005.

[16] O. Beaudoux. XML active transformation (eXAcT) : Transforming documents within interactive systems. In *Proceedings of the 2005 ACM Symposium on Document Engineering (DocEng'05)*, pages 146–148. ACM Press, 2005.

[17] O. Beaudoux. DoPIdom : Une approche de l'interaction et de la collaboration centrée sur les documents. In *Actes de la 18ème conférence francophone sur l'Interaction Homme-Machine (IHM'06)*, pages 19–26. ACM Press, 2006.

[18] O. Beaudoux. exact : A novel approach to transformations for rich internet applications. Technical report, ESEO, 2007.

[19] O. Beaudoux. Active transformation with kermeta. Kermeta Day 2009 (workshop), 2009.

[20] O. Beaudoux and M. Beaudouin-Lafon. DPI : A conceptual model based on documents and interaction instruments. In *People and Computer XV - Interaction without frontier (Joint proceedings of HCI'01 and IHM'01)*, pages 247–263. Springer-Verlag, 2001.

[21] O. Beaudoux and M. Beaudouin-Lafon. OpenDPI : A toolkit for developing document-centered environments. In *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'05)*, volume 5, pages 39–47. ICEIS Press, 2005.

[22] O. Beaudoux and M. Beaudouin-Lafon. OpenDPI : A toolkit for developing document-centered environments. In *Enterprise Information Systems VII*, pages 231–239. Springer, 2006.

[23] O. Beaudoux and A. Blouin. Linking data and presentations : from mapping to active transformations. In *Proceedings of the 2010 ACM symposium on Document engineering (DocEng'10)*, pages 107–110. ACM Press, 2010.

[24] O. Beaudoux, A. Blouin, O. Barais, and J.-M. Jézéquel. Active operations on collections. In *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, pages 91–105. Springer, 2010.

[25] O. Beaudoux, A. Blouin, O. Barais, and J.M. Jézéquel. Specifying and implementing ui data bindings with active operations. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'11)*, pages 127–136. ACM press, 2011.

[26] O. Beaudoux, A. Blouin, and S. Hammoudi. Correspondances et transformations actives dédiées aux ihm. In *Actes des èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM'09)*, pages 115–130, 2009.

[27] O. Beaudoux, A. Blouin, and J.-M. Jézéquel. Using model driven engineering technologies for building authoring applications. In *Proceedings of the 2010 ACM symposium on Document engineering (DocEng'10)*, pages 279–282. ACM Press, 2010.

[28] O. Beaudoux, M. Clavreul, and A. Blouin. Binding orthogonal views for user interface design. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO'13)*, page in press. ACM press, 2013.

[29] O. Beaudoux, M. Clavreul, A. Blouin, M. Yang, O. Barais, and J.M. Jézéquel. Specifying and running rich graphical components with loa. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems (EICS'12)*, pages 169–178. ACM press, 2012.

[30] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1) :70–118, 2005.

[31] J. Bézivin and I. Kurtev. Model-based technology integration with the technical space concept. In *In Metainformatics Symposium*, 2005.

[32] E. A. Bier, M. C. Stone, K. Fishkin, W. Buxton, and T. Baudel. A taxonomy of see-through tools. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'94)*, pages 358–364. ACM Press, 1994.

[33] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses : the see-through interface. In *Proceedings of the Conference on Computer Graphics and Interactive Techniques*, pages 73–80. ACM Press, 1993.

[34] A. Blouin. *Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données*. PhD thesis, Université d'Angers, 2009.

[35] A. Blouin, B. Combemale B. Baudry, and O. Beaudoux. Modeling model slicers. In *Proceedings of the 14th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*, pages 62–76. Springer, 2011.

[36] A. Blouin, B. Combemale B. Baudry, and O. Beaudoux. Kompren : Modeling and generating model slicers. *Software and Systems Modeling (SoSym)*, nov. 2012.

[37] A. Blouin and O. Beaudoux. Malai : un modèle conceptuel d'interaction pour les systèmes interactifs (fr). In *Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine (IHM'09)*, pages 129–138. ACM Press, 2009.

[38] A. Blouin and O. Beaudoux. Improving modularity and usability of interactive systems with Malai. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)*, pages 115–124. ACM Press, 2010.

[39] A. Blouin, O. Beaudoux, and S. Loiseau. Malan : A mapping language for the data manipulation. In *Proceedings of the 2008 ACM symposium on Document engineering (DocEng'08)*, pages 66–75. ACM Press, 2008.

[40] A. Blouin, O. Beaudoux, and S. Loiseau. Un tour d'horizon des approches pour la manipulation des données du web. *Document Numérique*, 11(1-2/2008) :63–83, 2008.

[41] A. Blouin, B. Morin, G. Nain, O. Beaudoux, P. Albers, and J.M. Jézéquel. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'11)*, pages 85–94. ACM press, 2011.

[42] A. Blouin, G. Nain, and O. Beaudoux. Malan et malai pour la conception de systèmes interactifs : perspectives d'intégration dans kermeta. IDM 2010 (atelier IDM-IHM), 2010.

[43] M. Blumendorf, G. Lehmann, S. Feuerstack, and S. Albayrak. Executable models for human-computer interaction. In *Proceedings of DSV-IS2008*, 2008.

[44] K. Brockschmidt. *Inside OLE, Second Edition*. Microsoft Press, 1995.

[45] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'98)*, pages 95–100. ACM Press, 1998.

[46] G. Calvary, J. Coutaz, . Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting With Computers*, 15(3) :289–308, 2003.

[47] D. Canfield, S. Charles, and H. Irby. Xerox star live demonstration. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'98)*, page 17. ACM Press, 1998.

[48] S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94)*, pages 195–204. ACM Press, 1994.

[49] S. Chatty. Engineering interactive systems. chapter Programs = Data + Algorithms + Architecture : Consequences for Interactive Software Engineering, pages 356–373. Springer-Verlag, Berlin, Heidelberg, 2008.

[50] S. Chatty. Supporting multidisciplinary software composition for interactive applications. In *Proceedings of the 7th International Conference on Software Composition (SC'08)*, pages 173–189, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] D. Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cumming, 1995.

[52] J. Coutaz. Pac-ing the architecture of your user interface. In *Proceedings of the Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, pages 15–32. Springer Verlag, 1997.

[53] J. Coutaz. User interface plasticity : Model driven engineering to the limit ! In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*, pages 1–8. ACM Press, 2010.

[54] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, pages 407–426. ACM, 2011.

[55] A.-M. Déry-Pinna, C. Joffroy, A. Occello, P. Renevier, and M. Riveill. L'ingénierie dirigée par les modëles au coeur d'un framework d'aide á la composition d'interfaces utilisateurs. In *Cinquièmes Journées sur l'Ingénierie Dirigée par les Modèles*, 2009.

[56] R. Dewsbury. *Google Web Toolkit Applications*. Addison-Wesley Professional, 2008.

[57] P. Dragicevic and J.D. Fekete. Input device selection and interaction configuration with icon. In *People and Computer XV - Interaction without Frontier (Joint proceedings of HCI 2001 and IHM 2001)*, pages 543–448. Springer Verlag, 2001.

[58] P. Drix. *XSLT fondamental*. Eyrolles, 2002.

[59] T. Elwert and E. Schlungbaum. Modelling and generation of graphical user interfaces in the TADEUS approach. In *DSVIS'95, Proceedings of the Eurographics Workshop*, 1995.

[60] R. Field. JavaFX language reference (chapter 7 - Data binding). `http://openjfx.java.sun.com/current-build/doc/reference/ch07s01.html`.

[61] J. Foley, C. Gibbs, and S. Kovacevic. A knowledge-based user interface management system. In *CHI '88 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 67–72. ACM, 1988.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[63] J.J. Garrett. Ajax : A new approach to web applications. Technical report, Adaptative Path, 2005.

[64] G. Gauffre, E. Dubois, and R. Bastide. Domain-specific methods and tools for the design of advanced interactive techniques. *Lecture Notes in Computer Science*, 5002/2008 :65–76, 2008.

[65] P. Gorny. Expose : an hci-counseling tool for user interface designers. *SIGCHI Bulletin*, 27(2) :35–37, 1995.

[66] T. Griffiths, J. Mckirdy, N. Paton, J. Kennedy, R. Cooper, P. Barclay, C. Goble, P. Gray, M. Smyth, A. West, and A. Dinn. An open model-based interface development system : The teallach approach. In *In Supplementary Proceedings of DS-VIS'98*, pages 32–49. Eurographics, 1998.

[67] P.J. Hayes, P.A. Szekely, and R.A. Lerner. Design alternatives for user interface management sytems based on experience with cousin. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '85)*, pages 169–175. ACM, 1985.

[68] K. Hinckley and M. Sinclair. Touch-sensing input devices. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'99)*, pages 223–230. ACM Press, 1999.

[69] C. Joffroy. Composition d'interfaces homme-machine dirigée par la composition du noyau fonctionnel. In *Proceedings of the 21th international conference on Association Francophone d'Interaction Homme-Machine (IHM'09)*, 2009.

[70] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. Irby, M. Beard, and K. Mackey. The xerox star : A retrospective. *IEEE Computer*, 22(9) :11–29, sep 1989.

[71] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer Berlin Heidelberg, 2006.

[72] F. Jouault and I. Kurtev. On the interoperability of model-to-model transformation languages. *Science of Computer Programming*, 68(3) :114–137, 2007.

[73] M. Kay. *XSLT 2.0 Programmer's Reference*. Wrox, 2004.

[74] C. Kazoun and J. Lott. *Programming Flex 2*. O'Reilly, 2007.

[75] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Objet Oriented Programming*, pages 26–49, aug 1988.

[76] G. Kurtenbach and W. Buxton. The limits of expert performance using hierarchic marking menus. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'93)*, pages 482–487. ACM Press, 1993.

[77] G. Kurtenbach, G. Fitzmaurice, T. Baudel, and B. Buxton. The design of a gui paradigm based on tablets, two-hands, and transparency. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'97)*, pages 35–42. ACM Press, 1997.

[78] S. Lepreux, A. Hariri, J. Rouillard, D. Tabary, J.-C. Tarby, and C. Kolski. Towards multimodal user interfaces composition based on usixml and mbd principles. In *HCII 2007*, 2007.

[79] F. Lonczewski and S. Schreiber. The fuse-system : an integrated user interface design environment. In *Proceedings of Computer-Aided Design of User Interfaces (CADUI'96)*, 1996.

[80] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. 7920 :707–731, 2013.

[81] C. Märtin. Software life cycle automation for interactive applications : The ame design environment. In *Computer-Aided Design of User Interfaces*, pages 57–74, 1996.

[82] T. Miller and R. Zeleznik. An insidious haptic invasion : adding force feedback to the x desktop. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'98)*, pages 59–64. ACM Press, 1998.

[83] W. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004.

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

[85] B. A. Myers. Encapsulating interactive behaviors. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'89)*, pages 319–324. ACM Press, 1989.

[86] B. A. Myers. A new model for handling input. *ACM Transaction on Information Systems*, 8(3) :289–320, 1990.

[87] R. Norman and S. Draper. *User Sentered System Design : new Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, 1986.

[88] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2010.

[89] D. R. Olsen. A programming language basis for user interface. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'89)*, pages 171–176. ACM, 1989.

[90] OMG. Idl to java language mapping specification. Technical specification, OMG, june 1999.

[91] OMG. MDA specification, 2001.

[92] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *Proceedings of International World Wide Web Conference (WWW'05)*, pages 671–681. ACM Press, 2005.

[93] F. Paternò, C. Santoro, J. Mäntyjärvi, G. Mori, and S. Sansone. Authoring pervasive multimodal user interfaces. *International Journal on Web Engineering and Technology*, 4(2) :235–261, 2008.

[94] A. Puerta. The MECANO project : Comprehensive and integrated support for model-based interface development. In *Computer-Aided Design of User Interfaces*, 1996.

[95] A. R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4) :40–47, 1997.

[96] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *Proceedings of the DocEng'04*, pages 115–123. ACM Press, 2004.

[97] C. Ramsdale. Building mvp apps. Web article, Google, 2010.

[98] D. Rubine. Combining gestures and direct manipulation. In *Proceedings of the ACM Conference on Human factors in Computing Systems (CHI'92)*, pages 659–660. ACM Press, 1992.

[99] C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly, 2005.

[100] M. Serrano, L. Nigay, J.-Y. L. Lawson, A. Ramsay, R. Murray-Smith, and S. Denef. The openinterface framework : a tool for multimodal interaction. In *CHI '08 extended abstracts on Human factors in computing systems*, pages 3501–3506, 2008.

[101] G. Seshadri. Understanding javaserver pages model 2 architecture. *JavaWorld*, 1999.

[102] B. Shneiderman. *Designing the User Interface, Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3rd edition, 1998.

[103] P. P. Da Silva. User interface declarative models and development environments : A survey. In *Design, Specification and Verification of Interactive Systems (DSV-IS'00)*, pages 207–226. Springer-Verlag, 2000.

[104] P. P. Da Silva and N. W. Paton. UMLi : The unified modeling language for interactive applications. In *UML 2000 - The Unified Modeling Language*, pages 117–132. Springer, 2000.

[105] D.C. Smith, C. Irby, R. Kimball, and E. Harslem. The star user interface :an overview. In *Proceedings of the National Computer Conference*, pages 515–528, 1982.

[106] J. Smith. Wpf apps with the model-view-viewmodel design pattern. *MSDN Magazine*, february 2009.

[107] J. Sottet, G. Calvary, J. Coutaz, and J.-M. Favre. A model-driven engineering approach for the usability of user interfaces. In *Proceedings of Engineering Interactive Systems 2007*, pages 140–157, 2007.

[108] C. Soutou. *UML 2 pour les bases de données*. Eyrolles, 2007.

[109] Sun. Javabeans api specification. Specification document, 1997.

[110] P. Szekely. Retrospective and challenges for model-based interface development. In *Design, Specification and Verification of Interactive Systems (DSV-IS'96)*, pages 1–27. Springer-Verlag, 1996.

[111] P. Szekely, P. Luo, and R. Neches. Facilitating the exploration of interface design alternatives : the humanoid model of interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'92)*, pages 507–515. ACM, 1992.

[112] J. Tapper, J. Talbot, M. Boles, B. Elmore, and M. Labriola. *Adobe Flex 2 : Training from the Source*. Adobe Press, 2006.

[113] J.C. Tarby. *Gestion Automatique du Dialogue Homme-Machine à partir de Spécifications Conceptuelles*. PhD thesis, Université Toulouse I, 1993.

[114] J. Vanderdonckt. A MDA-compliant environment for developing user interfaces of information systems. *Lecture Notes in Computer Science*, 3520/2005 :16–31, 2005.

[115] J. Vanderdonckt. Model-driven engineering of user interfaces : Promises, successes, and failures. In *Proceedings of 5th Annual Romanian Conference on Human-Computer Interaction (ROCHI'08)*, 2008.

[116] J. Vanderdonckt and F. Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems (CHI'93)*, pages 424–429. ACM, 1993.

[117] L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *Proceedings of the 11th international conference on World Wide Web (WWW'02)*, pages 474–485. ACM, 2002.

[118] W3C. Document Object Model level 3 events specification. Normative recommendation, W3C, 2003.

[119] J. B. Warmer and A. G. Kleppe. *The object constraint language : getting your models ready for MDA*. Addison-Wesley.

[120] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, and P. Markopoulos. Beyond hacking : a model based approach to user interface design. In *Proceedings of the British Conference on Human-Computer Interaction HCI'93*, pages 217–231. University Press, 1993.

[121] B. V. Zanden and B. A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'90)*, pages 27–34. ACM, 1990.

Glossaire

AcT.NET	ACtive Transformation on the .NET platform
API	Application Program Interface
ATL	Atlanmod Transformation Language
BO	Boîtes à Outils
DiverSE	DIVERsity-centric Software Engineering
DOM	Document Object Model
DPI	Document Presentation Instrument
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
EMF	Eclipse Modeling Framework
EMN	Ecole des Mines de Nantes
eXAcT	EXtended-markup-language ACtive Transformation)
GMF	Graphical Modeling Framework
GPL	General Purpose Language
GRI	Groupe de Recherche en Informatique
GWT	Google Window Toolkit
HID	Humain Interface Device
HTML	Hyper Text Markup Language
ICLN	Interaction, Connaissances et Langage Naturel
ICON	Input device CONfiguration
IDL	Interface Definition Language
IDM	Ingénierie Dirigée par les Modèles
IHM	Interface Homme-Machine
IIHM	Ingénierie des IHM
IRISA	Institut de Recherche en Informatique et Systèmes Aléatoires
JCJC	Jeunes Chercheurs, Jeunes Chercheuses
LERIA	Laboratoire d'Etude et de Recherche en Informatique d'Angers
LIG	Laboratoire d'Informatique de Grenoble
LIUM	Laboratoire d'Informatique de l'Université du Maine

Loa	Langage pour Opérations Actives
Malan	MApping LANguage
MB-UIDE	Model-Based User Interface Development Environment
MDA	Model Driven Approach
MINT	Méthodes et outils pour l'INTeraction à gestes
MOA	Métaheuristiques, Optimisation et Applications
MODESTE	MODEles, Services et TEsts
MOF	Meta Object Facility
MPIA	Maturation de Projets Innovants en Anjou
MVC	Model-View-Controller
MVP	Model-View-Presentation
MVVM	Model-View-ViewModel
NEXT-CITYGEN	exploring and exploiting smart CITY diversity to build the NEXT GENeration of intelligent user interfaces
OLE	Object Linking and Embedding
OO	Object Oriented (ou Orienté Objet)
OOE	Open Object Embedding
OOUI	Application Program Interface
OXII	Outils pour la Conception de Systèmes d'Informations Interactifs
PAC	Présentation, abstraction, contrôle
PC	Personal Computer
PDA	Personal Digital Assistant
PFE	Projet de Fin d'Etudes
POO	Programmation Orientée Objet
RIA	Rich Internet Application
SDK	Software Development Kit
SE	Systèmes d'Exploitation
SI	Systèmes Interactifs
SVG	Scalable Vector Graphics
TICE	Technologies de l'Information et de la Communication pour l'Education
TRAME	TRAnsformation de Modèle pour l'Embarqué
UML	Unified Modeling Language
VIFORM	superVIsion de situations de FORmation Médiatisée
WIMP	Window, Icon, Menu, Pointer
WPF	Windows Presentation Fundation
X3D	Extensible 3D
XAML	eXtensible Application Markup Language
XSLT	eXtensible Stylesheet Language Transformation

Annexes

Table des matières

A. Résumé des travaux I

B. Curriculum Vitae III

C. Liste des publications V

D. Publications sélectionnées VII

A. Résumé des travaux

Le développement des systèmes interactifs (SI) met en jeu un nombre significatif de composantes logicielles et matérielles sélectionnées parmi un panel très large et souvent versatile. Ainsi, chaque développement s'appuie au minimum sur un langage de programmation, une boîte à outils dédiée au développement d'IHM, une plate-forme d'exécution logicielle et une plate-forme matérielle. Par exemple, une application interactive pour plate-forme mobile d'exécution Android est construite sur la base du langage Java et de la boîte à outils Android, et doit pouvoir s'exécuter sur des plates-formes très variées telles que les tablettes et les « smartphones ». Une application interactive pour le Web est par exemple développée en utilisant les langages JavaScript et HTML pour la plate-forme d'exécution cliente, ainsi qu'en Java pour la plate-forme d'exécution côté serveur.

Cette grande variabilité induit une difficulté quant aux choix des multiples composantes de développement à arrêter. Une fois ces choix arrêtés, le développement est contraint par les capacités offertes par les composantes retenues. De plus, ces choix ne sont pas nécessairement pérennes. Le premier point ne facilite guère le développement des SI allant au-delà des systèmes habituels. Le second point implique une refonte intégrale du système dès lors qu'une des composantes utilisée devient obsolète.

Une solution à la difficulté inhérente du développement des SI est de focaliser l'effort sur l'élaboration des différents *modèles* du SI plutôt que sur leur implémentation. La modélisation devient alors la partie non contrainte aux choix technologiques des composantes retenues et reste pérenne car indépendante de ces composantes. Cependant, si de nombreux méta-modèles ont été proposés dans la littérature dans ce sens, force est de constater qu'ils ne sont pas mis en œuvre par les boîtes à outils et les langages de programmation associés. Ainsi, les modèles restent trop souvent des modèles documentaires formalisant l'expression du besoin et servant de point d'entrée à la programmation. Dans cette approche usuelle, la cohabitation des modèles et des programmes qui les implémentent n'est pas nécessairement simple : les modèles se cantonnent à la couche « spécification » du SI, tandis que les programmes restent dans la couche « implémentation ».

Les travaux présentés dans cette HDR visent à montrer comment il est possible de réduire le travail d'implémentation en se focalisant sur l'élaboration de modèles exécutables. Ceci est rendu possible en définissant un méta-modèle des SI sur la base des modèles conceptuels proposés par la littérature (tels que MVC, PAC et l'interaction instrumentale), et en outillant ce méta-modèle d'un langage de programmation permettant de décrire les modèles des différentes composantes d'un SI ainsi que leur relation. Les différents modèles, de part leur capacité à s'exécuter directement sur les plates-formes matérielles et logicielles cibles, ne sont plus de simples spécifications de programme. De plus, ils préservent une certaine indépendance relativement aux plates-formes ciblées, d'où une pérennité accrue. Les modèles présentés dans ce mémoire sont ainsi à considérer comme situés à un niveau au-dessus des boîtes à outils et langages de programmation associés. Sont donc exclus les modèles de plus haut niveau que sont, par exemple, les modèles de tâches des utilisateurs.

La langage Loa constitue la contribution principale de cette HDR. Une application écrite en Loa fournit un ensemble de modèles liés entre-eux par des correspondances. Ces modèles définissent les objets du domaine, leurs présentations graphiques, les interactions et les actions par le biais de collections. Les correspondances sont fondées sur le concept d'opération active, les opérations actives étant des opérations usuelles sur les collections, telles que la sélection, la transformation ou le tri, pour lesquelles des algorithmes incrémentaux sont fournis. La théorie des opérations actives, nouvellement formalisée dans le mémoire d'HDR sur la base de la théorie initiale publiée, définit un système de types élémentaires et de types de collection sur lequel un jeu d'une cinquantaine d'opérations actives a été défini. Il a été montré que l'usage de ces opérations actives dans le contexte du développement des SI permet de repousser les limites des langages ou/et mécanismes de liaison de données proposés par les boîtes à outils contemporaines dédiées aux RIA (Rich Internet Applications). L'outillage (éditeur et compilateur) du langage Loa est cours de finalisation. Il permettra aux modèles écrits en Loa de s'exécuter sur différentes plates-formes clientes telles la plate-forme JavaScript-HTML5 du Web et la plate-forme Java+Swing, pour lesquelles l'implémentation des opérations actives a déjà été réalisée.

B. Curriculum Vitae

Formation

2004 Doctorat en Informatique au LRI (Laboratoire de Recherche en Informatique), Université Paris XI, Orsay - *Espaces de travail interactifs et collaboratifs : Vers un modèle centré sur les documents et les instruments d'interaction.*
Jury : M. Beaudouin-Lafon (directeur de thèse, LRI et INRIA Futurs), A. Derycke (rapporteur, Trigone, Université de Lille), M. Riveill (rapporteur, ESSI, Université Sophia Antipolis), P. Palanque (examinateur, IRIT, Université de Toulouse), V. Quint (examinateur, INRIA Grenoble)

2000 DEA en Informatique au LRI (Laboratoire de Recherche en Informatique), Université Paris XI, Orsay

1991 Agrégation de Physique Appliquée, Université de Poitiers

1989 Diplôme d'ingénieur ENSIEG (École Nationale Supérieure d'Ingénieurs Électriciens de Grenoble), INPG, Université de Grenoble

1985 Classes préparatoires au lycée Clémenceau (Nantes)

1984 Baccalauréat - série E (La Rochelle)

Expérience professionnelle

1996-2014 ESEO, Angers - Enseignant / Chercheur, *Département Informatique* :
- Enseignements en Informatique et Génie Logiciel
- Activités liées à la pédagogie : suivi des étudiants, encadrement, projets, etc.
- Recherche académique autour du thème de l'*Ingénierie des Interfaces Homme-Machine* dans une approche centrée sur les modèles exécutables

1991 ESEO, Angers - Enseignant / Chercheur, *Laboratoire de Recherche Appliquée* :
- Participation aux projets de recherche et développement industriels
- Enseignements en Science Physique et en EEA
- Activités liées à la pédagogie : suivi des étudiants, encadrement, projets, etc.

1990 CELDUC Automation, Saint-Etienne - emploi en intérim Cadre (9 mois). *Mise en place d'une documentation technique des produits CELDUC-Automation.*

1989 EdF, GRPH de Grenoble (Groupement Régional de Production Hydraulique) - stage Ingénieur (5 mois), *Étude des algorithmes de régulation en tension et en fréquence du réseau EdF de l'île de la Réunion (macro réseau indépendant).*

Recherche (2001-2014)

Mots-clés

- **Interfaces Homme-Machine (IHM) et Ingénierie des modèles (IDM)**
- Documents interactifs, Collecticiels, « Rich Internet Application » (RIA)

Coopérations principales

- IRISA (INRIA - Rennes) - équipe DiverSE (ex. Triskell) : statut officiel de Collaborateur extérieur INRIA+IRISA depuis 2013 (coopération depuis 2008)
- LERIA (Université d'Angers) - équipes ICLN et MOA : co-encadrement de thèses
- ENSIM (Université du Maine) : montage projet ANR 2014, 2nde phase en cours

Publications

- Revues à comité de lecture : 2 internationales, 2 nationales
- Congrès avec actes et comité de lecture : 20 internationaux (dont 12 dans conférences de rang A [1]), 4 nationaux
- Congrès nationaux avec comité de lecture : 8

Réalisations

- *DoPIdom* (2008) : boîte à outils faisant la synthèse de différentes réalisations publiées (*OpenDPI*, *EventPoints* et *eXAcT*)
- *Loa* (2014) : langage (DSL) outillé faisant la synthèse des réalisations faites et publiées depuis 2001 - financement MPIA 2011 (Maturation Projet Innovant en Anjou).

Enseignement

1999-2013 : Cycle ingénieur ESEO

- 1ère année :
  - Langage à objet et langage impératif
  - Algorithmique et structures de données
- 2ème année : Documents XML
- 3ème année, option « Systèmes d'Information » :
  - Interfaces Homme / Machine, Technologies XML
  - Composants logiciels, Plate-forme .NET

1991-1998 : ESEO apprentissage (ex. ISGTA)

- 1ère année : Physique générale
- 2ème année : Physique des composants, Lignes de transmission
- 3ème année : Réseaux de terrain, Architecture distribuée et concept CIM

1. En se basant sur CORE 2008 et en considérant EICS de rang A

C. Liste des publications

Revue internationale à comité de lecture

[1] A. Blouin, B. Combemale B. Baudry et O. Beaudoux. « Kompren : Modeling and Generating Model Slicers ». In : *Software and Systems Modeling (Springer)* (nov. 2012), p. 1–17.

[2] O. Beaudoux et M. Beaudouin-Lafon. « OpenDPI : A Toolkit for Developing Document-centered Environments ». In : *Enterprise Information Systems VII (Springer)* (2006), p. 231–239.

Revues nationales à comité de lecture

[3] A. Blouin, O. Beaudoux et S. Loiseau. « Un tour d'horizon des approches pour la manipulation des données du Web ». In : *Document Numérique (Lavoisier)* 11.1-2/2008 (2008), p. 63–83.

[4] O. Beaudoux. « Un modèle de composants (inter)actifs centré sur les documents ». In : *Revue Information, Interaction, Intelligence (Cépaduès)* 4.1 (2004), p. 41–58.

Congrès internationaux avec actes et comité de lecture

[5] M. Smati et al. « Toward the design of a generic model of interoperability for SIEC ». In : *Proceedings the 15th International Conference on Enterprise Information Systems (ICEIS'13)*. ScitePress, 2013, p. 329–333.

[6] O. Beaudoux, M. Clavreul et A. Blouin. « Binding Orthogonal Views for User Interface Design ». In : *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO'13)*. ACM press, 2013, 4 :1–4 :5.

[7] O. Beaudoux et al. « Specifying and Running Rich Graphical Components with Loa ». In : *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems (EICS'12)*. ACM press, 2012, p. 169–178.

[8] O. Beaudoux et al. « Specifying and implementing UI Data Bindings with Active Operations ». In : *Proceedings of the 3rd ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'11)*. ACM press, 2011, p. 127–136.

[9] A. Blouin et al. « Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation ». In : *Proceedings of the 3rd ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'11)*. ACM press, 2011, p. 85–94.

[10] A. Blouin, B. Combemale B. Baudry et O. Beaudoux. « Modeling Model Slicers ». In : *Proceedings of the 14th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*. Springer, 2011, p. 62–76.

[11] O. Beaudoux et A. Blouin. « Linking Data and Presentations : from Mapping to Active Transformations ». In : *Proceedings of the 2010 ACM symposium on Document engineering (DocEng'10)*. ACM Press, 2010, p. 107–110.

[12] O. Beaudoux et al. « Active Operations on Collections ». In : *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*. Springer, 2010, p. 91–105.

[13] O. Beaudoux, A. Blouin et J.-M. Jézéquel. « Using Model Driven Engineering Technologies for Building Authoring Applications ». In : *Proceedings of the 2010 ACM symposium on Document engineering (DocEng'10)*. ACM Press, 2010, p. 279–282.

[14] A. Blouin et O. Beaudoux. « Improving modularity and usability of interactive systems with Malai ». In : *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)*. ACM Press, 2010, p. 115–124.

[15] A. Blouin, O. Beaudoux et S. Loiseau. « Malan : A Mapping Language for the Data Manipulation ». In : *Proceedings of the 2008 ACM symposium on Document engineering (DocEng'08)*. ACM Press, 2008, p. 66–75.

[16] A. Blouin et O. Beaudoux. « Mapping paradigm for document transformation ». In : *Proceedings of the 2007 ACM Symposium on Document Engineering (DocEng'07)*. ACM Press, 2007, p. 219–221.

[17] O. Beaudoux. « XML Active Transformation (eXAcT) : Transforming Documents within Interactive Systems ». In : *Proceedings of the 2005 ACM Symposium on Document Engineering (DocEng'05)*. ACM Press, 2005, p. 146–148.

[18] O. Beaudoux. « Event Points : Annotating XML Documents for Remote Sharing ». In : *Proceedings of the 2005 ACM Symposium on Document Engineering (DocEng'05)*. ACM Press, 2005, p. 159–161.

[19] O. Beaudoux et M. Beaudouin-Lafon. « OpenDPI : A Toolkit for Developing Document-centered Environments ». In : *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'05)*. T. 5. ICEIS Press, 2005, p. 39–47.

[20] O. Beaudoux et M. Beaudouin-Lafon. « DPI : A Conceptual Model Based on Documents and Interaction Instruments ». In : *People and Computer XV - Interaction without frontier (Joint proceedings of HCI'01 and IHM'01)*. Lille, France : Springer-Verlag, 2001, p. 247–263.

Congrès nationaux avec actes et comité de lecture

[21] O. Beaudoux, A. Blouin et S. Hammoudi. « Correspondances et transformations actives dédiées aux IHM (fr) ». In : *Actes des èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM 2009)*. 2009, p. 115–130.

[22] A. Blouin et O. Beaudoux. « Malai : un modèle conceptuel d'interaction pour les systèmes interactifs (fr) ». In : *Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine (IHM'09)*. ACM Press, 2009, p. 129–138.

[23] O. Beaudoux. « DoPIdom : Une approche de l'interaction et de la collaboration centrée sur les documents (fr) ». In : *Actes de la 18ème conférence francophone sur l'Interaction Homme-Machine (IHM'06)*. ACM Press, 2006, p. 19–26.

[24] O. Beaudoux. « DoPIdom : Une boîte à outils pour la conception d'interfaces centrées sur les documents XML ». In : *Actes de la conférence francophone sur l'Interaction Homme-Machine (IHM'04)*. ACM Press, 2004, p. 187–190.

[25] O. BEAUDOUX. « Un Pivot pour la Collaboration : les Places Publiques ». In : *Annexes des actes de la conférence francophone sur l'Interaction Homme-Machine (IHM'02)*. 2002.

Congrès nationaux avec comité de lecture

[26] O. BEAUDOUX, A. BLOUIN et M. CLAVREUL. *Serveur d'IHM pour plate-forme IDM : architecture générique et implémentation*. IDM 2010 (atelier IDM-IHM). 2010.

[27] A. BLOUIN, G. NAIN et O. BEAUDOUX. *Malan et Malai pour la conception de systèmes interactifs : perspectives d'intégration dans Kermeta*. IDM 2010 (atelier IDM-IHM). 2010.

[28] O. BEAUDOUX. *Active transformation with Kermeta*. Kermeta Day. 2009.

Thèses et masters

[29] A. BLOUIN. « Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données ». Thèse de doct. Université d'Angers, 2009.

[30] O. BEAUDOUX. « Espaces de travail interactifs et collaboratifs : vers un modèle centré sur les documents et les instruments d'interaction ». Thèse de Doctorat. LRI, 2004.

[31] O. BEAUDOUX. *Paradigmes et éléments architecturaux d'une boîte à outils post-WIMP*. Rapport technique. ESEO-LRI, 2000.

Rapports

[32] O. BEAUDOUX. *eXAcT : A Novel Approach to Transformations for Rich Internet Applications*. Rapp. tech. ESEO, 2007, p. 31.

[33] O. BEAUDOUX et M. BEAUDOUIN-LAFON. *Envisionning an Interactive and Collaborative Workspace based on Documents and Interaction Instruments*. Rapport de recherche + vidéo. LRI-ESEO, 2002.

[34] O. BEAUDOUX. *Tour d'horizon des systèmes centrés sur les documents*. Présentation dans le cadre du groupe de travail ALF (GdR I3). LRI-ESEO, 2001.

D. Publications sélectionnées

Les pages suivantes fournissent les quatre publications les plus représentatives des travaux présentés dans cette HDR : [20], [12], [8] et [7].

DPI, A Conptual Model Centered on Documents and Interaction Instruments

Olivier Beaudoux[1,2] and Michel Beaudouin-Lafon[1]

[1]*Laboratoire de Recherche en Informatique,*
Université Paris-Sud,
Bâtiment 490 - 91405 Orsay, France

[2]*Département Génie Informatique, Réseaux et Télécoms,*
Ecole Supérieure d'Electronique de l'Ouest,
4 rue Merlet de la Boulaye, 49009 Angers, France

Tel: *+33 2 41 86 67 67 et +33 1 69 15 69 10*
Fax: *+33 2 41 87 99 27 et +33 1 69 15 65 86*
EMail: *olivier.beaudoux@eseo.fr et mbl@lri.fr*
URL: *http://www.lri.fr/~ beaudoux et http://www.lri.fr/~ mbl*

Interactive workspaces are all based on the application concept while documents occupy a main place in modern information systems. Using such workspaces force to juggle between several applications in order to compose various content types into documents. This results in an increasingly complexity of computer systems.

The DPI model (Documents, Presentations, Instruments) provides an alternative way by proposing a conceptual model separating edited contents (documents) from means of edition (interaction instruments). Such a model makes it possible to edit a document through simultaneous multiple presentations. A same instrument can be used to edit contents of different kind, thus making interaction more fluid and reducing user cognitive load. This conceptual model is supplemented by a functional model which details implementation principles.

Keywords: Conceptual model, instrumental interaction, compound document, interactive workspace, metaphor, action and perception

1 Introduction

In their large majority, current interactive environments are based on the concept of application. An application is dedicated to the handling of typed data like text, image, vectorial drawing. The windows systems make it possible to toggle from one application to another, and techniques such as copy-paste allow to transfer contents between applications. Many activities, such as the building of a Web page, a technical document, or an illustration, force the user to juggle between several applications. Such an approach generates an additional cognitive load for the user and makes the current systems complex of use.

Software publishers started to react to this fact by integrating functionalities intended to reduce this useless complexity. In the Microsoft Office software, three applications (Word, PowerPoint and Excel) integrate functions of vectorial drawing. These drawing "mini-applications" have similar interfaces, but nevertheless are different: they do not share interactions neither document formats. Drawing functions are limited so they do not discard the need of a specialized application as soon as the drawings become a little complex. This approach results in duplicating functionalities without really solving the problem.

Another approach consists in offering an open architecture making it possible for third parties to develop and market application extensions called *plug-ins*. Many plug-ins are available for Adobe Photoshop (image editing) (Gray, 1997), QuarkXPress (page layout), Macromedia Director (multimedia development), and so on. For example, Photoshop plug-ins make it possible to carry out vectorial drawings or simple 3D models. Moreover, many applications are compatible with Photoshop plug-ins. From the user point of view, plug-ins make it possible to extend an application according to its needs, without introducing new application into its workflow, and to find same functionalities from one application to another (they accept the same plug-ins). Nevertheless, from the inteface point of view, plug-ins behave like "mini-applications", often integrated into their parent application by the way of one (bulky) modal dialogue box.

A last approach consists in creating applications whose interfaces are increasingly compatible in order to support an homogeneous interaction and an easy transition from one application to another. The best examples are the Adobe software products, which adopt a common presentation of interface, pallets and layered editing, shared between Photoshop (image edition), Illustrator (drawing), GoLive (Web sites design) and InDesign (page layout).

These approaches tend to put *documents*, rather than applications, in the center of interaction process. Historically, the Xerox Star system (Johnson et al., 1989) has adopted this document centered approach. More recently, frameworks such as OpenDoc (Apple, 1994) and OLE (Brockschmidt, 1995) tend to factorize application use through *compound documents*: a document is not managed by an application dedicated to its type, but its various parts can be handled by any part compatible applications. These approaches do not discard applications but make them less visible. For example, when a document part is clicked on, menubar and pallets are reconfigured to allow its content edition, as when we switch from one application to another. The main advantage is the *in-palce* editing ability, without copying-

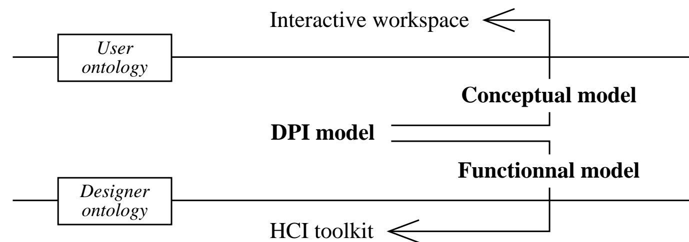

Figure 1: Position of the overall DPI model

pasting between applications. But transition from one document part to another creates a break in the interaction (each part manages its own interface), without any mechanism making it possible to share tools between interfaces of the various parts.

Thus, if one observes the current evolution towards centered document systems, one notes that this evolution is done to the detriment of the interaction. We propose to repair this imbalance by defining a conceptual model of interface separating document contents from interaction on these contents. This model is called DPI: Documents, Presentations, Instruments.

The DPI model combines a document model based on XML (W3C, 2000a) and an interaction model based on the instrumental interaction (Beaudouin-Lafon, 1997). This article presents two sides of the DPI model: the conceptual model that matches a *user ontology*, and the functional model that matches a *designer ontology* (Figure 1). To bring the DPI model to full maturity, the conceptual model will have to be concretized by an interface model and the functional model will have to be fully specified by a software architecture model. The final goal is a toolbox definition for new interfaces generation.

The following two sections explains the conceptual model and the functional model. Then, we compare the DPI model with some related works and we conclude by presenting futur works around this innovative model.

2 Conceptual Model

In user ontology, document and instrument concepts appear natural: the document is a data medium and the instrument is a means of creating and modifying document. Thus, the DPI model is based on the document and instrument *metaphors*.

2.1 Instrument Metaphor

When we daily act on our environment objects, we generally use tools or instruments and rather make direct actions on these objects. For example, we use a pen to write on a paper sheet. This observation forms the basis of the instrumental interaction model (Beaudouin-Lafon, 1997; Beaudouin-Lafon, 2000).

Interaction with instruments can be more or less direct. For example, we

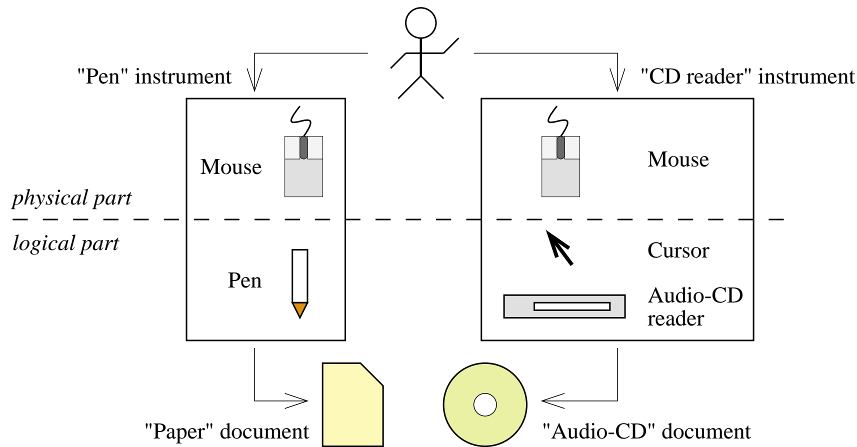

Figure 2: Instrument description

directly act onto the paper sheet by handling a pen, whereas we indirectly operate on a CD using a CD reader. This indirection appears in the same way when the pen or the CD reader is simulated by an interactive system and is handled using a mouse (Figure 2).

These examples show that an instrument has two facets, the physical facet (the instrument exists outside the system) and the logical facet (the instrument exists inside the system while being perceptible outside):

> “The physical part includes input-output transducers used by the instrument. Inputs receive the physical action of user and outputs give a feedback information to him (...) The logical part of the instrument includes the method to transform user actions onto the logical instrument (input), and the representation of the instrument itself (output)” (Beaudouin-Lafon, 1997)

However, we do not divide an instrument in two sub-instruments, one physical and one logical. Users must feel continuity between logical and physical sides.

Lastly, if most instruments can modify existing objects, other instruments just observe them: they are used to perceive rather than to act. Magnifying glass or magic lens of Perlin & Fox (1993) are such instruments. They behave as direct instruments whose action carries on the instrument itself and not on the object of interest. They provide *complementary presentations* of documents (see the following section).

This analysis of interaction instruments leads us to classify them in three categories (Figure 3):

1. *Direct instruments*, relating to the idea of tool, are handled with some hand / instrument continuity. Users act *directly* on document.

2. *Indirect instruments*, relating to the idea of device, are handled without direct

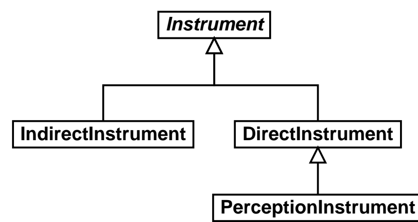

Figure 3: Instrument taxonomy

continuity with the object of interest. User act *indirectly* on document and *directly* onto instrument.

3. *Perception instruments* are handled as direct ones. Users act *directly* onto instrument (moving), but *not* on document.

2.2 Document Metaphor

A physical document (e.g. paper sheet, book, partition) is perceived by the user according to two merging sides:

1. The *persistence* side: the document (paper) is the persistence support (it absorbs pen ink). While writing on a document, the user perceives the persistent result of his action (feedback). By reading a document, the user first perceives its persistent side then interprets the contents.

2. The *presentation* side: the presentation is the intrinsic side of the document giving it a concrete appearance. A document has only one presentation, as a direct result of the persistent written informations.

If physical documents naturally couple persistence and presentation, electronic documents *uncouple* these two sides. Persistence of documents is carried by the file system, whereas presentations takes place on an output device, such as a screen. This decoupling appears in usual systems with the need for a regular document recording.

Several presentations of a same document can be achieve with such a decoupling, including different kinds of presentation (e.g. a text presentation in plan mode or in page mode, a visual or sound presentation of a musical partition). This capacity is interesting from an interaction point of view but spoil the initial document metaphor. In a user ontology, we have to erase as much as possible the decoupling between document and presentation.

2.3 Multiple Presentations Abstraction

In order to preserve the document metaphor while using the multiple presentations ability, the DPI model constantly links each presentation to corresponding document. Thus, from the user point of view, any document presentation *is* the document itself, as a persistent object. This implies a constant document updating.

If we limit any document to have only one presentation, the presentation idea becomes useless in a user ontology. However, the interest of multiple presentations

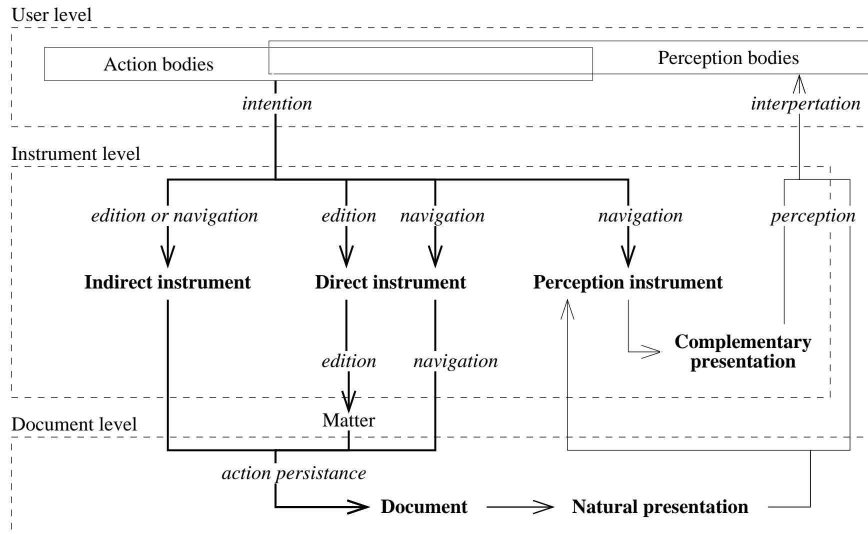

Figure 4: Conceptual model. Thick lines represent flows relating to the side action, and thin lines those relating to the side perception. For clearness, perception of instruments is not shown.

is well-known, as in the Zelig system (Celentano et al., 1992)). To benefit from the multiple presentations abilities, we need to clarify the idea of presentation.

We can introduce the multiple presentations idea using the cameras and monitors metaphor borrowed from the X_{TV} system (Beaudouin-Lafon et al., 1990): a document is filmed by one or more cameras and can be visualized by way of monitors. This metaphor makes it simple to understand the multiple presentations idea. It allows to edit a document using its *natural* presentation, and to visualize it by way of *complementary* presentations provided by monitors. This approach preserves the initial document metaphor. However, it goes against the Shneiderman (1983) principle of direct manipulation since complementary presentations remain passive.

Thus, we extend the metaphor making complementary presentations active. Any presentation allows document edition. This extension induces a fundamental need: the edition results must be *synchronous* from one presentation to another. Doing so, the abstract idea of multiple presentations becomes familiar to users, because they make the following report: since there is synchronism, presentations are about the *same* document, so presentations *are* the document.

Moreover, the idea of multiple and editable presentations opens the way with a multi-users model from which several participants simultaneously edit a same document from their workstations. This groupware extension is not developed front here.

2.4 Compatibility between Model Components

The conceptual model describes how the user handles documents using instruments through their presentations, and how it perceives the result. This model is shown using an action and perception flow diagram (Figure 4). Like the functional model explained in the following section, it uses on the *action-perception* idea of Norman & Draper (1986).

The model has three levels. The first level shows that users specify their intentions using action bodies, and interpret the result using perception bodies. According to the ecological approach of Gibson (1979), action and perception are strongly coupled: the user must perceive before acting and must also act to perceive, as in navigating through a document. Action and perception bodies can not be considered as independent.

The second level specifies how instruments transform user actions onto documents. Instruments take part in three essential functions: navigation, perception and edition. Navigation can be carried out by an indirect instrument (e.g. a search instrument), by a direct instrument (e.g. a scrolling tool), or by a perception instrument (e.g. a radar view). The edition can be carried out by an indirect instrument (e.g. a spell checker) or by a direct instrument transfering instrument "matter" towards the document (e.g. ink). Lastly, perception is carried out directly by a natural document presentation, or indirectly by instruments using complementary presentations (e.g. a magnifying glass).

The third level reveals the double role of documents: the action persistence role and the contents presentation role. We use the *matter* idea (e.g. the pen ink) as a means of document persistence.

The model separates instruments from documents. Thus, we can define instruments of edition, navigation or perception which can act onto various documents. This is not allowed in usual systems.

In order to specify relationship between the DPI model components, we introduce the concept of *compatibility*:

1. The *user* ↔ *instrument* compatibility indicates that an instrument if interaction is well suited to the instrument function in the action point of view as in the perception one. This is very much like the concept of *affordance* of Gibson (1977): instruments must express their functions in a directly perceptible way.

2. The *instrument* ↔ *document* compatibility indicates if an instrument can handle and/or perceive a document. For example, paper sheet is adapted to pen because it can receive pen ink. This compatibility gives the possible interactions between instruments and documents.

3. La *document* ↔ *user* compatibility indicates if the document perception is well suited to our various senses. For example, a plain paper sheet is compatible with visual perception, whereas a Braille code sheet is compatible with a tactile (and possibly visual) perception.

The compatibility is usual in our daily life, and we are not really aware of a compatibility but of its lack. The conceptual model objective is to give

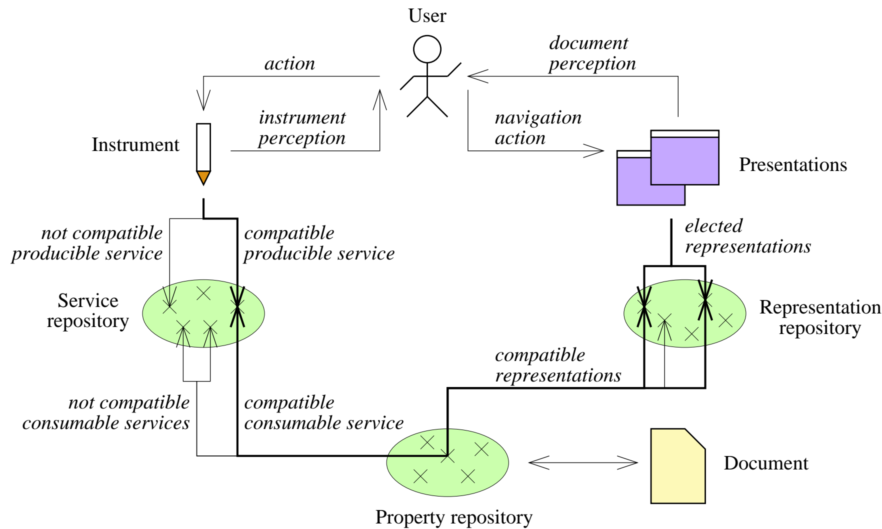

Figure 5: Functional model. The thick lines represent a sample instrument to presentation interaction path.

such an impression of possible combinations between documents, instruments and presentations. These combinations simplify interaction and reduce user cognitive load while offering a rich set of possibilities.

3 Functional Model

The functional model is the transcription of the conceptual model for the interface designer. Thus, it is expressed in a designer ontology. In order to build the functional model, we *reify* highlighted relations of the conceptual model using three "atoms" (Figure 5). These atoms are explained in sections 3.1 to 3.3 and are combined into documents and presentations (section 3.4), devices and instruments (section 3.5).

3.1 An Atom for Persistence: The Property

A document and its presentations are defined their *property* values. A property *definition* is a couple *(name, type)* and a property *value* is a couple *(name, value)*. A property can be *atomic* (it defines only one value), *composite* (it is made up of properties in a preset way), or *ensemblist* (it contains several properties in a nonpreset way).

Name-value associations are defined in documents and form their state. *Name-type* associations define the properties *scheme*. Document properties are visualized through presentations and edited using instruments. Some properties can not be edited, such as system properties (e.g. document creation date). This kind of information is stored in the scheme.

Using the property atom in our document model offers a low granularity, where systems such as OpenDoc or OLE give a strong granularity because they use to

the idea of *part*†. Moreover, the availability of the properties scheme makes the documents opened, at the opposite of owner formats. Thus, we can create interaction instruments independently of document types, so thay can act on known properties without modifying the remainder. For opening and extensibility, a well suited document format is (W3C, 2000a), where schemes are expressed as DTDs.

3.2 An Atom for Action: The Service

We use for the whole interaction objects the concept of *service*. The service represents the *basic medium* of user action transmission in the action-perception chain. A service of a *producer* can *activate* a service of a *consumer*. This chaining takes place from input devices to document properties by means of instruments. Service *activation* uses the idea of *compatibility* highlighted in the conceptual model (instrument → document compatibility).

The activation of a service S_{in}, consumable by a consumer C, by a service S_{out}, producible by a producer$P$, is defined as follows:

1. P and C are *likely to interact* if P is a direct instrument geometrically pointing on C, or if P is an indirect instrument targetting C.
2. If P and C is likely to interact, a *compatibility* test is carried out between each output service $(S_{out,i})_{i=1..n}$ of P, and each input service $(S_{in,j})_{j=1..p}$ of C. A *connection* $S_{out,i} \rightarrow S_{in,j}$ is made for each compatible pair $(S_{out,i}, S_{in,j})$.
3. A connection $S_{out,i} \rightarrow S_{in,j}$ is *activated* when $S_{out,i}$ service is invoked. The choice of invoked service must be carried out by producer. For example, an instrument which can move and recut an object provides two services. When the two services are connected, this instrument gives a means of choosing which one is activated, for example by offering two activation buttons.
4. Services remains connected as long as P and C are likely to interact.

The simplest kind of compatibility between service is the service names equality. However, defining generic compatibility rules is usefull. Thus, if we define a tree for services, two services S_1 and S_2 are compatible if and only if S_1 is an ancestor of S_2. For example, an instrument can provide the generic `move` service, and properties can provide specialized services such as `move_icon`, `move_note`, and so on. By defining `move_note` and `move_icon` as descendants of `move` in the service tree, we ensure their compatibility.

The service concept reify the usual command concept: a service is a command which an instrument can *activate* on a targetted document property. By using the compatibility rule, we can define generic instruments providing polymorphism: a same instrument will be able to edit different kind of properties. Thus, the service concept intrinsically applied the reification and polymorphism principles of Beaudouin-Lafon & Mackay (2000). In addition and from the interaction point of view, instrument services induce an fine grain interaction (of instrument level) while the OpenDoc editors (Apple, 1994) have a strong granularity (of editor interface level).

†A compound document is composed parts, which one contain specified data types.

3.3 An Atom for Perception: The Representation

Properties are perceptible using *representations*. The user perceives a document by way of representations *elected* among all property comptatible representations. Some standard (or canonical) representations are provided for simple types (e.g. integer, character string), composite types (e.g. date, time), and ensemblist types (e.g. tree, list, icons). Perception instruments are specially used to transform properties expressed in a document specific scheme into properties expressed in well-known schemes for which there are standard representations.

The representation election mechanism behaves like the service one: each representation provided by a document is selected if compatible with presentation medium. To create a presentation, a representation is elected either automatically, or asking user assistance. This election mechanism matches the document → user compatibility.

As for services, the use of representations allows a finer document presentation (of property level) whereas in OpenDoc or OLE, editors act with a larger granularity(of part level).

3.4 Documents and Presentations

The model does not make any assumption about document organization and availability. Hierarchical organization of traditional file systems is not necessarily well suited. *Collections* and dynamic queries of Dourish et al. (1999) are good candidates for the document organization because they are based on document properties. This side is not developed front here but was studied by Beaudoux (2000).

In addition, documents are not only users documents: they can be used for workspaces, collections, and for instruments themselves. This homogeneity increases the easiness of the interactive workspace management. All documents can also be queried by specialized languages such as XPath (W3C, 1999) and XML Query (W3C, 2000c).

A document is a tree of property values and a presentation is a tree of representation property value. For graphic interaction, these representations include geometrical properties (e.g. location, size). A direct instrument points to a document property by way of its presentation, so representations must maintain a inverse link towards the represented document property. Moreover, one particular presentation can contain properties that do not match any document properties. For example, a set representation using icons forces to store icon screen locations. Thus, presentations must store their own properties with their associated documents. This makes unusable any stylesheet idea, such as (W3C, 2000b): a more powerful mechanism is necessary to transform documents into presentations.

The use of multiple presentations allows edition of documents according to several viewpoints. In a scenario of a course support building, we need three components: students book, presentation transparencies, and teacher notes. A main presentation allows edition of the whole course (*semantic* level) through a framework tree facilitating navigation onto the document hierarchy. Three complementary presentations are adapted to the three course components (*geometry* level) with one specific formatting per presentation (e.g. figure location and size). Those four

presentations provides a powerful parallel edition of various course sides. Let us note that presentation state is restored with document reopening (e.g. windows size and location).

Representations are defined in a *repository* which dynamically evolves. When a document file is (down)loaded into the system, associated presentations can have elected representations not defined in this repository. In this case, we can (down)load all necessary representation definitions. This idea is similar to the plug-in loading of Web browsers.

3.5 Devices and Instruments

3.5.1 Physical, Logical and Simulated Devices

Input *devices* define a set of producible services. An input or output device is a tree whose nodes are sensors (see Beaudoux (2000) for more details). For example, potentiometers and buttons are the sensors of a mouse, while simple-click, double-click, and click'n drag are its producible services.

Physical input and output devices take part in the interactive process. We do not regard them as instruments (no *inheritance* relationship) but as the physical part of instruments (*association* relationship).

A *logical* device is built from one or more physical devices. A physical one button mouse can be seen as one "locator" and one "activator" *interactors*, within the meaning of Myers (1989). A logical three buttons mouse can be built from the physical mouse combined with two keyboard keys (as activators).

A *simulated* device emulated a physical device. A displayed keyboard handled by mouse is a simulated device. The Xerox Star system (Johnson et al., 1989) uses such a keyboard simultation for special input handling (e.g. mathematical formulas), and the X_{TV} toolbox (Beaudouin-Lafon et al., 1990) can be used to create its own simulated devices. Such devices can also be used for specilazed use as the simulated mouse of the *metamouse* system (Maulsby et al., 1989). Simlated devices must be conidererd as indirect instruments playing a device role.

The logical device is located in a designer ontology: an instrument builds an adapted logical device starting from the required functionalities. Thus, rather than define a logical device taxonomy, we release a *logical functionalities* taxonomy of physical devices.

3.5.2 Instrument Running

The instrument *physical part* gets user action and transfomrs them into commands send to the concerned document (Figure 6). Input functionalities required by an instrument are implicitly defined by its *consumable services*. The logical device is built as described in the preceding section. In case of multiple logical device possibilities, the instrument uses an overall document definig all logical devices, or request a user choice. Once the logical device built, *all* producible services of elected physical input devices are *connected* to all consumable services (1). The *logical part* defines the producible services. When a consumable services is *activated* by a physical device, an action is made onto the document (2), resulting in a producible service activation of targetted document property (3).

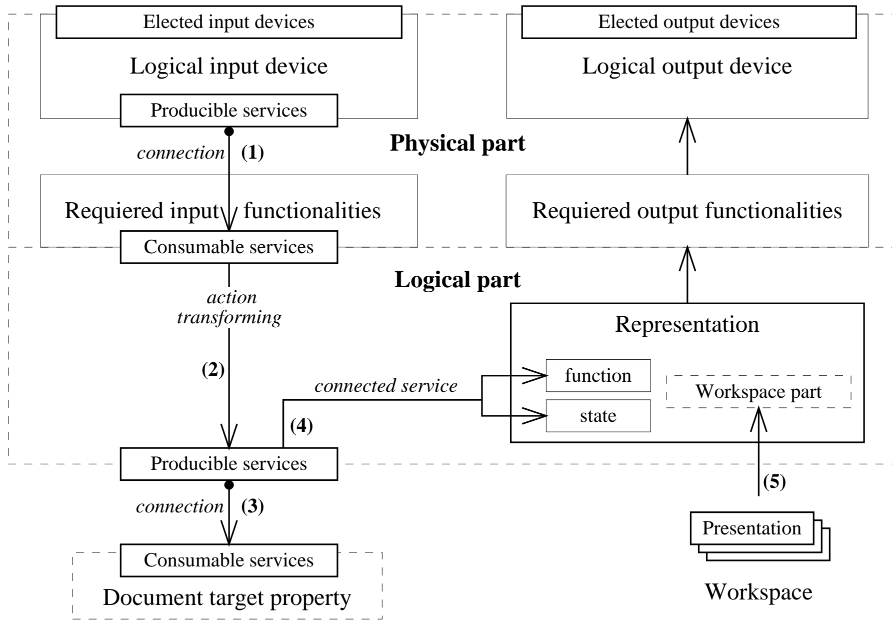

Figure 6: Instrument running

Instruments must be themself represented by way of output devices giving them a perception feature. This instrument feedback gives its function and state representations (4). Perception instruments also give a representation of surrounded documents (5).

Persistence of instrument properties is transprently ensured by an associated document. This document contains instrument state, configuration (logical device, behavior, representation), and on line help. The on line help is a significant point in a document centered system, as mentionned in (Sleeter, 1996). While help is attached to applications using a common format in usual systems, centered document systems must provide an overall help from different levels: workspace, documents and instruments levels.

The generics of services for property types induces the generics of instruments for documents (section 3.2). A same instrument can be here used in several contexts, unlike on existing systems. For example, a pen allows text typing into any writable document, annotation writing into any document, and label edition of a workspace element (e.g. a collection name). The service compatibility between instruments and document properties induces the ability to dynamically provide contextual paletts of instruments. Traditional contextual menus are preset applications and only depends on targetted objects while contextual paletts are dynamically calculated, according to available instruments and concerned properties.

Finally, different kind of instruments can be *chained* offering new interesting capcities. For example, combining a pen (direct instrument) with a magnifying

glass (perception instrument) allows a fine drawing, while the pen acts *through* the magnifying glass, as the Magic Lenses (Bier et al., 1993). No general system allows such an approach.

Instruments are declared in a dynamically evolving *repository* as for representations. Thus, we take a similar advantage as fro representation: when a document file is (down)load onto the workspace, handling instruments can also be (down)load for any edition, navigation or perception needs. This opens larger possibilites than the usual plug-ins use offers.

4 Related Works

4.1 Document Models

OpenDoc (Apple, 1994) is one of the rare document centered system design attempts. Dykstra-Erickson & Curbow (1997) insists that users do not have to handle applications in order to edit documents: they handle documents themself. Its document model is based on paper and *parts* metaphors. A document is a set of parts, and each part can be handled by a specialized *editor*. In our approach, we did not regard document as the most significant system entity: instrument is the interaction key component while document remains the interaction target entity. The main differences between OpenDoc and DPI model are:

- OpenDoc editors granularity is rather strong, lower than traditional applications, while DPI instrument granularity is low. In the same way, OpenDoc part granularity is stronger than DPI document property one.
- OpenDoc editors do not separate interaction from displaying whereas DPI instruments are uncoupled from presentations.
- Editors are specialized for a given part type while DPI instruments can be generic and DPI representations can be canonical.

Thus, the DPI model allows a better interaction continuity: in OpenDoc, the interactive environment (e.g. menubars, pallets) changes when moving from part to part, whereas DPI instruments can be continuously used on all workspace documents.

OLE framework (Brockschmidt, 1995) does not use any OpenDoc editors, but uses inter-application communication rules so that pultiple applications can act on a same document. Thus, OLE is not a really document oriented system, but it makes possible coumpond documents editing under Windows. In Microsoft Office, applications have homogeneous interfaces and interactions: interaction with OLE remains continuous. Therefore, this continuity breaks while using other applications.

OOE (Backlund, 1997) is a NextStep system extension which manages composite documents in a simplified way: in-place part editing is not available, and the displaying ability of a part in a whole document uses the *display PostScript* engine facility. OOE is application centered with a basic interaction mode: one click in a part invokes its creator application for opening and editing.

HotDoc (Buchner, 2000) is a Smalltalk extension of the MVC design pattern. It defines the `PartApp`, `PartView` and `PartController` classes, thus managing

the part idea under a MVC context. This extension remains simple and extends a well known pattern. Multiple views (or presentations) can handle a same model (or document) whereas OpenDoc, OLE and OOE cannot. However, the `PartApp` class usage illustrates that HotDoc interaction remains based on the OpenDoc editor idea, with the same strong granularities.

4.2 Interaction Models

Interactor (Myers, 1989) encapsulates interaction in a reduced number of object types (seven). An interactor is not based on an input device type but on its interaction *nature* (e.g. point set definition, one item selection). This approach shows the interaction / visualization separation needing, and the ability to use several input devices.

Our contribution holds the fact that the instrument model *unifies* the of physical, logical and simulated devices usage. Logical and simulated devices deal with the designer level: users see them through instruments in a transparently way. and are visible by the user only through the more general concept of instrument. Moreover, instruments can be from generic to strongly typed.

The instrumental interaction model (Beaudouin-Lafon, 2000) is the foundation of the DPI model. Therefore, it does not constitute a conceptual interface model. Combining interaction instruments and compound documents concepts made it possible to create generals and homogeneous conceptual and functional models. In particular, documents can be independent from instruments using DPI services.

4.3 Architecture Models

The model, view and controller components of the MVC design pattern (Krasner & Pope, 1988) could be considered as our document, presentation and instrument components. However, MVC is justified by its components synchronization and remains a purely compputering approach: MVC is an architecture model, not a conceptual interaction model. By opposition, the DPI model is user centered: our three atoms rise from a metaphorical approach and are based on a conceptual model. Lastly, we unify an interaction model based on instruments within a document model based on properties, in a context of direct manipulation.

5 Conclusion and Futur Works

The DPI conceptual model intends to break the complexity the current workspaces use and management. It is defined according to a user ontology *and* a designer ontology. User ontology underlines the document and instrument metaphors, and introduces the multiple presentations abstraction. The designer ontology is based on the property atom for the persistence side, the service atom for the action side and the representation atom for the perception side, and also on the compatibility concept managed by these three atoms.

We have illustrated the expressiveness of our DPI model with examples, thus comparing its capacities with usual system functionalities. However, the DPI model is abstract and must be validated through a concrete implementation. Thus, the next stage of this work is an interface model definition concretizing the conceptual side with realistic use cases. A pre-validation was however carried out with

the design and implementation of the CPN2000 application (Beaudouin-Lafon & Mackay, 2000; Beaudouin-Lafon & Lassen, 2000). The conceptual model of this application is a simplified version of the DPI model since it is about a "closed" application. Nevertheless, independence between documents and instruments, generic instruments, and instrument representation have been implemented and validated.

After its validation, the DPI model must be implemented in order to test it in a concrete way, building "real" documents and instruments. In addition, we will extend the model to *groupware* (shared edition of documents) and to some other environments like *mobile systems* and *increase reality* systems. In short, the DPI model constitutes a first essential stage in designing and implementing a new generation of interactive graphic workspaces using our new interaction paradigm.

References

Apple (1994), OpenDoc Technical Summary, Technical documentation, Apple Computer.

Backlund, B. E. (1997), "OOE: A Compound Document Framework", *SigCHI Bulletin* **29**(1), 68–75.

Beaudouin-Lafon, M. (1997), Interaction instrumentale: de la manipulation directe à la réalité augmentée, *in Actes des 9èmes Journées IHM, Poitiers*, Cépaduès Editions.

Beaudouin-Lafon, M. (2000), Instrumental Interaction: An Interaction Model for Designing Post-WIMP Interfaces, *in Proceeding of the ACM Human Factors in Computing Systems (CHI'2000)*, ACM Press, pp.446–453.

Beaudouin-Lafon, M. & Lassen, H. M. (2000), The Architecture and Implementation of CPN2000, A Post-WIMP Graphical Application, *in Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST'2000)*, ACM Press, pp.181–190.

Beaudouin-Lafon, M. & Mackay, W. (2000), Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces., *in Proceeding of the Advanced Visual Interfaces (AVI'2000)*, ACM Press.

Beaudouin-Lafon, M., Berteaud, Y. & Chatty, S. (1990), Creating Direct Manipulation Applications with XTV, *in Proceedings of the European X Window System Conference (EX'90)*.

Beaudoux, O. (2000), Paradigmes et éléments architecturaux d'une boîte à outils post-WIMP, Rapport technique, Computer science research labs of École Supérieure d'Électronique de l'Ouest and Université Paris-Sud (Orsay).

Bier, E. A., Stone, M. C., Pier, K., Buxton, W. & DeRose, T. D. (1993), Toolglass and Magic Lenses: The See-Through Interface, *in Proceedings of the 20th Annual Conference on Computer Graphics (SIGGRAPH'1993)*, ACM Press, pp.73–80.

Brockschmidt, K. (1995), *Inside OLE, Second Edition*, Microsoft Press.

Buchner, J. (2000), "HotDoc: A Framework for Compound Documents", *ACM Computing Surveys* **32**(1), 33–38.

Celentano, A., Pozzi, S. & Toppeta, D. (1992), A Multiple Presentation Document Management System, *in Proceedings of the 10th Annual International Conference on Systems Documentation (SIGDOC'92)*, ACM Press, pp.63–71.

Dourish, P., Edwards, W. K., LaMarca, A. & Salisbury, M. (1999), Using Properties for Uniform Interaction in the Presto Document System, *in Proceedings of the 12th Annual ACM Symposium on User interface Software and Technology (UIST'99)*, ACM Press, pp.55–64.

Dykstra-Erickson, E. & Curbow, D. (1997), The Role of User Studies in the Design of OpenDoc, *in Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS'97)*, ACM Press, pp.111–120.

Gibson, J. J. (1977), The Theory of Affordances, *in* R. Shaw & J. Bransford (eds.), *Perceiving, Acting, and Knowing*, Erlbaum Associates.

Gibson, J. J. (1979), *The Ecological Approach to Visual Perception*, Boston: Houghton Mifflin.

Gray, D. (1997), *The PhotoShop Plug-ins Book: Category Listings, Instructions and Examples*, Ventana Communications Group, Incorporated.

Johnson, J., Roberts, T. L., Verplank, W., Smith, D. C., Irby, C., Beard, M. & Mackey, K. (1989), "The Xerox "Star": A Retrospective", *IEEE Computer* **22**(9), 11–29.

Krasner, G. E. & Pope, S. T. (1988), "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Objet Oriented Programming* **1**(3), 26–49.

Maulsby, D. L., Witten, I. H. & Kittlitz, K. A. (1989), Metamouse: Specifying Graphical Procedures by Example, *in Conference Proceedings on Computer Graphics*, ACM Press, pp.127–136.

Myers, B. A. (1989), Encapsulating Interactive Behaviors, *in Conference Proceedings on Human Factors in Computing Systems (CHI'89)*, ACM Press, pp.319–324.

Norman, R. & Draper, S. (1986), *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates.

Perlin, K. & Fox, D. (1993), Pad: An Alternative Approach to the Ccomputer Interface, *in Proceedings of the 20th Annual Conference on Computer Graphics (SIGGRAPH'93)*, ACM Press, pp.57–64.

Shneiderman, B. (1983), "Direct Manipulation: A Step Beyond Programming Languages", *IEEE Computer* **16**(8), 57–69.

Sleeter, M. E. (1996), Building Online Help for a Component-Oriented Architecture, *in Proceedings of the 14th Annual International Conference on Marshaling new Technological Forces: Building a Corporate, Academic, and User-Oriented Triangle*, ACM Press, pp.87 – 94.

W3C (1999), XML Path Language (XPath) Version 1.0, W3C Recommendation, W3C.

W3C (2000a), Extensible Markup Language (XML) 1.0 (second edition), W3C Recommendation, W3C.

W3C (2000b), Extensible Stylesheet Language (XSL) Version 1.0, W3C Recommendation, W3C.

W3C (2000c), XML Query Data Model, W3C Working Draft, W3C.

Active Operations on Collections

Olivier Beaudoux[1], Olivier Barais[2], Jean-Marc Jézéquel[2], and Arnaud Blouin[3]

[1] ESEO Group, Angers - GRI Team olivier.beaudoux@eseo.fr
[2] University of Rennes 1 - Triskell Team {barais, jezequel}@irisa.fr
[3] INRIA Rennes - Triskell Team arnaud.blouin@inria.fr

Abstract. Collections are omnipresent within models: collections of references can represent relations between objects, and collections of values can represent object attributes. Consequently, manipulating models often consists of performing operations on collections. For example, transformations create target collections from given source collections. Similarly, constraint evaluations perform computation on collections. Recent research works focus on making such transformations or constraint evaluations active (*i.e.* incremental, or live). However, they propose their own solutions to the issue by the introduction of specific languages and/or systems. This paper proposes a mathematical formalism, centered on collections and independent of languages and systems, that describes how the implementation of standard operations on collections can be made active. The formalism also introduces a reversed active assignment dedicated to bidirectional operations. A case study illustrates how to use the formalism and its Active Kermeta implementation for creating an active transformation.

1 Introduction

The promise of model-driven engineering (MDE) is that the development and maintenance effort can be reduced by working at the model instead of the code level. Models define what is valuable in a system, and code generators produce the functionality that is common in the application domain. One of the main current issue for the MDE community is to support evolution in the stage of the software development process (*e.g.* to support incremental code generation). To address this issue, this paper works on formalizing operation on collections to support incremental models manipulation. Indeed, collections are omnipresent in the Model Driven Engineering field: collections of references can represent relations between objects, and collections of values can represent object attributes. Consequently, manipulating models often consists of performing operations on collections. Two essential model manipulations are constraint checking and model transformation. OCL collection iterators and transformation language collection shortcut illustrates the importance of collections in Model Driven Engineering. However, whenever a model is *modified*, these operations are not efficient since they require a full re-execution as if the model was newly created. Many different solutions have been proposed to solve such an issue; they

are all based on the concept of incremental [1], live [2] or active [3] model manipulation. Despite their omnipresence, collections are not the central piece of these approaches that are often tied to a specific language and/or system [4,5], or use usual manipulations combined with merge strategies [6].

This paper proposes a formalism that makes standard operations on collections active, independently from languages and systems. It shows how active transformations can be reduced to active operations on collections, thus underlining the interest of collections as first class objects for model manipulation. It evaluates the approach by studying complexities of active operations, and by explaining how active operations can be written with Active Kermeta, an implementation of our formalism on top of Kermeta [7].

The remainder of this paper is structured as follows. Section 2 explains how standard operations on collections can be made active through active loops, thus showing the foundation of our proposal. Section 3 illustrates the use of the formalism and its Active Kermeta implementation for writing active transformations. Section 4 evaluates complexities of active operations, discusses the possible optimizations, and compares the approach with related works. Finally, section 5 concludes on our contribution and its perspective.

2 From Standard to Active Operations

This section explains the semantics of active operations by describing their active loops: usual binary operations, application, selection, sort, and reversed assignment. Such a set of operations has been mainly inspired by OCL [8].

2.1 Preliminary: Definitions

The set of all collections is noted $\mathbb{C}$. Its subsets $\mathbb{U}$ and $\mathbb{O}$ define collections that respectively manage *uniqueness* and *order*. The four combinations define usual collection types: order set ($oset = \mathbb{U} \cap \mathbb{O}$), set ($set = \mathbb{U} - \mathbb{O}$), sequence ($seq = \mathbb{O} - \mathbb{U}$), and bag ($bag = \mathbb{C} - \mathbb{U} - \mathbb{O}$).

The following table introduces the minimal set of operations that is sufficient to express any other operation:

Operation	$C \notin \mathbb{O}$	$C \in \mathbb{O}$
$\lvert C \rvert$	cardinality of C	
$e \in C$	presence of e in C	
$e \in_i C$	n/a	presence of e in C at position $i \in [0..\lvert C \rvert[$
$C[i]$	n/a	element in C at position $i \in [0..\lvert C \rvert[$
$C[i..j]$	n/a	sub-collection of C from pos. i to pos. j
$C + e$	adds e into C	appends e at the end of C
$C +_i e$	n/a	inserts e into C at position $i \in [0..\lvert C \rvert]$
$C - e$	removes the first occurrence of e from C	
$C -_i$	n/a	removes from C the element at pos. $i \in [0..\lvert C \rvert[$

If $C \in \mathbb{U}$, operations $C + e$ and $C +_i e$ check the uniqueness of e within C: if $e \in C$ before the insertion, operations have no effect. The following table gives the notation used to iterate on collections:

Iteration	$C \notin \mathbb{O}$	$C \in \mathbb{O}$
$\forall e \in C,\, p(e)$	calls $p(e)$ for each element e of C	
$\forall e \in_i C,\, p(e,i)$	n/a	calls $p(e,i)$ for each element e at position i in C
$\forall^t e \in C,\, p_a(e)$	calls $p_a(e)$ for each element e added into C at time t	
$\forall^t e \notin C,\, p_r(e)$	calls $p_r(e)$ for each element e removed from C at time t	
$\forall^t e \in_i C,\, p_a(e,i)$	n/a	calls $p_a(e,i)$ for each e inserted at pos. i at time t
$\forall^t e \notin_i C,\, p_r(e,i)$	n/a	calls $p_r(e,i)$ for each e removed from pos. i at time t

The first two rows represent usual iterations throughout loops: the iteration is performed once for all elements of the collection. The next two rows define an iteration throughout an *active loop* that is composed of two *rules*: the *addition* rule *immediately* invokes procedure p_a for each element e of C (similar to usual loops), and *subsequently* invokes p_a each time a new element e is added into C; the *removal* rule *subsequently* invokes procedure p_r each time element e is removed from C. The last two rows define the *indexed active loop* dedicated to ordered collections; such a loop can also be used in some situations with unordered collections (*e.g.* selection and sort, see sections 2.4 and 2.5). Active loops thus observe additions and removals performed on collections; they also observe replacements since a replacement is considered as a removal+addition pair (see section 4.2). Usual and active loops can use a predicate; for example, $\forall e \in C \,/\, e \neq 1,\, p(e)$ calls p for each e different from 1.

Usual operation $B = op(A_1, ..., A_n)$ computes the resulting collection B from the source collections A_i. The computation is based on a usual loop: changing A_i collections afterward does not change B. Conversely, an *active operation* is based on an active loop, and thus reevaluates B each time an addition or a removal occurs on A_i. One may find that using operator $=$ is ambiguous since it suggests bidirectionality; however, a change on B does *not* affect A_i. For this reason, operator $:=$ is used so that expression $B := op(A_1, ..., A_n)$ becomes not ambiguous.

2.2 Union, Intersection and Difference

Usual binary operations, such as union, intersection and difference, have simple active loops. For example with operation $C := A \cup B$, each time an element e is added into A or B, it is also added into C; conversely, each time e is removed for A or B, it is also removed from C:

Operation	Order	Active loop	
$C := A \cup B$	$(A,B) \notin \mathbb{O}^2$	$\forall^t e \in A,\, C + e$	$\forall^t e \notin A,\, C - e$
		$\forall^t e \in B,\, C + e$	$\forall^t e \notin B,\, C - e$
	$(A,B) \in \mathbb{O}^2$	$\forall^t e \in_i A,\, C +_i e$	$\forall^t e \notin_i A,\, C -_i$
		$\forall^t e \in_i B,\, C +_{\lvert A \rvert + i} e$	$\forall^t e \notin_i B,\, C -_{\lvert A \rvert + i}$

Union cannot preserve uniqueness since any element of A can also be in B: $C \notin \mathbb{U}$; it preserves the order only if A and B are ordered. Active loops for intersection and difference have been defined similarly.

2.3 Application

Application $B := A(f)$ consists of applying f on each element of A. It preserves the order and verifies $|A| = |B|$; it cannot preserve uniqueness since f may have introduced pairs. Application allows the definition of navigation *paths*. For example, if elements e of A define property[4] p, path $B := A.p$ is equivalent to $B := A(e \to e.p)$. However, the result must be flattened since $A(e \to e.p)$ returns a collection of properties, *i.e.* a collection of collections.The following table gives the active loops for applications and paths:

Operation	Order	Active loop	
$B := A(f)$	$A \notin \mathbb{O}$	$\forall^t e \in A,\ B + f(e)$	$\forall^t e \notin A,\ B - f(e)$
	$A \in \mathbb{O}$	$\forall^t e \in_i A,\ B +_i f(e)$	$\forall^t e \notin_i A,\ B -_i$
$B := A.p$	$A \notin \mathbb{O}$	$\forall^t e \in A,$ $\forall^t e' \in e.p,\ B + e'$ $\forall^t e' \notin e.p,\ B - e'$	$\forall^t e \notin A,$ stop observation of $f(e)$ $\forall e' \in f(e),\ B - e'$
	$A \in \mathbb{O}$	$\forall^t e \in_i A,$ $\forall^t e' \in_j e.p,\ B +_k e'$ $\forall^t e' \notin_j e.p,\ B -_k$	$\forall^t e \notin_i A,$ stop observation of $e.p$ $\forall e' \in_j e.p,\ B -_k$
		where $k = j + \sum_{n=0}^{n=i-1} \lvert e[n].p \rvert$	

2.4 Selection

Selection $B := A[f]$ consists of selecting elements of A that match predicate f. It preserves uniqueness and order since it filters A. Other operations can be derived from selection, such as operations *reject*, *detect*, *exists* and *forAll* defined by OCL [8], or operation $B := toUnique(A)$ that converts $A \in \mathbb{C}$ into $B \in \mathbb{U}$.

Operation	Order	Active loop	
$B := A[f]$	$A \notin \mathbb{O}$	$\forall^t e \in A \,/\, f(e),\ B + e$	$\forall^t e \notin A \,/\, f(e),\ B - e$
	$A \in \mathbb{O}$	$\forall^t e \in_i A \,/\, f(e,i),\ B +_j e$	$\forall^t e \notin_i A \,/\, f(e,i),\ B -_j$
		where $j = \mid A[0..i][f] \mid$	

As one can note, the previous active loops do not take into account any reevaluation of f required in some situations. For example, selection $persons[p \to p.age < 18]$ returns a collection of persons under 18. The active loop works fine whenever a person is added or removed from the collection; however, it fails whenever the age of a person goes above 18.

[4] A property is either a relation or an attribute. As explained in section 3.1, all properties are considered as collections.

To solve such an issue, we propose to *reify* predicate f into a *predicate collection* represented as a sequence of booleans. Let us consider that collection *persons* contains three people with ages 16, 42 and 12. Expression $persons.age$ $[a \rightarrow a < 18]$ returns predicate collection $(true, false, true)$ indicating that $persons[0]$ and $persons[2]$ are below 18. Here we assume that all collections, including unordered ones, store their elements in an array (see section 4.1), thus allowing the use of the indexed accessor $C[i]$ and indexed loops $\forall^t e \in_i C$. By *overriding* operation $B := A[f]$ with $B := A[P]$ where P is a predicate collection, the desired selection can be performed: $persons[persons.age[a \rightarrow a < 18]]$. The following table *replaces* the previous one where $A[exp]$ returns a predicated collection if exp is a function, or the collection resulting from the active selection if exp is a predicate collection:

Operation	Order	Active loop	
$P := A[f]$		$\forall^t e \in_i A,\, P +_i f(e)$	$\forall^t e \notin_i A,\, P{-_i}$
$B := A[P]$	$A \notin \mathbb{O}$	$\forall^t p \in_i P \,/ p,\, B + A[i]$ $\forall^t p \in_i P \,/ \neg p,\, B - A[i]$	$\forall^t e \notin_i A \,/\, e \in B,\, B - e$
	$A \in \mathbb{O}$	$\forall^t p \in_i P \,/ p,\, B +_j A[i]$ $\forall^t p \in_i P \,/ \neg p,\, B{-_j}$	$\forall^t e \notin_i A \,/\, e \in_j B,\, B{-_j}$
		where $j = \vert P[0..i][p \rightarrow p] \vert$	

The addition rule of $B := A[P]$ is based on observing P but not A, which implies that $P := A[f]$ must be computed *before* $B := A[P]$. The removal rule is based on observing A but not P, which allows retrieving element e of A that must be removed from B. Predicate collections can be combined through usual boolean operators:

Operation	Active loop	
$P' := \neg P$	$\forall^t p \in_i P,\, P' +_i \neg p$	$\forall^t p \in_i P,\, P'{-_i}$
$P := P_1 \wedge P_2$	$\forall^t p_2 \in_i P_2,\, P +_i (P_1[i] \wedge p_2)$	$\forall^t p_2 \notin_i P_2,\, P{-_i}$
$P := P_1 \vee P_2$	$\forall^t p_2 \in_i P_2,\, P +_i (P_1[i] \vee p_2)$	$\forall^t p_2 \notin_i P_2,\, P{-_i}$

These active loops are simple. However, it is necessary to decide which collection P_1 or P_2 must be observed for operators $\wedge$ and $\vee$. By convention, we fix that P_1 is defined *before* P_2 so that a change on P_1 is followed by a change on P_2; rules are thus based on observing P_2, which guaranties that $|P_1| = |P_2|$ when the rule is called ($P_1[i]$ can thus be used).

2.5 Sort

Sort $B := A\{f\}$ consists of sorting A accordingly to the value returned by f assuming that its type defines operator $<$. Sort operation preserves uniqueness but, naturally, not the order.

Operation	Order	Active loop	
$B := A\{f\}$	$A \notin \mathbb{O}$	$\forall^t e \in A,\, B +_j e$	$\forall^t e \notin A,\, B{-_j}$
	$A \in \mathbb{O}$	$\forall^t e \in_i A,\, B +_j e$	$\forall^t e \notin_i A,\, B{-_j}$
		where $j = \vert A[e' \rightarrow f(e') < f(e)] \vert$	

As for selection, the previous active loops do not take into account any reevaluation required if f uses paths on e. For example, sort $persons\{p \rightarrow p.name\}$ returns a collection of persons sorted by their name. This works fine whenever a person is added or removed from the collection but fails whenever the name of a person changes.

To solve such an issue, we propose again to *reify* function f into an *order collection* represented as a sequence of integers that gives positions after the sort. Let us consider that collection *persons* contains three persons named "Emma", "Oliver" and "Alice". Expression $persons.name\{n \rightarrow n\}$, abbreviated on $\{persons.name\}$, returns order collection $(1, 2, 0)$, which means that $persons[0]$ representing "Emma" will occupy position $\{persons.name\}[0] = 1$ after sorting. The order can then be used for sorting the collection:$persons\{\{persons.\ name\}\}$. The following table *replaces* the previous one where $O := \{A\}$ returns an order collection and $B := A\{O\}$ sorts A according to order O:

Operation	Active loop	
$O := \{A\}$	$\forall^t e \in_i A,\ O +_i j$ where $j = \|A[e' \rightarrow e' < e]\|$	$\forall^t e \notin_i A,\ O -_i$
$B := A\{O\}$	$\forall^t j \in_i O,\ B +_j A[i]$	$\forall^t j \notin O,\ B -_j$

Active loop of $O := \{A\}$ consists of adding or removing order j at/from position i. However, operation $+$ and $-$ must be refined for order collections to manage resulting positions correctly. For example, $O +_i j$ requires to increment (silently) all orders greater than j.

Moreover, the previous active loops do not allow sorting on multiple criteria. The full version is based on *partial order collections* that specify all *possible* positions; a partial order collection is represented by a sequence of sequences of integers. The previous example can be extended so that the persons are sorted by their last names and then by their first names: $persons\{\{persons.lastName\} \wedge \{persons.firstName\}\}$. Let us now consider that collection *persons* contains three persons named "Emma G.", "Oliver B." and "Alice B.". We now have $\{person.\ lastName\}$ that returns $((2), (0, 1), (0, 1))$ and $\{person.firstName\}$ that returns $((1), (2), (0))$: "Oliver B." and "Alice B." have the same possible positions (0 or 1) represented by the two sequences $(0, 1)$, and final positions are given by combining the two partial order collections: $\{persons.lastName\} \wedge \{persons.firstName\} = ((2), (1), (0))$.

2.6 Reversed Assignment

Previous operations are unidirectional: in operation $B := op\,A$, modifying A induces a change on B, but modifying B does not induce any change on A, thus motivating the use of operator $:=$ instead of $=$. Bidirectionality implies that op is reversible, *i.e.* $A := op^{-1} B$, so that a change on B impacts collection A. Union, intersection, difference, selection and sort are not reversible; the only operation that can be reverted is the application: $B := A(f)$ can be reverted as soon as f^{-1} exists.

However, application $B := A(f)$ and its reversed version $A := B(f^{-1})$ cannot be defined together since they both create a *new* resulting collection (respectively B and A). We thus introduce the *reversed assignment* operator (symbol =:) that can only be used on applications: application $B := A(f)$ creates B from A, while its reversed version $B =: A(f)$ (also written $B(f^{-1}) =: A$) allows the reverse *update*. Since A is always initialized before the reversed assignment, its addition rule *must only* be called subsequently to additions. Having $B =: A(f)$ implies that $B := A(f)$: we use operator $=$ so that $B = A(f)$ defines a *bidirectional application*. Active loops for reversed applications are defined as follows:

Operation	Order	Active loop	
$B =: A(f)$	$(A, B) \notin \mathbb{O}^2$	$\forall^t e \in B,\ A + f^{-1}(e)$	$\forall^t e \notin B,\ A - f^{-1}(e)$
	$(A, B) \in \mathbb{O}^2$	$\forall^t e \in_i B,\ A +_i f^{-1}(e)$	$\forall^t e \notin_i B,\ A -_i$

Navigation throughout collections is based on the flattening version of the application. In order to preserve the semantics of the active loop of path $B := A.p$ (see section 2.3), the active loop of the reverse path assignment $B =: A.p$ should define the following addition rule: $\forall^t e' \in B,\ A + e_n,\ e_n.p + e'$ where e_n is the owner element of property p that contains e'. Figure 1 helps in understanding this rule.

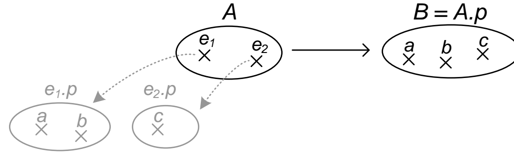

Fig. 1. Path principle

Such a definition has no general meaning: if e_n is already contained in A, which one is it (*e.g.* e_1 or e_2 of figure 1)? if not, to what corresponds e_n? This demonstrates that the path operation is not reversible since the transformation loses the required information due to the flattening. However, if $|A| = 1$ at any time, $e_n = A[0]$ necessarily: in this specific case, operations $B := A.p$ and $B =: A.p$, written $B = A.p$, mean that property p of singleton A equals B at any time. Operation $B = A.p$ is very useful to "bind" a property to another property. Reversed path assignment is defined as follows:

Operation	Order	Active loop	
$B =: A.p$	$(A, B) \notin \mathbb{O}^2$	$\forall^t e \in B,\ A[0].p + e$	$\forall^t e \notin B,\ A[0].p - e$
with $\|A\| = 1$	$(A, B) \in \mathbb{O}^2$	$\forall^t e \in_i B,\ A[0].p +_i e$	$\forall^t e \notin_i B,\ A[0].p -_i$

The following case study includes such a reversed path assignment.

3 Case Study

This section illustrates how previous active operations can be used for implementing active transformations. We first motivate the use of collections for representing any object property. We then give the active operations required for implementing an active transformation in the context of a user interface. We finally explains how the active transformation has been successfully implemented within Kermeta using our *Active Kermeta* framework.

3.1 Prerequisite: all Properties are Collections

An object property can be either a relation or an attribute, and is *always* represented by a collection. This means that, if the property has a cardinality 0..1 or 1..1, its representing collection is a *singleton*. In such a case, an empty collection represents a null property value, and replacing a property value consists of removing its old value then adding the new value. Using collections for representing any property allows a unified use of active operations, independently of the property cardinalities.

3.2 Active Transformation

Figure 2 gives the outline of the sample transformation: the left part represents source domain data, a directory of contacts; the right part represents the associated user interface (UI) that displays the contacts within a list widget, and allows editing contact properties throughout three text fields[5].

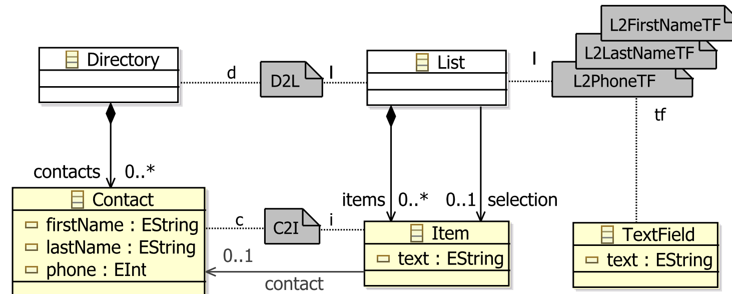

Fig. 2. Transformation outline

Linking domain data to UI is usually performed using "UI bindings" that are platform dependent and offer limited features. Moreover, creating a new

[5] UI objects are rendered through a graphical server not represented in the figure. Adding a contact is achieved through the button "Add" of the user interface: clicking on the button creates a new contact in the source data directly..

graphical component, which is often required when designing highly graphical applications, requires programming the binding schema for the component. Using active operations avoids such drawbacks, and addresses a more general problem than UI binding. A complex example is provided within the Active Kermeta framework, including the use of a new graphical component.

In the example of figure 2, *D2L* transforms directory d into list l that displays the contacts sorted by their last name and first name:

$$l.items := d.contacts\{\{d.contacts.lastName\} \wedge \{d.contacts.firstName\}\}(C2I)$$

C2I transforms each contact c into an item i that displays his/her first and last name, and also saves the link between c and i in reversed relation *contact*:

$$i.text := c.firstName + "\ " + c.lastName$$
$$i.contact := c$$

Operation +, not presented in this paper, is also active for a *String* singleton (see section 3.3). Relation *contact* allows a *reversible navigation* from the transformation target of the transformation source: $i = C2I(c)$ and $c = C2I^{-1}(i) = i.contact$. Such a relation is called a *trace*.

L2FirstNameTF (respectively *L2LastNameTF*) transforms the first name (respectively the last name) of the selected contact (relation *l.selection.contact*) into text-field *tf* in a bidirectional way:

$$tf.text = selection.contact.firstName$$

Finally, *L2PhoneTF* performs the same transformation but adds a bidirectional conversion between the *phone* number (an *Integer*) and the text field content (a *String*):

$$tf.text := selection.contact.phone(IntegerToString)$$
$$tf.text(StringToInteger) =: selection.contact.phone$$

3.3 Kermeta Implementation

Active operations have been implemented on top of Kermeta by the *Active Kermeta* framework, freely available at `http://gri.eseo.fr/software/activekermeta`. The framework proposes two packages dedicated to the four collection classes *Set*, *OrderedSet*, *Bag* and *Sequence*. Package *kermeta::observable* defines the minimal set of operations and active loops for these classes; the following table gives the syntax of Kermeta active loops:

$\forall^t e \in C$, ...	C.eachAdded{e|...}	$\forall^t e \in_i C$, ...	C.eachAddedAt{e,i|...}
$\forall^t e \notin C$, ...	C.eachRemoved{e|...}	$\forall^t e \notin_i C$, ...	C.eachRemovedAt{e,i|...}

Package *kermeta::active* implements active operations based on the active loops presented in section 2; the following table gives the syntax of Kermeta active operations:

$C := A \cup B$	C := A.union(B)	$B := A(f)$	B := A.collect{e|f(e)}
$C := A \cap B$	C := A.intersection(B)	$B := A.p$	B := A.path{e|e.p}
$C := A - B$	C := A.difference(B)	$B =: A.p$	A.assignPath{e|e.p}.from(B)
$P := A[f]$	P := A.predicate{e|f(e)}	$O := \{A\}$	O := A.sortOrder()
$B := A[P]$	B := A.select(P)	$B := A\{O\}$	B := A.sortedBy(O)
$P' := \neg P$	Pbis := P.not()	$O := O_1 \wedge O_2$	O := O1.and(O2)
$P := P_1 \wedge P_2$	P := P1.and(P2)		
$P := P_1 \vee P_2$	P := P1.or(P2)		

Writing an active Kermeta transformation respects the same principle than writing a usual Kermeta transformation: aspects are used to add new transformation operations on Ecore models [7]. The transformation of figure 2 has been implemented with active operations as follows:

```
1  aspect class Directory {
2     operation D2L(): List is do
3        result := List.new
4        result.items := contacts.sortedBy(
5           contacts.path{c|c.lastName}.sortOrder().and(
6              contacts.path{c|c.firstName}.sortOrder()
7           ).collect {c|c.C2I()}
8        )
9     end
10 }
11
12 aspect class Contact {
13    operation C2I(): Item is do
14       result := Item.new
15       result.text := firstName.plusValue("␣").plus(lastName)
16       result.contact.add(self)
17    end
18 }
19
20 aspect class List {
21    operation L2FirstNameTF(): TextField is do
22       result := TextField.new
23       var contact: Set<Contact> init selection.path{i|i.contact}
24       result.text := contact.path{c | c.firstName}
25       contact.assignPath{c|c.firstName}.from(result.text)
26    end
27
28    operation L2PhoneTF(): TextField is do
29       result := TextField.new
30       var contact: Set<Contact> init selection.path{i|i.contact}
31       result.text := contact.path{c|c.phone}.collect (a|a.toString())
32       contact.assignPath{c|c.phone}.from(result.text.collect {t|t.toInteger()})
33    end
34 }
```

Operation *plus* (line 15), not described in this paper, is an active operation that concatenates two string singletons; its companion operation *plusValue* concatenates the literal string with the string singleton. Relation *contact* is added through an aspect of class *Item*:

```
aspect class Item {
   reference contact: Set<Contact>
}
```

As one may note, such a Kermeta code can be easily generated from the formal specification given in the previous section.

4 Evaluation

This section evaluates the worst case complexities of active loop rules, discusses the resulting performance in the contexts of both constraint evaluation and model transformation, and then compares the approach with related works.

4.1 Worst Case Complexities

We can infer worst case complexities of active loops from the following usual complexities of array based implementation of collections:

Operation	$C \notin \mathbb{U}$	$C \in \mathbb{U}$
$\|C\|$, $e \in_i C$, $C[i]$	$\mathcal{O}(1)$	
$C + e$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
$e \in C$, $C +_i e$, $C - e$, $C -_i$	$\mathcal{O}(n)$	

Since active loops of ordered collections differ from those of not ordered collections, we must study complexities for each of the four collection types (*bag*, *seq*, *set* and *oset*). Moreover, we distinguish three cases of the active construction of collections: the initialization ("i.") that invokes the addition rule n times; the addition ("a.") that invokes the addition rule on each addition performed in the source collection; and the removal ("r.") that invokes the removal rule on each removal performed in the source collection. The following table synthesizes complexities for each rule of action loops:

	bag			seq		set			oset	
Operation	i.	a.	r.	i.	a./r.	i.	a.	r.	i.	a./r.
$A \cup B$, $A(f)$, $A.p$	n	1	n	n	n	n	1	n	n	n
$A\{O\}$, $A[f]$, $A[P]$	n	1	n	n	n	n^2 (n)	n (1)	n	n^2 (n)	n
$B =: A.p$	-	1	n	-	n	-	1	n	-	n
$\neg P$, $P_1 \wedge P_2$, $P_1 \vee P_2$, $O_1 \wedge O_2$	-	-	-	n	n	-	-	-	-	-
$A \cap B$, $A - B$, $\{A\}$	n^2	n	n	n^2	n	n^2	n	n	n^2	n

Complexity for an initialization is not necessarily equal to $n \times c$ where c is the complexity of its addition rule. For example, the addition rule of $A \cup B$ where $(A, B) \in seq^2$ is based on operation $C +_i e$ that costs $\mathcal{O}(n)$: the initialization would thus cost $\mathcal{O}(n^2)$; however, operation $C +_i e$ here appends e at the *end* of C, which costs only $\mathcal{O}(1)$: the initialization phase thus costs $\mathcal{O}(n)$.

Operations in the first four rows are mainly $\mathcal{O}(n)$: additions cost $\mathcal{O}(1)$ for unordered collections, but cost $\mathcal{O}(n)$ for ordered ones because of the required shifting; removals always cost $\mathcal{O}(n)$ since they require a search in unordered collections, or a shifting of ordered collections. Note that the complexity of operation $A.p$ is given considering $|A.p| = n$, but not $|A| = n$. Collections with uniqueness have $\mathcal{O}(n^2)$ operations because $C + e$ must ensure the uniqueness. However, operations on row 2 do not require uniqueness from operation $+$: for example, selection $B := A[f]$ guaranties that $B \in \mathbb{U}$ if $A \in \mathbb{U}$ since it filters A; these operations can thus reduce their complexities (values surrounded by parenthesis). The last row has $\mathcal{O}(n^2)$ complexities: $A \cap B$ and $A - B$ require presence tests that cost $\mathcal{O}(n)$, and $\{A\}$ naturally requires two loops.

4.2 Discussion

The previous table illustrates that complexities for removals are always $O(n)$. Consequently, replacing element e_a by element e_b in collection C costs $O(n)$ since it performs a removal of e_a and an addition of e_b. This might be optimized for ordered collections since replacing one of its element only costs $O(1)$. This optimization requires the introduction of a *replace rule* within active loops. Moreover, replace rule can be mandatory in some circumstances; for example, replacing the value of a property with cardinality 1..1 violates the minimal cardinality constraint if a removal if performed. The replace rules can be easily inferred from addition and removal rules; they have not been considered in this paper for clarity.

An active transformation that counts m operations constructs its initial result in between $\mathcal{O}(m \times n)$ and $\mathcal{O}(m \times n^2)$, and subsequently updates each *individual* resulting collection in between $\mathcal{O}(1)$ and $\mathcal{O}(n)$. However, operations are not independent: modifying one source collection can result in multiple chained updates. A simplified view of such dependencies consists of representing the operations in a two dimensioned space where height h counts the independent chains of operations, and w counts the operations involved in a chain: $\mathcal{O}(m) = \mathcal{O}(h \times w)$. The number of operations Δw required to reify functions involved in selections and sorts (see sections 2.4 and 2.5) is included in number w. For example, single selection $persons(p \rightarrow p.age < 18)$ must be rewritten in active selection $persons(persons.age(a \rightarrow a < 18))$ that counts $\Delta w = 2$ (path + predicate collection), *i.e.* $w = 3$.

Complexity of the initial construction varies from $\mathcal{O}(h \times w \times n)$ to $\mathcal{O}(h \times w \times n^2)$, and complexity of the subsequent chained updates varies from $\mathcal{O}(w)$ to $\mathcal{O}(w \times n)$: their ratio λ thus varies from $\mathcal{O}(h)$ to $\mathcal{O}(h \times n^2)$. This result illustrates the interest of active operations, especially for large models ($n \gg 1$) and/or complex transformations ($h \gg 1$).

The active loops proposed in section 2 should be implemented in a way that depends on the *context of use*, thus allowing possible optimizations that would increase λ by reducing width w of the dependency chains. In the context of incremental constraint evaluation, Cabot and Teniente [4] propose to take into account *only* changes that can induce constraint violation, thus defining a new specific context of use. Such a specific context requires that our model for observing collections should be refined so that addition and removal rules does not systematically invoke their associated procedures. Section 3 has presented the specific context of active transformations for user interfaces: the transformation binds the domain objects to their presentation. In such a context, the user works on a presentation that represents a (small) part of the full application model: this typically means that the transformation starts by a *selection* that filters the full model. Such a selection thus naturally "optimizes" the transformation by pre-filtering changes that do not impact the presentation. Moreover, the user can perform many changes on a single object in a short time-slot, thus resulting in many reevaluations of active operations. Once again, the observability model can be refined by using an asynchronous invocation of addition and removal rules, as done within Viatra [9]. This would improve performances by filtering any redundant modifications, such as multiple intermediate changes of a single property (only the last change should be considered).

Since it is centered on operations on collections, our approach is more suited to imperative transformations (*e.g.* Kermeta) than to declarative ones (*e.g.* ATL [10]). However, we think that our formalism can *help* in making declarative transformations active. Moreover, mappings expressed with higher level languages, such as Malan [11], should be automatically converted into active operations.

4.3 Related Works

Many research works have been done on incremental evaluation of constraints and incremental transformations. We herein only cite some of the most recent ones.

Blanc *et al.* propose an original approach for detecting model inconsistency (constraint violation): the detection is performed on the model considered as a sequence of elementary construction operations, rather than a model considered as a set of elements [12]. The approach is thus naturally incremental. It shares some similarities with active operations: their elementary construction operations match the elementary rules (addition and removal) of active loops, and our addition rules are also used to initially build the content of collections. However, their approach is dedicated to constraint evaluation only, and the implementation is based on Prolog which is not widely used and not well adapted to MDE.

Cabot et Teniente optimize OCL constraint evaluation by considering only constraint violations: model changes that cannot violate constraints are filtered [4]. They also translate OCL contexts to better contexts. The proposed optimization is interesting and forms a specific context of use, as previously explained. However, users often temporally violate constraints when editing models (*e.g.*

a user omits the type of a class attribute within an Ecore diagram): transition from state *violated* to state *respected* should not be ignored. The optimization, specific to constraint checking, cannot be used in the context of incremental transformation.

XSLT is probably the best known transformation language. Villard and Layaïda have developed incXLST, an incremental XSLT processor, thus showing the broader interest of incremental transformation [5]. The processor is based on re-instantiating transformation rules and merging the resulting fragments within the target document, and has limited featured. Framework eXAcT allows the transformation of DOM documents into DOM presentations (*e.g.* SVG presentations) [13]. However, eXAcT transformations are complex Java programs with limited features. Moreover, both incXSLT and eXAcT are not MDE tools.

QVT has established that incremental transformation is an important issue of MDE [14], but no incremental QVT-based transformation engine has been implemented yet. Xiong *et al.* proposed SyncATL, an incremental ATL processor [6], on the same principle than incXSLT: elicited ATL rules are re-executed and their results are merged with the target. As for incXSLT with XSLT, the processor is dedicated to ATL only. Hearnden *et al.* propose an original approach based on the use of SLD resolution, where SLD trees store the transformation context and dependency tables record dependencies between the transformation and the source model [2]. The drawback of the approach is the maintenance cost of the SLD trees and dependency tables.

The previous works make *declarative* transformations incremental by implementing new processors and/or algorithms tied to specific languages and/or systems. Using active operations on collections allows their direct execution on model instances, without requiring any specific processor or complex algorithm: *definition* of operations are directly *executable* in an active manner. We have shown that active operation can be easily used to implement Kermeta *imperative* transformations. We think that our formalism can *help* in making declarative transformation languages active, such as ATL [10], by generating the active operations for a given "passive" transformation. Some authors considered that declarative transformations should be expressed as mappings [3,1,11]. Here again, we think that active implementations of mappings, as defined by Akehurst [3], can be achieved by active operations.

5 Conclusion and perspective

This paper proposes a formalism, based on active loops, for implementing active operations on collections. The standard set of operations, mainly inspired by OCL [8], is supplemented by a reversed assignment that allows the definition of bidirectional operations. A case study, fully implemented in Kermeta, illustrates that making a transformation active by using such a formalism does not require to change much the usual (*i.e.* passive) transformation; it also gives a specific context that requires active transformations with bidirectionality features: user interfaces. The complexity study shows that running active operations results

in an interesting gain when compared to running all the "passive" operations. Moreover, such a gain can be increased by reducing operation dependencies with optimization strategies that can be implemented depending of the contexts of use (*e.g.* transformation within UI or evaluation of constraint violation).

We first plan to create an active implementation of the Malan language [11] based on *Active Kermeta*, thus showing the ability of active operations to implement declarative mappings and transformations. We will secondly focus on the use of active operations for incremental constraint validation by extending the proposed set of active operations (*e.g.* OCL function *isUnique*), by implementing them in the Active Kermeta framework, and by defining active class invariants through Kermeta aspects. We finally plan to enhance the collection observability model of *Active Kermeta* with filtering and asynchronous treatment capabilities, so that the execution time of active operations can be optimized depending on their context of use.

References

1. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling **8**(1) (2008) 21–43
2. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: MoDELS 2006, LNCS 4199. (2006) 321–335
3. Akehurst, D.H.: Model Translation: A UML-based specification technique and active implementation approach. PhD thesis, University of Kent (2000)
4. Cabot, J., Teniente, E.: Incremental evaluation of OCL constraints. In: CAiSE 2006, LNCS 4001. (2006) 81–95
5. Villard, L., Layaïda, N.: An incremental XSLT transformation processor for XML document manipulation. In: Proceedings of WWW '02, ACM Press (2002) 474–485
6. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proceedings of ASE '07, ACM Press (2007) 164–173
7. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proceedings of MoDELS'05, LNCS 3713. (2005)
8. Warmer, J.B., Kleppe, A.G.: The object constraint language: getting your models ready for MDA. Addison-Wesley
9. Varró, D., Balogh, A.: The model transformation language of the viatra2 framework. Sci. Comput. Program. **68**(3) (2007) 187–207
10. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at MoDELS 2005, LNCS 3844, Springer (2006) 128–138
11. Blouin, A., Beaudoux, O., Loiseau, S.: Malan: A mapping language for the data manipulation. In: Proceedings of DocEng '08, ACM Press (2008) 66–75
12. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Proceedings of ICSE '08, ACM Press (2008) 511–520
13. Beaudoux, O.: XML active transformation (eXAcT): transforming documents within interactive systems. In: Proceedings of DocEng '05, ACM Press (2005) 146–148
14. OMG: Mof qvt final adopted specification. Omg document, OMG (2005)

Specifying and implementing UI Data Bindings with Active Operations

Olivier Beaudoux
ESEO Group, GRI Team
Angers, France
olivier.beaudoux@eseo.fr

Arnaud Blouin
INRIA, Triskell Team
Rennes, France
arnaud.blouin@inria.fr

Olivier Barais
Jean-Marc Jezequel
Univ. of Rennes 1, Triskell
(barais, jezequel)@irisa.fr

ABSTRACT
Modern GUI toolkits propose the use of declarative data bindings to link the domain data to their presentations. These approaches work fine for defining simple bindings, but require an increasing programming effort as soon as the bindings become more complex. In this paper, we propose the use of active operations for specifying and implementing UI data bindings to tackle this issue. We demonstrate that the proposed approach goes beyond the usual declarative data bindings by combining the simplicity of the declarative approaches with the expressiveness of active operations.

ACM Classification Keywords
D.2.2 Software Engineering: Design Tools and Techniques—*Computer-aided software engineering (CASE), User interfaces*

General Terms
Algorithms, Design, Languages

Author Keywords
Data binding, active operation, GUI

1. INTRODUCTION
The problem of linking domain data to their presentations has appeared very early in computer science. The Model-View-Controller (MVC) design pattern has been introduced with the SmallTalk-80 language to formalize and implement such a linking [13]. The main drawback of this pattern is that the view is bound to a specific model: the view has to observe the model and refresh its state on model changes. Consequently, applications must adapt their domain data to the specific models bound to their possible views. The Java *Swing* API applies such a schema; for example, displaying a collection of elements within a *JTable* requires adapting the collection by implementing interface *TableModel*[8].

The concept of data binding can be seen as an evolution of MVC that avoids such a drawback. A data binding is a "controller" that binds the model and the view: it observes the model and updates the view whenever the model changes; conversely, it observes the view and updates the model whenever the view changes. The concept of data binding is very close to the controller of the PAC model where the Controller binds the Abstraction and the Presentation [6]. Recent GUI toolkits propose the data binding as the first-class object for linking models and views. However, as this paper will illustrate, they all suffer from two main limitations: 1) they offer a limited expressiveness so that hand-programming is often required when the complexity of the bindings grows; 2) they are platform-dependent implying that a data binding written within a given GUI toolkit must be entirely rewritten to be used within another one.

In this paper, we propose the use of active operations for specifying and implementing UI data bindings. The concept of active operation extends the usual concept of operation by allowing the result of an operation to be re-evaluated afterward. For example, usual operation $b := a.select(f)$ constructs collection b containing each element of collection a that satisfies predicate f [18]; the active version of this operation does more: it adds into b (respectively removes from b) any element newly added into a (respectively removed from a) that satisfies f. Active operations formalize and generalize our initial work on active transformations in the context of GUI [3, 2]; the mathematical definition of action operations is provided in [4].

Our proposal aims at combining the simplicity of the declarative approaches with the expressiveness of active operations, which is achieved through: a *unified* and *platform-independent* language for specifying data bindings; the use of simplified *class diagrams* (Ecore documents) for representing both the models and the views; a formalism for *implementing* the specification of bindings independently from the final implementation language; a *compatibility* schema with GUI toolkits.

The remainder of this paper is structured as follows. Section 2 presents, through three concrete examples, our language used to specify data bindings with active operations. Section 3 explains why and how the speci-

fication of active operations must be translated before being implemented. Section 4 focuses on the implementation and execution of active operations on GUI platforms. Section 5 introduces works related to data binding. Section 6 complements section 5 by comparing our proposal with the declarative data bindings of three representative Rich Internet Application (RIA) toolkits. Section 7 finally concludes on our work and its perspective.

2. SPECIFYING ACTIVE OPERATIONS

This section introduces active operations through three complementary examples. These examples have been carefully designed to illustrate the expressiveness of active operations for specifying various UI data bindings. They have also been motivated to illustrate the limitation of usual binding mechanisms (see section 6).

The following specifications of active operations are expressed using our own domain specific language (DSL) that is close to the OCL language [18].

2.1 Example 1 - A Simple XML Editor

Figure 1 gives the model of a simple XML editor. The left part represents a simplified XML document model; the right part represents the model of a tree widget; the middle part represents bindings between these two models.

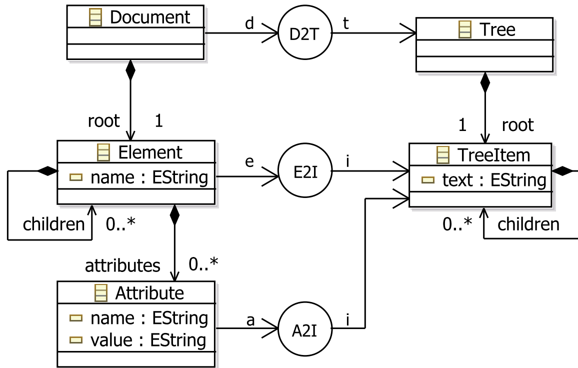

Figure 1. A Simple XML Editor

Binding *D2T* binds XML document *d*, instance of class *Document*, to tree widget *t*, instance of class *Tree*, as follows:

```
t.root := d.root.map(E2I)
```

Expression *c.map(f)* applies function *f* to each element *e* of collection *c*, and maintains a link (called a *trace*) between *e* and the element returned by *f(e)*. In the example, the root element of document *d* is bound to the root item of tree *t* through binding *E2I* defined as follows:

```
1  i.text := "<" + e.name + ">"
2  i.children :=
3      e.attributes.sortedBy(a | a.name).map(A2I) +
4      e.children.map(E2I)
```

The *text* of tree item *i* displays the *name* of element *e* surrounded by angle brackets. *Children* of item *i* result from concatenating *attributes* of element *e* sorted by their *name* and mapped to tree items (line 3), with *children* of element *e* recursively mapped to tree items (line 4). Finally, binding *A2I* displays the *name* and *value* of attribute *a* into tree item *i*:

```
i.text := "@" + a.name + "=" + a.value
```

All the previous bindings are *unidirectional*. The next example illustrates how active operations can be used to define *bidirectional* bindings.

2.2 Example 2 - A Directory Editor

Figure 2 gives the model of a directory editor. The left part represents a directory *d* of contacts; the right part represents the model of a list widget *l*; three text-fields *ff*, *lf*, and *pf*, are used to respectively edit the first-name, last-name and phone number of the contact selected from the list widget *l*; finally, a fourth text-field *tf* is used to filter the content of the list so that a user can quickly find a contact within a large directory.

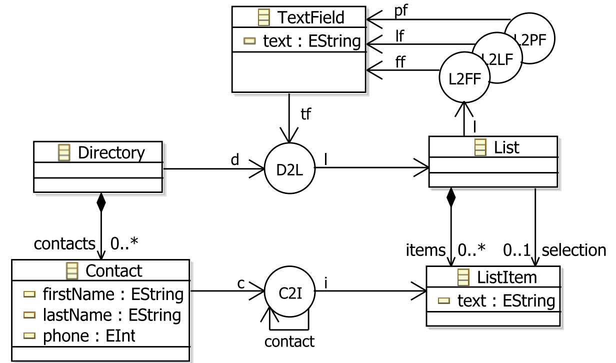

Figure 2. A Directory Editor

Binding *D2L* binds list of contacts *d*, instance of class *Directory*, to list widget *l*, instance of class *List*. The binding has a second argument *tf*, instance of class *TextField*, used to filter the list:

```
1  l.items := d.contacts
2      .sortedBy(c | c.lastName + c.firstName)
3      .select(c | tf.text.isEmpty() or
4                c.lastName.startsWith(tf.text))
5      .map(C2I)
```

The list of contacts is first sorted (line 2); a simple string concatenation is here used to specify a sort on the *lastName* and then on the *firstName* of the contacts. The resulting list is then filtered through the *select* operation (lines 3-4): if text-field *tf* is empty, all the contacts are selected; otherwise, only contacts whose last-name starts with the text-field content are selected. Each resulting contact *c* is then mapped to a list item *i* by calling binding *C2I* (line 5), which is defined as follows:

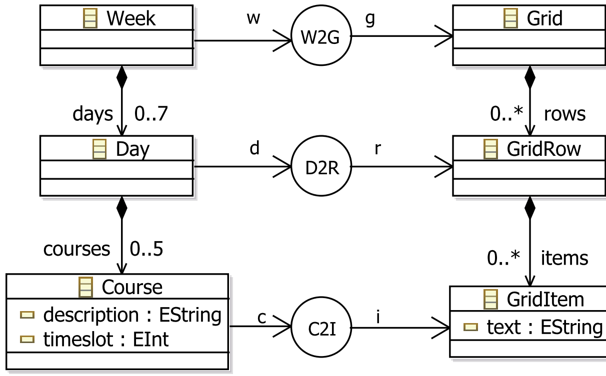

Figure 3. An academic calendar

```
1  i.text := c.firstName + " " + c.lastName
2  i.contact := c
```

This binding displays each contact in the list widget with their first and last names (line 1). A *trace* of binding *C2I*, represented by reverse arrow *contact* in figure 2, is then set (line 2).

Binding *L2PF* binds text-field *pf* used to edit the *phone* number to the selected item of the list *l*, as follows:

```
1  pf.text := l.selection.contact.phone.toString()
2  l.selection.contact.phone := pf.text.toInteger()
```

The *phone* number of the selected contact is converted into a string displayed into text-field *pf* (line 1); conversely, the *text* of text-field *pf* is converted into an integer that represents the phone number of the selected contact (line 2). Text-field *pf* must be designed to restrict the editing to integers; this must be achieved at the GUI platform level, but *not* by the binding itself. Binding *L2PF* is *bidirectional*: if the user changes the selection, the text-field is automatically updated; conversely, if the user changes the text-field content, the phone number of the selected contact is automatically updated. Infinite loop on such a bidirectional binding is avoided by only notifying changes on *new* values: if the new value does not differ from the previous one, no notification is performed by the property setter method.

Only path assignments, along with an optional conversion, can be used in a bidirectional way [4]. If no conversion is required between two properties, the double use of operator := defines a *bidirectional assignment* that can be shortened with the operator =, such as binding *L2FF* illustrates:

```
ff.text = l.selection.contact.firstName
```

Contrary to the previous binding, no conversion is required here. Binding *L2LF* is similar to *L2FF* for property *lastName*.

These bindings illustrate the bidirectionality feature of active operations; however, as within example 1, these bindings remain quite simple. The next example uses intensively active operations to achieve a complex binding.

2.3 Example 3 - An Academic Calendar

Figure 3 gives the model of an academic calendar. The left part represents a simplified model of a weekly academic calendar; the right part represents the model of an HTML-like table widget; the middle part represents bindings between these two models.

Binding *W2G* binds week w, instance of class *Week*, to grid widget g, instance of class *Grid*. It maps *days* of week w to *rows* of grid g by calling binding *D2R*:

```
g.rows := w.days.map(D2R)
```

Binding *D2R* binds day d to a grid row r, which consists of populating relation *r.items* from relation *d.courses*. Figure 4 synthesizes the principle of binding *D2R* through a simple example: collection *ts* contains all the predefined time-slots of the calendar (for example, time-slot 0 corresponds to slot 9:05-9:55am); day d contains two courses a and b respectively located on time-slots 1 and 4. The resulting grid-items *r.items* of grid-row r must contain empty grid-items (noted $i_\varnothing$) that represent the free time-slots, and non-empty grid-items (noted i_a and i_b) that represents courses a and b.

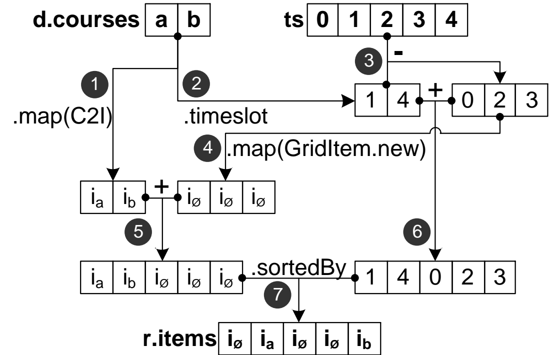

Figure 4. Principle of binding *D2R*

The collection (i_a, i_b) is computed by mapping courses *d.courses* through binding *C2I* ①. The collection of free time-slots $(0, 2, 3)$ is computed by subtracting *d.courses.timeslot*[1] equals to $(1, 4)$ ②, from collection *ts* representing all defined time-slots (0..4) ③. The collection of empty grid-items $(i_\varnothing, i_\varnothing, i_\varnothing)$ is computed by mapping collection of free time-slots with function *GridItem.new* ④. The two collections of grid-items are

[1]Expression *d.courses.timeslot* is a path, which is a specific operation detailed in section *3.1*.

then concatenated to form the collection of all grid-items $(i_a, i_b, i_\emptyset, i_\emptyset, i_\emptyset)$ ⑤. Such a collection is however not ordered accordingly to time-slots. Since time-slot numbers represent positions, the "order collection" $(1, 4, 0, 2, 3)$, resulting from the concatenation of a and b time-slots and the free time-slots ⑥, is used to sort the grid-items ⑦: operation *sortedBy* sorts collection $(i_a, i_b, i_\emptyset, i_\emptyset, i_\emptyset)$ with order $(1, 4, 0, 2, 3)$ that gives the position of each item after sorting.

The previous principle is quite complex due to the calendar model that does not divide a day into time-slots. However, this proposed model is a simplification of a calendar model that allows a time-slot to contain multiple courses and a course to span multiple time-slots. This simplified model has been designed to illustrate that active operations allow the definition of bindings between "something missing" (a time-slot without any course) to a UI object (an empty grid-item).

Binding *D2R* is specified with our DSL as follows, and illustrates the expressiveness of our DSL:

```
ts := seq(0 to 4)
courseTS := d.courses.timeslot
freeTS := ts − courseTS
items := d.courses.map(C2I) +
        freeTS.map(n | GridItem.new)
r.items := items.sortedBy(courseTS + freeTS)
```

Binding *C2I* finally binds the text of the not-empty grid-item i to the description of its corresponding course c, as follows:

```
i.text := c.description
```

3. TRANSLATING ACTIVE OPERATIONS

The previous section focuses on the specifications of bindings. However, as this section will illustrate, these specifications cannot be executed on a target platform as is: they must be *translated* into an active implementation. Such a translation uses again our own DSL by introducing new specific operations, thus making the translation process independent from the final implementation language and platform (see section 4). Moreover, some of the new operations motivated by the translation might be useful at the specification level, such as example 3 illustrates through its use of operation *sortedBy* (see section 2.3).

3.1 Paths

Paths extend the dotted notation of OOP so that the dot symbol can be applied more than once on collections [18]. For example, expression *d.contacts.firstName* represents a path that returns a *flattened* collection, as the OCL operation *collect* does [18], containing the firstname of all contacts of directory d. The dedicated operation *path* must be used for implementing path on collections on top of an OOP language; in binding *L2FF* of example 2, expression:

```
ff.text = l.selection.contact.firstName
```

must be *translated* into:

```
ff.text = l.selection
   .path(s | s.contact).path(c | c.firstName)
```

3.2 Observability

Usual loops can be used to implement usual operations on collections. For example, expression *b := a.collect(f)* computes collection b by converting each element e of collection a into element *f(e)* within collection b; operation *collect* is easy to implement using a loop, as follows:

```
a.each(e, i | b.add(f(e), i) )
```

The previous code states that each element e at position i within collection a must be converted to the corresponding element *f(e)* at the same position i within b.

Making an operation active, such as operation *collect*, requires the *observability* of collections: an observable collection is a collection from which additions and removals can be observed [1, 9]. We have proposed in [4] to extend usual loops with *active loops* that manage collection observability and allow implementing active operations as usual loops do for implementing the usual operations. For example, the previous usual loop that implements operation *collect* can be easily translated to the following active loop:

```
1  a.eachAdded(e, i | b.add(f(e), i) )
2  a.eachRemoved(e, i | b.remove(i) )
```

This code states that: each element e *newly* added into a results in adding the converted element *f(e)* into b (line 1); each element e *newly* removed from position i within collection a results in removing the corresponding element from b (line 2).

Such a translation process from usual loops to active loops works fine for operations *collect*, *map* and *path*. However, it must be augmented for operations *select* and *sortedBy*, as explained in the next section.

3.3 Selection and sort

The following expression:

```
minors := persons.select(p | p.age < 18)
```

computes collection *minors* that contains persons p under 18. The active loop of such a *select* operation can be easily translated from a usual loop. This loop works fine whenever a person is added or removed from collection *persons*; however, it fails whenever the age of a person goes above 18: this last change is not captured by the active loop. We thus propose to *reify* predicate function f involved in expression *c.select(f)* into a *predicate collection* represented as a sequence of booleans. The predicate collection related to the previous example is defined by the following expression:

```
persons.path(a | a.age).predicate(a | a < 18)
```

If *persons* contains three people with ages 16, 42 and 12, this expression returns predicate collection $true, false, true$. By *overriding* operation *select*, the previous example should be translated into:

```
minors := persons.select(
    persons.path(a | a.age).predicate(a | a < 18)
)
```

The same problem arises with operation *sortedBy*. For example, binding *E2I* of example 1 sorts attributes of element e by their name:

```
e.attributes.sortedBy(a | a.name)
```

If an attribute a is added or removed from *e.attributes*, the active loop updates correctly the resulting collection. However, any change to *a.name* is not captured by the active loop. We thus propose to *reify* the order function f involved in expression *c.sortedBy(f)* into an *order collection* represented as a sequence of integers. In the previous example, the order collection is defined by the following expression:

```
e.attributes.path(a | a.name).ascendingOrder()
```

If element e defines three attributes named "first", "second" and "last", the previous expression returns order collection $\{0, 2, 1\}$ that gives the position of names after the sort. By *overriding* operation *sortedBy*, the previous example should be translated into:

```
e.attributes.sortedBy(
    e.attributes.path(a | a.name).ascendingOrder()
)
```

Order collections are also useful for solving various ordering problems, independently from the translation phase. For example, they have been used in the specification of binding *D2R* to bind courses and free time-slots with grid-items (see section 2.3).

3.4 Tuples

Anonymous functions f involved in active operations, such as *b := a.select(f)*, may require more than one argument. For example, binding *D2L* includes a selection based on two arguments *tf* and c:

```
d.contacts.select(c |
    tf.text.isEmpty() or
     c.lastName.startsWith(tf.text)
)
```

The reified version of this *select* operation uses a *tuple* from which a *predicate* operation is performed, as follows:

```
d.contacts.select(
    (d.contacts.lastName, tf.text).predicate(n,t |
        t.isEmpty() or n.startsWith(t)
    )
)
```

By doing so, the predicate collection is updated whenever *d.contacts.lastName* or *tf.text* changes. Tuples can thus be used for extending the initial scope of operation select to multiple arguments; a similar extension can be applied to other active operations, such as operation *collect*[2].

4. IMPLEMENTING ACTIVE OPERATIONS

The previous section explains the translation of binding specifications to their active implementation, independently from the target GUI platform. This section explains how to make the final implementation *compatible* with a given GUI platform so that its widgets can be reused. It gives theoretical and experimental results regarding performances of the final implementation.

4.1 Implementing Active Operations on a GUI Toolkit

In order to evaluate the use of active operations for UI data binding, we have implemented active operations on top of the Flex GUI platform [12]. Flex has been chosen for its own internal binding mechanism, its rich set of widgets, and its ability to pass anonymous functions as function arguments. All the three examples presented in this paper have been implemented with the resulting *ActiveFlex* project; more details about the Flex implementation can be found within the project.[3].

Figure 5 gives an overview of our *ActiveFlex* model that extends Flex classes with active operation capabilities.

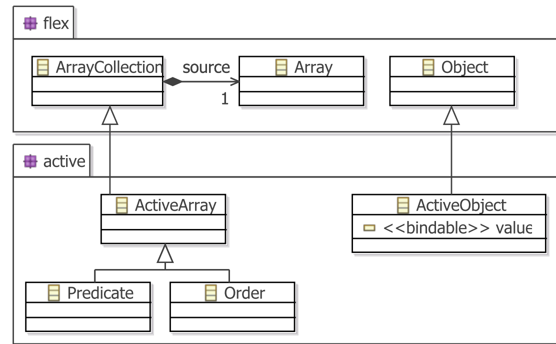

Figure 5. ActiveFlex model

The Flex API defines a built-in data binding mechanism accessible through classes *ArrayCollection* and *Object*. An *ArrayCollection* represents an observable and ordered collection of elements; it is bound to a source *Array* that holds the elements. Array collections can be bound to Flex widgets displaying collections to users, such as widget *List*. Consequently, our class *ActiveArray* inherits from this Flex *ArrayCollection* class so that instances of *ActiveArray* returned by active operations can be bound to such Flex widgets. Class *Predicate* defines predicate collections and adds usual boolean operations to class *ArrayCollection*; class *Order* defines or-

[2]Such an extension is however out of the scope of this paper.
[3]*ActiveFlex* is available under GPL license at `http://gri.eseo.fr/software/activeflex.html`.

der collections by providing operation *and* that allows sorting on multiple criteria.

Any class derived from the Flex class *Object* can mark any of its attribute as *bindable*: by doing so, the bindable attribute can be bound to Flex widgets displaying simple values, such as a text-field or a check-box. Our class *ActiveObject* thus defines its content through attribute *value* that is marked as bindable so that any active object can be bound to such Flex widgets.

The following MXML code[4] illustrates how simple the binding of the *TreeItem* of example 1 to a Flex *mx:Tree* widget is:

```
<mx:Tree dataProvider="{tree.root}" ... />
```

Object *tree* is an instance of our class *Tree*. The *mx:Tree* is bound to the *tree.root*, which is an instance of our *TreeItem* class; the *mx:Tree* then uses internally: the *children* property to recursively construct the tree content, and the bindable property *label* to display the content of each tree item. Any Flex widget can be bound to an instance of our classes *ActiveArray* or *ActiveObject* similarly.

Such a scheme can be applied to any RIA toolkit that proposes a declarative data binding mechanism, such as with WPF/Silverlight [16] or JavaFX [10]. Toolkits without data binding capabilities require more coding effort. For example, the use of active operations on top of the Swing toolkit requires implementing Java interfaces such as *TableModel*, *TreeModel* or *Document* within active collection and/or object classes.

4.2 Theoretical Complexities

Table 1 summarizes the best-case (C_{min}), average-case (C_{av}) and worst-case (C_{max}) complexities of active operations (see [4] for more details on computation of these complexities).

	C_{min}			C_{av}			C_{max}		
Operation	I	+	-	I	+	-	I	+	-
collect	n	1		n					
path	n	1		n					
select	n	1		n					
predicate	n	1		n					
not, and, or	n	1		n					
sortedBy	n	1		n					
asc.Order	n	1		$n.log_2\, n$	n		n^2	n	
and	n	1		n					
union	n	1		n					
difference	n	1		n^2	n		n^2	n	
intersection	n	1		n^2	n		n^2	n	

Table 1. Complexities of active operations

Columns labeled "I" give complexities for the initial construction of the operation result; columns labeled "+" (respectively "-") give complexities for updating the result when an addition (respectively a removal) occurs on the source collection. Initial constructions mainly cost a linear time since they globally use elementary operation *add* in the best-case (*i.e.* append); subsequent mutations also cost a linear time due the required shifting. These results only vary for sorting that requires two loops, and for difference/intersection that must check the presence with operation *contains* which costs a linear time.

[4]MXML is the Flex XML dialect used to specify user interfaces.

The previous complexities are asymptotic. However, the *translation cost* representing the amount of time and memory used by the intermediate operations that appear during the translation (see section 3) should be also taken into account. For example, expression:

```
c.select(e | f1(e.p1) and f2(e.p2))
```

must be translated into expression:

```
c.select(c.path(e|e.p1).predicate(f1) and
         c.path(e|e.p2).predicate(f2))
```

This last expression includes 2 paths ($n_{pa} = 2$), 2 predicates ($n_{pr} = 2$), plus 1 boolean expression ($n_{pr} - 1$), and thus costs 5 intermediate active collections and operations. A similar principle can be applied to operation *sortedBy* that can be based on multiple criteria (n_{cr}). Table 2 synthesizes such a cost.

Operation	Translation cost
union, diff., inter.	0
collect	0
path	1
select	$n_{pa} + 2 \times n_{pr} - 1$
sortedBy	$n_{pa} + 2 \times n_{cr} - 1$

Table 2. Translation cost

The previous complexities of *individual* operations can be used to compute complexities of an *overall* binding. Let us consider the binding defined by example 1 (the XML editor) for a source XML document with a depth d, a number of child elements per element n_e, and a number of attributes per element n_a, both common to all elements. For simplification purpose, we consider that $n_e = n_a = n$. Document thus counts $N_e = \frac{n^d - 1}{n-1}$ elements, and $N_a = n \times N_e$ attributes; the tree widgets counts $N_i = N_a + N_e$ items. By considering that $N = n^d \gg 1$, we have: $N_e \simeq n^{d-1} = N/n$ and $N_a \simeq N_i \simeq n^d = N$. Number $N = n^d$ thus represents the overall and approximate number of both document nodes and tree items, while n represents the cardinality of each relation. The worst-case complexity of the associated binding is given by the complexity of the sort that binding *D2R* performs, and thus costs $\mathcal{O}(n^2)$. The initial construction of the tree widget requires N_e sort operations, and thus costs $C_{init} = N_e \times \mathcal{O}(n^2)$, *i.e.* $C_{init} = \mathcal{O}(N \times n)$. Mutations cost a linear time for all operations within the binding, *i.e.* $C_{update} = \mathcal{O}(n)$.

The ratio N between these two complexities well illustrates the interest in making active binding operations, rather than recomputing all the resulting tree widget content. The next section illustrates that our experimental results match these theoretical ones.

4.3 Experimental results

Our experimental measurements focus on two objectives: verifying that our theoretical complexities match the measured performance; comparing the cost of active operations with the cost of widget rendering. They have been performed on a PC running Windows XP on top of an Intel Pentium 1.8 GHz + 1 GB RAM and an ATI FireGL 128 MB. Figure 6 gives the performance of initial construction of the tree widget for the XML document defined in the previous section with depth 3 ($N = n^3$). Axis x represents number n, and axis y gives time performance in ms.

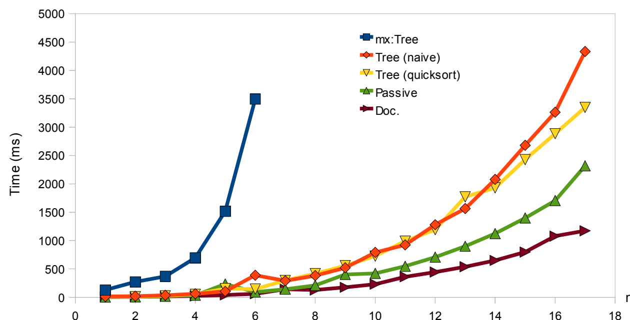

Figure 6. Construction performance

The five curves represent respectively: the XML document construction (curve "Doc.") that includes the creation of instances of classes *Document*, *Element* and *Attribute*, and the establishment of their relation; the "passive" data binding (curve "Passive") that represents the usual operations that can be used (only) for the initial construction; the tree construction (curve "Tree (naive)") that represents the binding uses to build the target instances of class *Tree* and *TreeItem* with active operations; the tree construction that uses a quicksort-based algorithm (curve "Tree (quicksort)") instead of a naive one (previous curve); finally, the tree widget rendering (curve "mx:Tree") performed by the Flex engine to display the graphical content of the widget. Curve "Doc" tends to $0,25 \times N$, thus illustrating that the document construction is linear. Curve "Tree (naive)" gives the performance of the bindings in the worst-case since it uses a naive implementation of the sort that costs $\mathcal{O}(n^2)$; it tends to $0,052 \times N \times n$, thus matching the expected theoretical result. Curve "Tree (quicksort)" tends to $0,18 \times N \times log_2\, n$; the expected logarithmic dimming effect appears for a significant value of n (here around 14). Curve "Passive" tends to $0,028 \times N \times n$, which is around half the curve "Tree (naive)": this illustrates that the translation cost is around 1, as expected (one intermediate order collection is here required). Finally, curve "mx:Tree" tends to $0,6 \times N \times n^2$, which probably results from the traversing required for computing location y of each tree-item i. This location may be recursively computed by summing heights h of the previous-sibling items of i: $y(i) = \sum_j h(i.sibling[j])$; in turn, computing the height of a node consists of summing the height of its children: $h(i) = \sum_j h(i.children[j])$. The overall layout includes N computation of y, so that the complexity tends to $N \times n^2$.

This experimentation illustrates that complexity of data bindings can often be neglected regarding the complexity of rendering complex graphics, such as a tree. Simpler widgets, such as a list widget, can however require complexity comparable to the one of the associated binding. Experiments performed on example 2 with a large list of contacts confirms this point: because the binding performs a sort, displaying the list can become faster than performing the binding, but for a large number of list items (around 10.000 in our experimentation).

Figure 7 gives the performance of the addition of an attribute; performance of its removal is very close to this curve, and is thus not shown in the figure. In both cases, updating the document (curve "Doc.") takes a constant and negligible time; data binding takes a linear time $t(ms) = 0,015 \times n$, as expected (curve "Tree"); the Flex tree widget does not have to compute new y locations: this rather consists of applying a vertical offset in the display buffer, which requires a constant time around 150 ms (curve "mx:Tree").

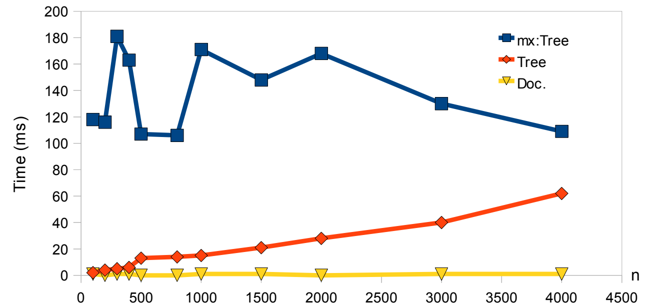

Figure 7. Addition performance

These performances illustrate that the time required to refresh a drawing is often bigger than the time for computing a binding. In the previous example, the update time for the Flex tree and the active operations become equal when n reaches $150/0,015 = 10.000$, which represents a really huge XML document (10^{12}nodes for a depth of 3).

5. RELATED WORKS

Constraint systems have been widely used for defining data bindings in the context of GUI. The Garnet toolbox proposes the use of "formula", expressed in Lisp, to bind an object value to other object values: the for-

mula is reevaluated whenever the values within the formula change, like in spreadsheets [14]. The Rendezvous architecture defines the concept of "active value": an active value is a variable used as en entry point to constraint definitions (also expressed in Lisp); when the value changes, the constraint is reevaluated to reflect the change [11]. Similarly, but within a C++ infrastructure, the Amulet environment allows the definition of constraint through "formula" that bind object values [15]. However, all these constraint-based systems are limited to the binding of object values: they do not address the binding of collections, and are thus limited to simple widgets such as text-field or combo-box.

The *JFace* toolkit defines interfaces *IObservableValue* and *IObservableCollection* that respectively allow the observation of values and collections [9]. These interfaces are implemented in classes that can be used for representing "bindable" data in both the model and the view. This approach defines the required foundation classes for data binding; however, it is a pure *programmatic* approach where bindings must be implemented rather than being declared. The *ObjectEditor* toolkit uses a more subtle approach that consists of extending the JavaBean syntax so that data bindings appear in a more declarative way [7]. Moreover, this toolkit extends the scope of data binding to action binding that allows user actions to be bound to "do", "undo" and "redo" methods. However, *ObjectEditor* offers a low expressiveness regarding data binding capabilities.

RIA toolkits, such as WPF [16], Flex [12], and JavaFX [10], have adopted a *declarative* approach to data binding. The aim of these approaches is to simplify the process of writing bindings: rather than being implemented, bindings are declared. These approaches work fine for simple bindings where the view does not differ much from the model; they however lose their simplicity as soon as the complexity of the model is increased, which often results in requiring some low level programming that contradicts the initial aim of the declarative approach. Section 6 focuses on proving these limitations.

Using rule-based incremental transformation languages solves the previous limitations of declarative data bindings. The incremental XSLT processor *incXSLT* allows the incremental execution of XSLT transformations [17]. However, XSLT concerns source XML documents and target text documents, which makes it difficult to use on various GUI platforms. Moreover, XSLT is a complex language that does not match well the required simplicity of data binding. Using a *mapping* language, such as Malan [5], combines the benefits of the simplicity of data bindings and the expressiveness of transformation languages. However, this higher level approach postpones the problem of executing the mappings: for example, active operations can be generated from a mapping so that the mapping can be executed on a target platform.

6. COMPARISON WITH BINDINGS OF RIA TOOLKITS

From our knowledge, recent RIA toolkits offer the best data binding mechanisms. This section compares our approach with three representative RIA toolkits: Flex [12], WPF[16], and JavaFX [10]. It illustrates why the three examples presented in this paper cannot be fully implemented by using their data binding capabilities.

6.1 Active Operation Capabilities

Table 3 summarizes the capabilities of active operations presented throughout this paper. The capabilities belong either to active collections, active objects, or both.

Category	Capability	Collection	Object
Math.	*union*	√	√
	intersection	√	√
	difference	√	√
	tuple	√	√
View	*select*	√	√
	sort	√	n/a
	multiple views	√	√
Transf.	*collect*	√	√
	path	√	√
	map	√	√
Bidir.	bidir. assignment	n/a	√
	+ conversion	n/a	√

Table 3. Active operation capabilities

Category "Math" includes the usual mathematical set operations, and the definition of tuples; tuples are useful in operations such as *select(f)* and *collect(f)* with more than one parameter in *f*. Category "View" (in the database sense) includes operations *select* and *sort*; it also defines the ability for an active collection or object to be treated more than once by an active operation (such as *select* and *sort*), thus allowing *multiple views* on a common model. Category "Transformation" concerns operations *collect*, *path* and *map*, that allow a progressive transformation from the source to the target. Finally, category "Bidir." defines the bidirectional assignment used to perform two-way bindings on active objects; this functionality is supplemented by a reversible conversion.

All these capabilities have been illustrated in the 3 examples of section 2. The next section discusses if and how these capabilities are present within the data binding mechanisms of the three RIA toolkits Flex, WPF and JavaFX.

6.2 Comparison with RIA Toolkits

Table 4 summarizes the data binding capabilities of the three RIA toolkits Flex, WPF and JavaFX.

As mentioned in section 4.1, instances of the Flex class *ArrayCollection* can be bound to collection widgets, and bindable properties defined within instances of class *Object* can be bound to simple widgets [12]. These two capabilities are equivalent to operation *map* on collections

	Flex		WPF		JavaFX	
	Coll.	Obj.	Coll.	Obj.	Coll.	Obj.
Math op.						
select	(√)		√			
sort	(√)	n/a	√	n/a		n/a
m. views			√	√	√	√
collect		√		√	√	√
path		√		√		√
map	√	√	√	√	√	√
assign.	n/a	√	n/a	√	n/a	√
+ conv.	n/a		n/a	√	n/a	

Table 4. binding capabilities of RIA toolkits

and on objects. Bidirectional assignment is possible through property binding, but without conversion. A path can be specified when binding two properties; however, paths cannot be defined on collections. It is also possible to define a "transformation function" within a property binding, which is analogous to operation *collect* for a single object. All the other capabilities are not available with Flex. Operations *select* and *sort* exist on class *ArrayCollection*, but they do not return new collections: they rather modify the source model directly; this does not separate well the model from the GUI. Moreover, these operations cannot use "active" parameters (for example, through tuples).

WPF offers richer binding capabilities than Flex [16]. Operations *select* and *sort* return new bindable collections, and thus allow the definition of multiple views; bidirectional conversion is also possible. However, math operations and transform operations *collect* and *path* remain undefined for collections.

JavaFX has adopted a radically different strategy for defining bindings: rather that proposing a bunch of binding artifacts, such as with Flex or WPF, JavaFX introduces new language *keywords* dedicated to the definition of bindings [10]. Consequently, JavaFX bindings are clearer than that of Flex or WPF since they are based on very few constructs, while Flex and WPF bindings use many different artifacts that often require low level *ad hoc* programming. However, as table 4 illustrates, important features are unavailable in JavaFX thus making it not as expressive as expected.

6.3 Implementing the Examples using RIA Toolkits

Regarding example 1, Flex data binding cannot be used to bind XML documents to the tree widget since this requires *union* operations. The example could be built by coding a specific data provider that implements Flex interface *ITreeDataDescriptor*. This practice is similar to the implementation of Swing models, and thus does not respect the initial philosophy of binding. Filtering contacts in example 2 is impossible with Flex binding: this requires the programming of an *ad hoc* code that recomputes the filtering whenever the user changes the filtering text. Moreover, sorting and filtering the contacts must be specified by altering the code of the model: the design of the GUI thus impacts the source model, which contradicts the fact that binding clearly separates the model and its GUIs. Finally, example 3 cannot be implemented by using Flex binding. This is due to the free-time slots objects that are not present in the source model, and thus cannot be bound to grid-items.

As with Flex, example 1 cannot be implemented using WPF binding due to the lack of operation *union*. Example 2 is easier to implement than with Flex due to WPF view capabilities; however, the active filtering of contacts again requires the development of *ad hoc* code. But most of all, implementing example 2 with WPF requires a bunch of data binding artifacts: "bidirectional converters" must be programmed by implementing interface *IValueConverter*; "data templates" must be defined with the XAML document[5] for implementing functions that define the text of the tree items; and "observable collection" must specify the filtering and sorting of contacts in a programmatic way. The developer thus needs to juggle with multiple artifacts, and with both programmatic and declarative paradigms. Example 3 cannot reap benefit from WPF binding and must be implemented from scratch.

Due to the lack of view capabilities, examples 1 and 2 cannot be implemented using JavaFX binding. As with Flex and WPF binding, example 3 is not feasible with JavaFX.

6.4 Discussion

Data binding of RIA toolkits are simple to use for simple bindings. Their binding mechanisms remain sufficient as long as the source models do not differ much from the target models. Moreover, these mechanisms allow binding to existing relational databases and/or XML documents directly, thus making them attractive in an enterprise context. In counter part, the offered binding capabilities become insufficient to address various and/or complex problems. These toolkits often require multiple artifact juggling that alter the initial simplicity of the declarative approach. Moreover, each RIA toolkit proposes its own data binding mechanism. Consequently, a binding written in WPF cannot always be translated to a Flex or JavaFX binding: they are all platform dependent and do not share a common model.

The use of active operations for specifying and implementing data binding removes such limitations. A single DSL is used for specifying the bindings, and translating them before being implemented; it is thus platform-independent. The resulting implementation performances are good for the initial construction of operation results, and very good for updating these results. In counter part, active operations have to be implemented through a dedicated compiler; however, RIA toolkits also require some hidden code generation

[5]XAML is the XML dialect used by WPF to specified user interfaces.

(for example, to implement paths between two bound properties). Active operations currently do not provide mechanisms for binding UI directly with databases or XML documents. However, such an issue is not specific to the domain of active operations: it concerns the serialization and deserialization of models on different data platforms.

7. CONCLUSION

This paper presents how to reap benefits from active operation expressiveness and easiness for specifying and implementing UI data bindings. Active operations offer the ability to reevaluate the result of the operations in real-time while the user interacts with the system. The proposed approach overcomes the limitations of data binding mechanisms provided by modern UI toolkits. These limitations are: GUI bindings are platform-dependent and lack at defining complex bindings. It is based on a single DSL, mainly inspired by OCL, that allows both the specification and the platform-independent implementation of data bindings. Active operations have been implemented on top of the Flex RIA toolkit; the resulting ActiveFlex API offers good performance for the initial construction of active operation results, and very good ones regarding their updates.

The next step of our work is to implement a compiler that will generate active implementations from any active operation specification. Such a compiler would generate the implementations on various GUI platforms. Since our approach offers a great expressiveness, we will use and evaluate it in the context of "Information Visualization" in order to confront its performance to large models.

8. REFERENCES

1. D. H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent, 2000.
2. O. Beaudoux. XML active transformation (eXAcT): Transforming documents within interactive systems. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 146–148. ACM, 2005.
3. O. Beaudoux and A. Blouin. Linking data and presentations: from mapping to active transformations. In *DocEng '10: Proceedings of the 2010 ACM symposium on Document engineering*, pages 107–110. ACM, 2010.
4. O. Beaudoux, A. Blouin, O. Barais, and J. M. Jezequel. Active operations on collections. In *MoDELS '10: Proceedings of the 13th ACM/IEEE International Conference on on Model Driven Engineering Languages and Systems (LNCS 6394)*, pages 91–105. Springer, 2010.
5. A. Blouin, O. Beaudoux, and S. Loiseau. Malan: A mapping language for the data manipulation. In *DocEng '08: Proceedings of the 2008 ACM symposium on Document engineering*, pages 66–75. ACM, 2008.
6. J. Coutaz. PAC, on object oriented model for dialog design. In *Interact'87*, 1987.
7. P. Dewan. Increasing the automation of a toolkit without reducing its abstraction and user-interface flexibility. In *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 47–56. ACM, 2010.
8. R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 2002.
9. Eclipse Foundation. JFace data binding. `http://wiki.eclipse.org/index.php/JFace_Data_Binding`.
10. R. Field. JavaFX language reference (chapter 7 - Data binding). `http://openjfx.java.sun.com/current-build/doc/reference/ch07s01.html`.
11. R. D. Hill. The Rendezvous constraint maintenance system. In *UIST'93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 225–234. ACM, 1993.
12. C. Kazoun and J. Lott. *Programming Flex 2*. O'Reilly, 2007.
13. G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in smalltalk80 system. *Journal of Object Oriented Programming*, 1:26–49, 1988.
14. B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, 1990.
15. B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: new models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.
16. C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly, 2005.
17. L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 474–485. ACM, 2002.
18. J. B. Warmer and A. G. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley.

Specifying and Running Rich Graphical Components with Loa

Olivier Beaudoux[1], Mickael Clavreul[1], Arnaud Blouin[2], Mengqiang Yang[1], Olivier Barais[2], Jean-Marc Jezequel [2]

[1]ESEO, TRAME Team
LUNAM University
Angers, France
[first-name].[last-name]@eseo.fr

[2]University of Rennes 1
IRISA, Triskell Team
Rennes, France
[last-name]@irisa.fr

ABSTRACT

Interactive system designs often require the use of rich graphical components whose capabilities go beyond the set of widgets provided by GUI toolkits. The implementation of such rich graphical components require a high programming effort that GUI toolkits do not alleviate. In this paper, we propose the Loa framework that allows both the specification of rich graphical components and their integration within running interactive applications. We illustrate the specification and integration with the Loa framework as part of a global process for the design of interactive systems.

ACM Classification Keywords

D.2.2 Software Engineering: Design Tools and Techniques—*Computer-aided software engineering (CASE), User interfaces*

General Terms

Algorithms, Design, Languages

Author Keywords

Graphical components ; Graphical User Interface (GUI) ; Domain Specific Language (DSL) ; Active Operations

INTRODUCTION

Building Rich Interactive Applications (RIA) often leads to the design of new graphical components. Recent interactive systems such as smart-phones and tablets well illustrate such a purpose. The design of complex graphical components that goes beyond the composition of existing widgets requires implementation effort and prevents reuse or capitalization. While recent GUI toolkits, such as GWT [26], WPF [22] and Flex [14], target the implementation of RIAs, the design of rich graphical components is still a time-consuming activity.

Providing the right methodologies, models and tools that facilitate the design and the implementation of such rich graphical components is thus an important challenge. Standard GUI toolkits are centered on a *low level* implementation of graphical components and are based on primitive drawing functions. This statement applies to proven toolkits such as Swing [9] and remains for recent standards such as HTML 5 [10]. This situation leads to interoperability and synchronization issues between the graphical design made by the designers and its implementation written by the programmers. Current RIA environments solve this issue by proposing tools that integrate (or link) both the design and the programming environments. For instance, FlexBuilder is both a designing and programming environment dedicated to Flex applications ; Expression Blend is a designer environment that produces XAML documents that can be directly used within VisualStudio, a programming environment.

Integrating design and programming environments provides interoperability and synchronization to some extent. But in this case the specification of new graphical components requires a significant implementation effort. Effort includes the definition of components, the implementation of the associated interactions and the related actions performed on the domain data, and their integration within the final application.

In this paper, we propose the Loa framework that allows both the specification of rich graphical components and their integration within running interactive applications. This framework homogenizes the process of extending and integrating previous works on data binding [3, 4], graphical templates [13, 27, 2], and interactors [19, 1, 6]. The paper focuses on the specification and integration with the Loa framework as part of a global *process* for the design of interactive systems.

The remainder the paper is structured as follows. Next section presents a global overview of the Loa framework that includes a DSL (Domain Specific Language), a development process, and a supporting tool implementation. Sections “Step 1” to “Step 6” detail each step of the process through the concrete example of building a planner application. We discuss this work and evaluate Loa with regard to existing GUI toolkits in Sections “Rela-

ted Works" and "Evaluation". Last section concludes this work and proposes perspectives.

OVERVIEW OF THE LOA FRAMEWORK

A Scala-based DSL

The Loa framework is based on the Scala language [20]. This choice is motivated by the following facts : 1) Scala is an OO language fully compatible with Java, thus allowing the reuse of the numerous Java API and tools, especially GUI ones ; 2) Scala embeds the imperative the functional and the object-oriented programming paradigms in a way that greatly facilitates the implementation of the framework ; 3) Scala is a self-extensible language that enables the construction of new DSLs[1]while reusing the infrastructure and tools (*i.e.*, IDEs and compilers) of the Scala language for these DSLs.

The Loa[2] DSL is thus built as an extension of the Scala language that defines the very language of the framework.

With this short introduction in mind, the Loa DSL allows designers to capture the problem domain of GUI engineering within a precise, concise and dedicated language. This paper focuses on the *usage* of the Loa framework supported by the Loa DSL in the process of building new graphical components for interactive systems. Details on the Loa DSL are thus out of the scope of the paper ; the DSL is rather explained through a concrete example.

The development process

The Loa framework splits the development process of interactive systems into six steps as illustrated in Figure 1. Using the usual concept of classes, application designers specify the *domain data* (step ①). Graphical designers sketch new graphical components using a graphical design tool such as Illustrator or Inkscape (step ②) that produces an XML document. Sketches are then formalized into *graphical templates* (step ③) that both define the graphical parts of the components in XML and their parameters. While steps ①, ②, and ③ require interaction between application and graphical designers, they can be executed in parallel. Application designers specify the *data binding* (step ④) using the data binding capabilities of active operations [3, 4]. Data binding consists in linking the domain data to the graphical templates. Application designers specify the *interactors* (step ⑤) using the data binding capabilities provided by the framework and an interaction model inspired from Malai [6]. The specifications that result from steps ①, ③, ④ and ⑤ are finally integrated together in a sole executable application. While the execution of the resulting application requires the whole set of specifications, each specification can be tested independently. For instance, a given graphical template can be tested while the specification of domain data or interactors are not yet provided.

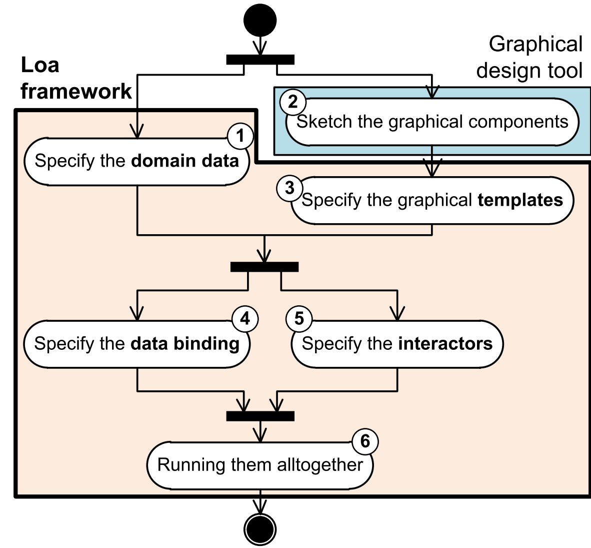

Figure 1. The development process

While the process is proposed for building new graphical components, the use of existing widgets provided by GUI toolkits fits in. In such a case, step ② relies on GUI design tools, such as Adobe FlexBuilder, Microsoft Expression Blend or Google WindowBuilder. Step ③ and ⑤ thus consists in reusing the Loa model of these widgets. Steps ①, ④ and ⑥ remain unchanged.

As one may note, the process focuses on the idea of *specification*. The ability to run a specification without requiring a hand-written implementation of the specification is an essential contribution that we summarize as :

"Specifying GUIs rather than implementing them"

However, the process is *not* a methodology for the development of interactive systems. It is rather a process that formalizes the use of the Loa framework ; it can be used jointly to well-known methodologies such as user's task analysis [21].

The implementation

The implementation of the Loa framework consists in an API that bridges the three main concepts of data bindings, of graphical templates, and of interactors with existing GUI toolkits.

The current *implementation* provides a bridge to *Swing widgets* as well as *Batik* for the definition of graphical templates based on the SVG standard. The case study that we present in the paper use the latter. The case study has been built using Inkscape for sketching the graphical templates, using Google WindowBuilder for designing the Swing UI in a WYSIWYG manner, and using the Eclipse workbench with Scala support for the edition and the compilation of the code written in Loa.

1. A DSL is a programming language that targets a specific problem. It contains the syntax and semantics of the language concepts at the same level of abstraction that the problem domain offers.

2. French acronym that stands for Language for Active Operations

STEP 1 : SPECIFY THE DOMAIN DATA

Figure 2 gives the class diagram that represents the domain data of the academic planning : *Planning* of a given year is composed of 0 to 52 *weeks* ; a teaching *Week* is composed of 0 to 5 *days* ; and each *Day* defines its *teachings*. A *Teaching* can span among multiple *TimeSlots* : the first element of relation *timeSlots* defines the starting time-slot, while the second element gives the ending time-slot. A *Teaching* also references a *topic*, *a teacher*, and *a room*.

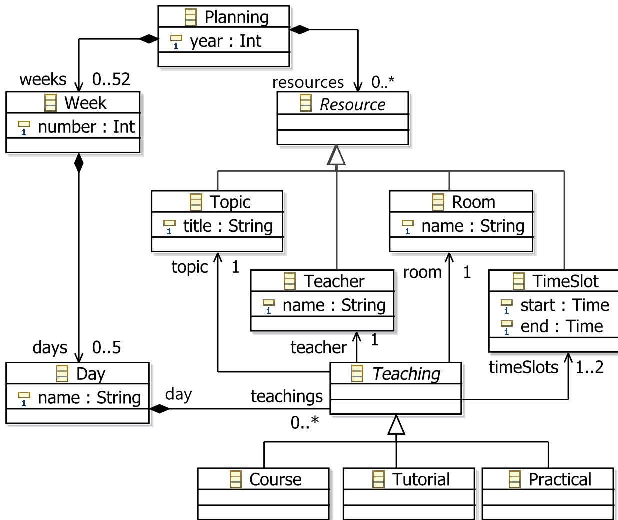

Figure 2. Class diagram of an academic planning

The Loa *data model* provides a textual representation of classes based on the use of six kinds of Loa *container*. This data model is used for specifying the domain data, the graphical templates, as well as the interactors. The following listing illustrates the specification of the class *Planning* from the domain data using the three Loa containers *One*, *OSet* and *Set* :

```
class Planning {
  val year = One(2012)
  val weeks = OSet[Week]()
  val resources = Set[Resource]()
}
```

Attribute *year* is represented as an integer boxed into a container with initial value *2012*. Relation *weeks* is represented as an initially empty ordered set of *Week* instances, and relation *resources* as a set of *Resource* instances.

	Cardinality	Uniqueness	Order
Opt	0..1		
One	1..1		
Bag	0..*		
Seq	0..*		√
Set	0..*	√	
OSet	0..*	√	√

Table 1. Loa container kinds

Table 1 lists the six kinds of container : the four last kinds are equivalent to the OCL collections [29] and are supplemented by the two kinds *Opt* and *One*. These six kinds of container differ by three properties : cardinality, uniqueness and order.

Container contents can be modified using assignment operators. Operator *:=* is used for *Opt* and *One* containers, and operators *+=* and *-=* are used for the four collection containers, as follows :

```
val p = Planning()
p.year := 2012
p.weeks += Week()
```

All the six kinds of container implement an observability mechanism that allows each change (addition, removal or update) to be captured. Observability is an important feature for graphical components, data bindings and interactors.

Finally, methods written in Scala define the application logic that is integrated in the classes of the system.

STEP 2 : SKETCH THE GRAPHICAL COMPONENTS

Figure 3 gives a screen-shot of the *Academic Planning Application* that displays a weekly view of the academic planning, as introduced in the previous section ; each teaching is presented to users through a *stamp* graphical component. Menu *File* allows loading and saving XML documents that contain planning data related to a specific academic year. Menu *Edit* allows editing planning resources (*e.g.,* rooms, topics), and allows duplicating or clearing the week contents. The combo-box at the top allows selecting a specific week within a year (from 1 to 52). The canvas allows direct manipulation of the selected week. Undo and redo buttons allow undoing and redoing past actions. Finally, the tab *Tree* displays the tree of the graphical scene for debugging purpose.

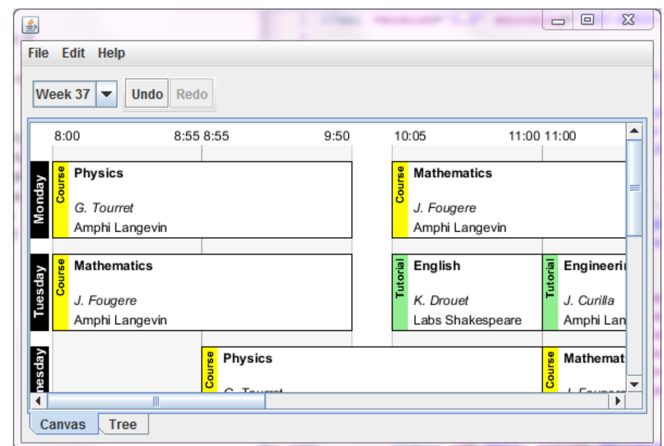

Figure 3. The Academic Planning Application

Based on the application requirements, the graphical designer sketches the necessary graphical components exported by the tool as XML representations. The following code illustrates an excerpt of a *stamp* component

that uses the SVG format for its XML representation :

```
<g transform="translate(10,10)">
  <!-- background -->
  <rect width="170" height="70" fill="white"/>
  <!-- type bar -->
  <rect width="15" height="70" fill="yellow"/>
  <text x="-5" y="11" transform="rotate(-90)" ...>
    Course
  </text>
  <g transform="translate(20,15)">
    <!-- labels -->
    <text font-weight="bold">Mathematics</text>
    <text y="20" font-style="italic">J. Fougere</text>
    <text y="50">Amphi Langevin</text>
  </g>
  <!-- border -->
  <rect width="170" height="70" fill="none"
    stroke="black"/>
</g>
```

The *stamp* is an SVG group composed of : a white background (line 3) ; a type bar representing the *stamp* type with a text (lines 6 to 8) and a yellow background (line 5) ; a group of labels displaying the event description (lines 11 to 13) ; and a black stroke defining the stamp border (line 16 and 17).

Since SVG does not support the parametrization of symbols, the code of the *stamp* graphical component includes hard-coded value. Step ③ addresses such parametrization issues.

STEP 3 : SPECIFY THE GRAPHICAL TEMPLATES

Loa formalizes sketches of graphical components (step ②) as the combination of a Loa data model and templates. In the following subsections, we present the specification of such templates with Loa and we explain how templates are transformed into an executable form.

Specification of templates

Figure 4 shows the class diagram which contains the *template classes* that formalize the sketched graphical components used by the planning application. The root template class *Calendar* represents the weekly presentation of a planning.

The presentation of a calendar is a grid that is composed of *dayLabels* and *timeAreas*. Its contents is displayed by *stamps*, each *Stamp* including a *typeBar* (*i.e.*, the type of the displayed event) and *labels*.

A template class is specified using both the Loa data model and the specification of its *template*. Template class *Stamp* specifies attributes *x*, *y*, *width* and *height* and relations *typeBar* and *dayLabels* as follows :

```
class Stamp extends Fragment {
  val x = One(0)    val width = One(300)
  val y = One(0)    val height = One(100)
  val typeBar = One(TypeBar())
  val dayLabels = OSet[Label]()

  template {
    <g transform="translate($x,$y)">
      <rect width="$width" height="$height"
         fill ="white"/>
      $typeBar
      <g transform="translate(20,15)">
        $dayLabels
      </g>
      <rect width="$width" height="$height"
         fill ="none" stroke="black"/>
    </g>
  }
}
```

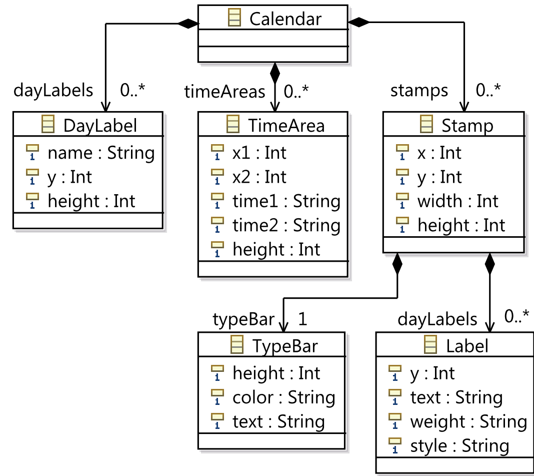

Figure 4. Class diagram of the planning templates

Loa introduces the keyword *template* defined by class *Fragment* (term fragment is explained in the next subsection). Parametrization of a template is carried out by the concept of *anchor* : an anchor is bound to a container that defines the anchor contents and the anchor location within the template through symbol *$*. In the *Stamp* example, anchors *$x*, *$y*, *$width* and *$height* (lines 8 and 9) receive the contents of their associated containers *x*, *y*, *width* and *height* (lines 2 and 3) ; anchor *$typeBar* (line 11) receives the associated *TypeBar* instance contained in *typeBar* (line 4) ; anchor *$dayLabels* (line 13) will receive multiple *DayLabel* instances subsequently added into the ordered set *dayLabels* (line 5).

Instantiation of templates

Instantiation of a Loa template into a fragment consists of creating a DOM fragment with an initial content defined by the template itself, and binding the anchors of the templates within the DOM fragment. The interlacing between fragments and anchors allows the incremental construction of the final graphical components. The interlacing approach has been borrowed from eXAcT [2]. Since the mechanism is quite complex, we provide an overview of the core principles. Figure 5 illustrates the

instantiation of the template *Stamp* into a fragment. For the sake of clarity, Figure 5 only contains the *$width*, *$height* and *$dayLabels* anchors.

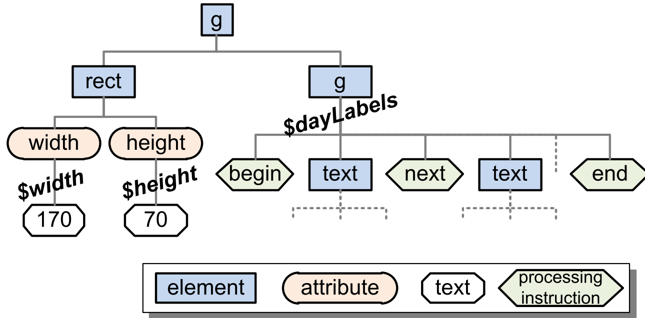

Figure 5. A subset of an instance of fragment *Stamp*

Instantiation of the template *Stamp* consists of creating a DOM fragment containing a root <*g*> element and all its nested nodes until an anchor specification is reached. For instance, when attribute *width* of element <*rect*> is reached, an anchor is created. This anchor observes the value of its associated container *width* defined in the enclosing fragment class. Similarly, when child <g> element is reached, an anchor is created. The content of this anchor is delimited by two processing instruction <*?begin?*> and <*?end?*> that reflects the container *dayLabels* : adding a new instance of class *DayLabel* to the relation *dayLabels* triggers the corresponding template (containing element <*text*>). The corresponding template is thus instantiated and the resulting child fragment is inserted in between the delimiters. Separation of subsequent *DayLabel* fragments is represented by the processing instruction <*?next?*>.

STEP 4 : SPECIFY THE DATA BINDING

Besides the fact that Loa containers are observable, the Loa data model allows binding two containers so that their contents remains the same. Loa introduces the assignment operator *::=* so that expression $a ::= b$ means that container a receives the same contents as container b. When used concurrently with operations available in the Loa DSL (*e.g. apply*), the assignment operator makes possible to bind multiple containers, as illustrated by the following example :

```
1 val a = Seq[Int]()
2 val b ::= a.apply{e => e.toString()}
3 a += 123
```

This example binds the sequence of integers a to the sequence of strings b. Line 2 means that b contains the string representation of integers contained in a. Since a is initially empty, b is also initially empty. When the sequence a is modified on line 3, the sequence b is updated by the execution of operation *apply* accordingly, such that b finally contains “123”. The implementation of the function *apply* is based on the observation of sequence a and on the execution of the anonymous function $e =>$ *e.toString()* in an appropriate manner. More details on the implementation of complex functions such as *select* and *sort*, is detailed in [3].

Binding containers allows the specification and execution of complex data bindings that bind the domain data to graphical components [4]. The instantiation of templates is thus driven by the changes performed on the domain data. As an illustration, the following function *P2C* is a Loa specification that represents the data binding between a planning p loaded from menu *File*, a combo-box *ws* for selecting the current week, and a calendar c that gives a weekly presentation of p to users (see Figure 3) :

```
1 def P2C(p: Planning, ws: ComboBox, c: Calendar) = {
2   c.dayLabels ::= Seq(0 to 4).apply(DN2DL)
3   c.timeAreas ::= p.timeSlots.map(TS2TA)
4   ws.items ::= p.weeks.map(W2I)
5   c.stamps ::= ws.selectedItem.rmap(W2I)
6     .days.teachings.map(T2S)
7 }
```

From a static sequence of day numbers (0 to 4), the binding function *DN2DL* creates the fragments of anchor *dayLabels* (line 2). Line 3 defines the contents of anchor *timeAreas* using the binding function *TS2TA* on *timeSlots*. Line 4 populates the *ws* combo-box with the planning weeks by calling the mapping function *W2I* : the selected item of *ws* drives the creation of *Stamp* fragments within anchor *stamps* retrieves the selected week (function *rmap*, line 5) and applies the mapping function *T2S* to all teachings of this week.

The contents of mapping functions *DN2DL*, *TS2TA*, *W2I* and *T2S* follows the same constructs as *P2C*. This example shows that we use equivalent mapping constructs for both existing widgets (*e.g.*, Swing combobox), or specific graphical components (*e.g.*, an SVG graphics).

STEP 5 : SPECIFY THE INTERACTORS

The Loa interactors are based on Malai [6]. An interactor transforms an interaction into an action on the target object. Within Loa, the target object can be either a domain data or a fragment.

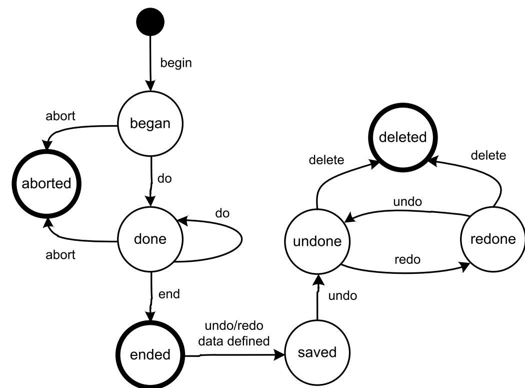

Figure 6. Action life-cycle

Figure 6 illustrates the *life-cycle* of the *action* of any Loa interactor as *a generic* state-machine. For a mouse-based interactor, each transition corresponds to a mouse event as follows : *begin* = mouse button pressed, *do* = mouse dragged, *abort* = escape key pressed, *end* = mouse button released. If the interactor defines undo/redo data, these data are *saved* into the undo-redo stack. An interactor, such as a key or button, can request an *undo* or a *redo*, as illustrated at the end of this section. Finally, when the undo-redo stack reaches its maximal capacity, the undo/redo data are *deleted*.

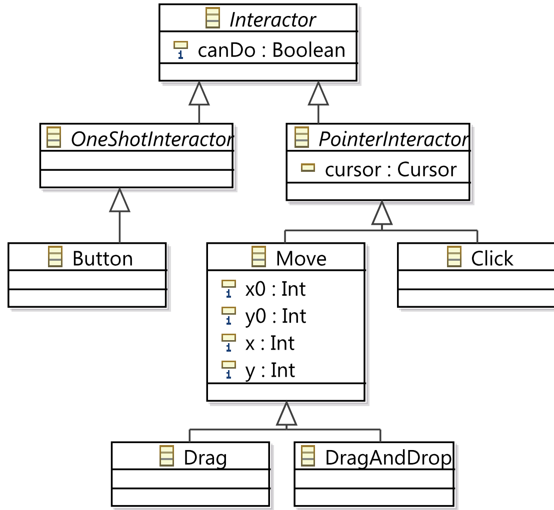

Figure 7. Loa interactors

Figure 7 shows a subset of the Loa interactor classes. Base class *Interactor* defines the attribute *canDo* that indicates if the action associated with this interaction can be started or not. Class *OneShotInteractor* represents interactors with a state *began* that is immediately followed by a state *end* (*e.g.*, *Button* and *MenuItem*). Conversely, class *PointerInteractor* represents interactors with a complex action life-cycle (*e.g.*, *Move* and *Click*). Class *Move* defines properties (*x0*,*y0*) and (*x*,*y*) that represent respectively the starting mouse location and the current mouse location.

Class *Interactor* defines a Scala *when block* that allows designers to specify actions for each possible transition of the interactor, as follows :

```
when {
  case Begin =>
  case End =>
  case Abort =>
  case Undo =>
  case Redo =>
}
```

Each *case* corresponds to a transition and triggering a transition invokes the corresponding *case block*. The specification of actions for states depends on the kind of interactor. *OneShotInteractor* interactors only rely on the definition of either case block *Begin* or *End*. *PointerInteractor* interactors rely on case blocks *Begin* and *End*, and an optional case block *Abort* if the action needs to be aborted. Case blocks *Undo* and *Redo* provide undo/redo capabilities for *OneShotInteractor* and *PointerInteractor* interactors.

The following *Hand* interactor specifies how the user can move a stamp to another location :

```
1  object Hand extends Drag {
2    when {
3      case Begin(s: Stamp) =>
4        cursor := Cursor.Move
5        calendar.stamps += s // put s on the top
6        val t = s.rmap(T2S)
7        t.day ::= y.apply(y2Day)
8        t.start ::= x.apply(x2TimeSlot)
9        t.end ::= (s.width + s.x − x).apply(x2TimeSlot)
10       undoData += (t, t.day, t.start , t.end)
11     case End(s: Stamp) =>
12       cursor := None
13       val t = s.rmap(T2S)
14       redoData += (t, t.day, t.start , t.end)
15     case Undo(t: Teaching, d: Day,
16         s: TimeSlot, e: TimeSlot) =>
17       t.day := d
18       t.start := s
19       t.end := e
20   }
21 }
```

Stamp *s* is given as a parameter in case blocks *Begin* and *End*, and corresponds to the stamp picked at location (*x*,*y*). Line 6 retrieves the teaching *t* that has been previously mapped to stamp *s* with the mapping function *T2S*, thanks to the reverse mapping operation *rmap*. Lines 7 to 9 define the new location of the picked teaching *t* by using layout functions that convert location (*x*,*y*) to *Day* and *TimeSlot* instances. The three properties *day*, *start* and *end* are bounded to the interactor location using operator *: :=*, thus meaning that these three locations will be updated while the drag interaction continues. Case block *Begin* ends with saving undo data. Case block *End* terminates the action by resetting the cursor to its default value (line 12), then by saving the undo/redo data into the Loa undo stack (line 13 and 14). Based on the undo stack, case block *Undo* restores the saved state of teaching *t* when necessary (line 17 to 19).

We voluntary omit case block *Redo*, since the undo/redo actions are based on the same data. Similarly, case block *Abort(s : Stamp)* has been omitted for simplification purpose.

Button *undoButton* (see figure 3)[3] specifies its action in a simpler manner, as follows :

```
undoButton.when { case End => UndoStack.undo() }
```

3. Button *redoButton* is defined similarly.

	Data model ①		Data binding ④			Gr. templates ②③		Interaction model ⑤		
	obs. prop.	obs. coll.	prop.	coll.	lang.	simple	nesting	events	action	interactor
Swing	H	L	L	L				H	L	
JFace	H	H	H	L				H	L	
ObjectEditor	H	H	H	M	L			H	H	
Garnet	H		H		L			H		H
Amulet	H		H			M		H	H	H
Malai			H	M	M			**H**	**H**	**H**
JavaFX 2.0	H	H	H	L		L		H		
Flex	H	H	H	L	L	M		H	L	
JavaFX 1.3	H	H	H	L	M	M	L	H		
WPF	H	H	H	M	L	H		H	L	
Ext GWT	H	H	H	M	L	H		H	L	
Hayaku			L	L	L	H	H	H		
incXSLT	M	M	H	H	M	H	H			
sXBL	M	M	H	M	L	H	H	H		
eXAcT	M	M	H	H		**H**	**H**	H		
Definitive pr.	H	H	H	L	M					
Active op.	**H**	**H**	**H**	**H**	**H**					
Loa	H	H	H	H	H	H	H	H	H	H

Table 2. Synthetic View of Related Works

Global object *UndoStack* represents the undo/redo stack used by interactors to store undo/redo data. Button *undoButton* binds its *canDo* attribute as follows :

```
undoButton.canDo ::= UndoStack.canUndo
```

The *UndoStack* object defines attribute *canUndo* that specifies whether the undo/redo stack contains undo data or not. This attribute is bound to the *canDo* attribute of the undo button, thus resulting in disabling or enabling the button depending on the stack state.

STEP 6 : RUNNING THEM ALL TOGETHER

The following Scala code defines the main entry point of the application :

```
1  object Application extends Frame {
2    def main(args: Array[String]) {
3      val calendar = Calendar()
4      val svg = SVGScene(calendar, 1400, 600)
5      val canvas = SVGCanvas(svg)
6      val planning = Planning()
7      planning.load("Planning 2011-12.xml")
8      P2C(planning, weekSelector, calendar)
9      canvas.interactors += Hand
10     canvasScrollPane.setViewportView(canvas)
11     treeScrollPane.setViewportView(DOMTreeView(svg))
12     setVisible (true)
13   }
14 }
```

The main window of the application is represented by a Java class *Frame* created with Google WindowBuilder. Since the integration of Scala code with Java is smooth, we may define a singleton class *Application* that extends *Frame*. Fragment *calendar* (line 3) is the root fragment of scene *svg* (line 4), which is rendered by the object *canvas* (lines 5 and 10) and displayed as a DOM tree view (line 11). Instance *planning* is loaded from an XML file (lines 6 and 7), and then bound to instance *calendar* with the application of function *P2C* (line 8). Combo-box *weekSelector* allows a user to select a week to edit (see Figure 3), and is thus involved in *P2C*. The creation of an *Application* ends (lines 9 to 12) with the association of an interactor *Hand* with *canvas*, followed by the display of the *canvas* and its components.

RELATED WORKS AND DISCUSSION

Table 2 gives a synthetic comparison of related works for each of the four domains related to steps ① to ⑤ : the *data model*, the *data binding*, the *graphical templates* and the *interaction model*. Each domain is then evaluated against two or three criteria indicating whether : *i)* the data model allows the definition of *observable properties* and/or *observable collections* ; *ii)* the data bindings can be defined for *properties*, *collections* using a dedicated *language* or not ; *iii)* the graphical templates allow the definition of *simple* components (*e.g.,* a *TimeArea*, a *TypeBar* - see Figure 4) and/or *nesting* components (*e.g.,* a *Calendar*, a *Stamp* - see Figure 4) ; *iv)* the interaction model provides an *event*-based interaction model, an *action* model, and/or an *interactor*-based model. Values for criteria are : high (H), medium (M), low (L), and none (empty cell). Concepts of the related works that we reuse within Loa are in bold.

The table splits the related works in the following five categories (from top to bottom) : Java-based toolkits, interactor-based toolkits, RIA toolkits, template-based systems, and data binding languages. The following sections discuss these categories with regard to the four proposed criteria. Readers must be aware that the values presented in the table are all subject to discussion according to the subjectivity of the selected criteria. However, we think they well represent the overall tendency

of each category.

Java-based toolkits

Swing [9] is based on the MVC pattern rather than on the data binding concept. Consequently, the definition of bindings requires a significant programming effort for implementing interfaces. Creating a new component is a time-consuming task that requires a full implementation from scratch. As for all GUI toolkits, the interaction model listens to predefined events. The concept of action is minimalist and does not integrate *undo/redo* features. JFace [12] explicitly uses data binding between observable properties, and allows binding observable collections with some predefined widgets. ObjectEditor [8] allows the definition of data bindings on properties and on collections, and enhances the action model with *undo/redo* capabilities. Since ObjectEditor is based on extending the JavaBean syntax, it might be considered as a language. We observe that the category "Java-based toolkits" both lacks mechanisms for the definition of *graphical templates* and of *interaction model*, and globally lacks dedicated languages for *data binding*. These issues are answered by the dedicated toolkits presented in the next sections.

Interactor-based toolkits

Garnet [17] and Amulet [18] explicitly introduce the concept of interactor to facilitate the use of predefined interactions. Amulet provides a high level action model with *undo/redo/repeat* features, as well as the ability to define simple graphical components easily. Garnet and Amulet allow the definition of constraints for binding properties with no support for collections. Malai [6] formalizes the instrumental interaction [1] into a well polished conceptual framework. Malai allows binding collections but does not support complex computation on collections. Malai interaction model has been reused and adapted to fit in the Loa data model. We observe that while "Interactor-based toolkits" support a high level interaction model with several limitations about data binding, *graphical templates* are out of their scope.

RIA Toolkits

RIA toolkits, such as JavaFX 2.0 [30], Flex [14], JavaFX 1.3 [24], WPF [22] and Ext GWT [26], provide better data binding capabilities than the two previous categories. Data bindings are often specified as string values within XML files representing the interface (*e.g.*, MXML for Flex or XAML for WPF). An alternative is to use a programming language, such as JavaFX script for JavaFX 1.3. WPF and Ext GWT use their own template model to represent simple graphical components (*e.g.*, items of a list), but this model does not allow the definition of new nesting components. Java FX 2.0 uses a hand-written representation of the scene that describes the contents of the template. JavaFX 1.3 provides a bind statement for loops on collection that allows the definition of nesting components. As a summary, RIA toolkits support a large range of *data binding* capabilities but propose poor *interaction models* and do not support *nesting graphical templates*.

Template-based systems

As far as we know, Hayaku [23] is the only toolkit that specifically tackles the definition of post-WIMP graphical components. Using abstract representations, Hayaku ends up with good expressiveness and good overall performances. However, Hayaku does not specifically focuses on data binding except for linking the template to its data. Hayaku interaction model provides the mandatory picking capabilities. Within the selected tools that are not specific to GUI engineering, incXSLT [27], sXBL [28] and eXAcT [2] all work with a data model based on XML that we consider as a limitation here. incXSLT is an incremental XSLT processor that allows the incremental transformation of an XML document with the limitation that it does not cover the whole XSLT 1.0 language. We can use incXSLT to define graphical templates as standard *<template name>* XSLT elements. The SVG's XML Binding Language (sXBL) supports the parametrization of SVG symbols natively. ; However, it offers limited data binding capabilities. eXAcT is an incremental/active Java-based transformation processor that offers good data binding capabilities. Loa uses the concept of fragment and anchor as defined by eXAcT while leveraging the inherent complexity of eXAcT transformations.

"Template-based systems" are relevant approaches for the definition of *graphical templates*. However they provide poor support for the integration of an *interaction model* and of a *data model*.

Data binding languages

Definitive principles and notations [5] proposes a novel approach for the programmatic evaluation of functions. For instance, instead of calling $y=f(x)$ each time x changes, definitive notations represent $y=f(x)$ as a constraint f that binds x and y. This works well for property bindings but there are limitations for collection bindings. Notations are defined within a dedicated language, not close to Object Oriented Programming (OOP) usage. Active operations are based on a similar principle [3] but allows the definition of data bindings using a classic OOP approach and standard operations on collections [29]. [4] demonstrates that active operations allow the definition of complex data bindings that cannot be expressed with RIA toolkit without *ad hoc* coding. Loa is based on the concept of active operations implemented as a Scala internal DSL. "Data binding languages" propose languages that support simple and complex *data bindings*. *Graphical templates* and *interaction models* are however not in the scope of these techniques.

Conclusion

This preliminary evaluation shows that GUI toolkits (*i.e.*, Java-based, interactor-based and RIA) have little to no support for the definition of nesting templates and interactor models, and limited support for data binding.

The integration of relevant approaches that are not dedicated to GUI, such as XSLT templates and active operations, provides new functionalities that overcome these limitations. Loa has been designed for this very purpose : the successful integration of the various concerns of interactive systems in a unique DSL. Next section provides evaluation and discussion on the Loa language.

EVALUATION

In this section, we propose to evaluate Loa and its supporting process against the Cognitive Dimensions of Notation proposed by Green [11]. We discuss the dimensions of abstraction, hidden complexity, closeness of mappings, viscosity and progressive evaluation and support our claims with van Deursen's observations [25] about the benefits of using DSLs.

Abstraction / Hidden Complexity

Providing a good level of abstraction is essential for the designer to be productive in the design of new graphical components. Since DSLs are concise in general [16], we consider that Loa proposes a representative set of constructs with precise semantics that allows designers to focus on each specific concern of the domain rather than on the complexity of the implementation or of the development artifacts.

The use of an homogeneous representation (*i.e.,* the Loa DSL) across five of the six steps of the development process for new graphical components is a very valuable asset towards reliability and maintainability [7, 15] of the final application. The Loa DSL is also provided with a concrete syntax as a textual notation. The textual notation allows designers to set the links between the development artifacts (*i.e.*, data model, data binding, graphical templates and interaction model) that was not cover by Hayaku [23].

Closeness of Mappings

The second step of the Loa development process allows application designers and graphical designers to agree on a design that is close to the final product. From an initial sketch of the graphical component, the production of the component may be realized in parallel : the graphical designer works towards a final version of the visuals while the application designers takes care of the implementation and integration of the component into the final application.

Viscosity

Resistance to changes is a challenging activity in any process of development. While this paper does not focus on the activity of maintaining consistence as changes occurs, every step of the process is based on an homogeneous representation (*i.e.,* Loa). Homogeneity is one solution to get confidence in the consistency of the various development artifacts instead of manipulating multiple representations.

Since Loa is an internal Scala DSL, we benefit from the existing Scala and Java tooling to detect the side effects of changes at design time. As a matter of fact, the manual propagation of changes is limited to the synchronization of the data model with the data binding, and of the data model with the interaction model.

Progressive Evaluation

The development process that supports Loa covers the various specifications (*i.e.,* from the graphical design to its implementation) required to build a fully functional graphical component. However, testing a new graphical component does not require for all steps to be completed. For instance, the design of a new graphical component can be tested against its static properties while the behavior is not yet implemented. This allows designers to design components incrementally.

CONCLUSION

This paper presents the Loa framework dedicated to the engineering of interactive systems. Switching from implementation to specification, the Loa framework focuses on the definition of specifications that are executable within a final application. The framework integrates previous works on data binding, graphical templates and interactors. The Loa framework provides : 1) a dedicated DSL based on the Scala language ; 2) a six-step development process that provides guidance in the use of the framework ; and 3) an implementation that currently bridges the Swing toolkit and allows the definition of graphical components in SVG.

Perspectives of the framework target the implementation of bridges to other Java toolkits, such as JGraph and SWT/JFace, as well as bridges to other platforms such as .NET and Web-oriented technologies. Long term perspectives will target the integration of other concerns specific to UI design such as groupware and constraint management.

REFERENCES

1. M. Beaudouin-Lafon. Instrumental interaction : An interaction model for designing post-wimp interfaces. In *Proc. of CHI'00*, volume 2, pages 446–453. ACM Press, 2000.
2. O. Beaudoux. XML active transformation (eXAcT) : Transforming documents within interactive systems. In *Proc. of DocEng'05*, pages 146–148. ACM, 2005.
3. O. Beaudoux, A. Blouin, O. Barais, and J. M. Jezequel. Active operations on collections. In *Proc. of MoDELS '10*, pages 91–105. Springer, 2010.
4. O. Beaudoux, A. Blouin, O. Barais, and J.-M. Jézéquel. Specifying and implementing ui data bindings with active operations. In *Proc. of EICS'11*, pages 127–136. ACM, 2011.
5. W. Beynon. Definitive principles for interactive graphics. *NATO ASI Series F*, 40(3) :1083–1097, 1988.

6. A. Blouin and O. Beaudoux. Improving modularity and usability of interactive systems with malai. In *Proc. of EICS'10*, pages 115–124. ACM, 2010.
7. A. V. Deursen and P. Klint. Little languages : little maintenance ? *Journal of Software Maintenance : Research and Practice*, 10(2) :75–92, 1998.
8. P. Dewan. Increasing the automation of a toolkit without reducing its abstraction and user-interface flexibility. In *Proc. of EICS '10*, pages 47–56. ACM, 2010.
9. R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 2002.
10. S. Fulton and J. Fulton. *HTML5 Canvas*. O'Reilly Media, 2011.
11. T. R. G. Green. Cognitive dimensions of notations. In *People and Computers V*, pages 443–460. Cambridge University Press, 1989.
12. R. Harris and R. Warner. The definitive guide to swt and jface, 2004.
13. M. Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Wrox, 2008.
14. C. Kazoun and J. Lott. *Programming Flex 2*. O'Reilly, 2007.
15. R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proc. of ICSE '96*, pages 542–552. IEEE Computer Society, 1996.
16. D. A. Ladd and J. C. Ramming. Two application languages in software production. In *Proc. of USENIX 1994*, pages 1–9. USENIX Association, 1994.
17. B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet : comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11) :71–85, 1990.
18. B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment : new models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6) :347–365, 1997.
19. B. A. Myers. A new model for handling input. *ACM Transaction on Information Systems*, 8(3) :289–320, 1990.
20. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2010.
21. J. Redish and J. T. Hackos. *User and Task Analysis for Interface Design*. John Wiley & Sons, 1998.
22. C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly, 2005.
23. B. Tissoires and S. Conversy. Hayaku : designing and optimizing finely tuned and portable interactive graphics with a graphical compiler. In *Proc. of EICS'11*, pages 117–126. ACM, 2011.
24. K. Topley. *JavaFX Developer's Guide*. Addison-Wesley, 2010.
25. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, June 2000.
26. D. Vaughan. *Ext GWT 2.0*. Packt Publishing, 2010.
27. L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *Proc. of WWW'02*, pages 474–485. ACM, 2002.
28. W3C. Svg's xml binding language (sxbl). Technical report, W3C, 2005.
29. J. B. Warmer and A. G. Kleppe. *The object constraint language : getting your models ready for MDA*. Addison-Wesley.
30. J. Weaver, W. Gao, S. Chin, and D. Iverson. *Pro JavaFX 2 Platform*. Apress, 2011.