Baron, M. and Girard, P. Construction interactive d'application à partir du noyau fonctionnel, in *Proc. Ergonomie et informatique avancées (Ergo-IHM'2000)* (Biarritz, France, 3-6 octobre, 2000), ESTIA, pp. 85-93.

Construction interactive d'application à partir du noyau fonctionnel

Patrick Girard, Mickaël Baron

Laboratoire d'Informatique Scientifique et Industrielle, ENSMA,
1 rue Clément Ader, 86961 Futuroscope Chasseneuil
girard@ensma.fr

RESUME

Les outils de conception d'interface utilisateur utilisent des approches diverses. Les constructeurs d'interface partent de la présentation, alors que générateurs et systèmes basés sur modèle partent de la description de différents modèles pour construire l'application. Dans tous les cas, il est difficile de s'affranchir de phases de compilation qui ralentissent le processus de conception.

L'approche présentée dans cette contribution vise à faire collaborer ces deux points de vue de la construction d'interface. La démarche est validée sur une étude de cas, la construction d'un jeu de TicTacToe à partir de son noyau fonctionnel. Une première génération automatique, suivie d'une phase alternant construction et test interactif permettent une construction réellement interactive de l'application.

MOTS CLES : Outil de construction, programmation visuelle, programmation sur exemple.

INTRODUCTION

Nombreux sont aujourd'hui les outils qui permettent d'aider un concepteur à réaliser une application graphique interactive. Parmi les outils les plus évolués, deux grandes tendances peuvent être opposées, selon le point de départ à partir duquel ils cherchent à construire l'application.

Les constructeurs d'interfaces privilégient l'aspect « présentation » et permettent de réaliser le noyau fonctionnel au fur et à mesure de la construction de l'interface. L'association de ces environnements à des langages interprétés permet de raccourcir le cycle de développement, avec l'inconvénient majeur de rendre plus difficile la sécurisation de l'application finale.

À l'inverse, générateurs automatiques d'applications et systèmes basés sur modèles tirent parti de spécifications, notamment du noyau fonctionnel, le plus souvent enrichies de descriptions supplémentaires, pour construire ou générer l'application interactive. L'apprentissage de langages spécifiques et la nécessaire traduction des modèles utilisés en code exécutable complexifient le processus de création, bien que la seconde tâche n'incombe pas au concepteur.

Pourtant, lorsqu'un noyau fonctionnel peut être développé de façon indépendante de toute perspective d'interaction homme-machine, ses spécifications, pour peu qu'elles soient formalisées et complètes, constituent un vrai modèle, qui peut être pris comme point de départ pour la construction de l'application. Rapprocher les deux points de vue consisterait, en partant ce ces spécifications, à créer un outil capable de construire interactivement une application directement exécutable. L'utilisation des techniques de programmation visuelle et de programmation sur exemple serait alors de nature à réduire le cycle de développement.

L'objet de notre contribution est de montrer la faisabilité d'une telle approche sur une étude de cas, et d'identifier les problèmes posés par sa généralisation.

Dans un premier temps, nous décrivons notre cahier des charges, ainsi que les principaux choix que nous avons effectués. Ensuite, nous décrivons l'outil que nous avons développé, en présentant tout d'abord son principe général, puis en détaillant les phases de génération et de construction interactive. La troisième section permet de situer notre travail en regard de la littérature. Enfin, nous traçons quelques unes des nombreuses perspectives qu'ouvre ce travail.

CAHIER DES CHARGES

En partant d'un noyau fonctionnel complet, notre objectif est de construire interactivement l'application finale en utilisant directement ce noyau fonctionnel, sans phase de compilation additionnelle. Nous sommes partis des principes de développement modulaire semi-formel exposés dans [12] et du travail de JD Fekete [3] sur les services minimums pour l'interaction homme-machine, et avons identifié les éléments supplémentaires nécessaires à la réalisation de notre but.

L'application

Nous avons choisi comme sujet de l'étude de cas le jeu « TicTacToe » qui consiste pour deux adversaires jouant à tour de rôle à aligner trois symboles identiques sur une grille de neuf cases. Nous nous sommes restreints dans un premier temps à un jeu interactif entre deux joueurs, le système n'étant pas en mesure de remplacer l'un des joueurs.

Une interface possible pour l'application finale pourrait être, par exemple, celle présentée dans la figure 1. L'espace de jeu est constitué d'une grille 3x3, chaque case pouvant contenir un symbole correspondant à l'un ou l'autre joueur (ici un carré et un rond). L'état de l'application (le joueur devant jouer, le gagnant lorsque le jeu est terminé) est représenté par des messages textuels.

Les trois services préconisés par JD Fekete se traduisent dans cette application par les éléments suivants :

- **l'annulation d'opération** consiste à permettre à un utilisateur d'annuler la dernière interaction. Ici, il s'agit d'annuler l'action de sélection d'une case de la grille.
- **la prévention d'erreur** consiste à éviter par exemple de jouer dans une case déjà utilisée par un joueur. Une implémentation possible est par exemple « d'attacher » le symbole du joueur actif au curseur et d'en modifier l'apparence selon l'état de la case « survolée ».
- **la notification** consiste à permettre au noyau fonctionnel de prévenir l'application d'un événement. Dans notre cas, la fin du jeu constitue un exemple typique de notification.

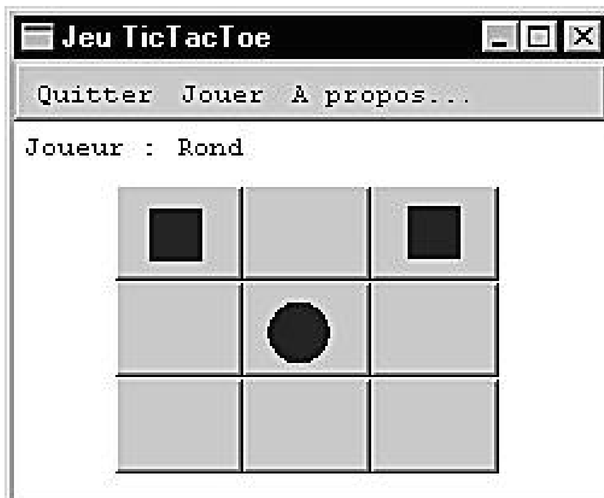

Figure 1 : Exemple d'interface du TicTacToe

Choix techniques

La réalisation de notre objectif nécessite de disposer d'une boîte à outil adaptée à des besoins très forts en interactivité. En effet, après la phase de génération s'appuyant sur le noyau fonctionnel, il faut être capable de créer de nouveaux objets d'interaction supportant par exemple la manipulation directe, tout en permettant l'activation directe des actions du noyau fonctionnel. Notre choix s'est porté sur la boîte à outils AMULET, qui nous semblait réunir toutes les conditions requises.

Amulet [10] signifie Automatic Manufacture of Usable and Learnable Editors and Toolkits. Il a été développé par le « User Interface Software Group » au « Human Computer Interaction Institute », à l'université de Carnegie Mellon. Ce véritable environnement de développement d'application interactive a été réalisé pour concevoir facilement une interface hautement interactive, graphique, pour les systèmes Unix, Win32 et Macintosh. C'est une boite à outils évoluée qui permet la manipulation directe. Elle utilise de nouveaux modèles d'objets, de contraintes, d'animations, de manipulation directe et d'annulations. Le système d'objet est un modèle de prototype – instance dans lequel il n'y a pas de distinction entre classes et instances, ou entre méthodes et données. Un système de contraintes s'applique à n'importe quelle valeur d'un objet. L'animation peut être rattachée à des objets existants par une simple ligne de code. L'utilisateur réalise ses actions à travers des objets appelés interacteurs (« *interactor* »). Nous illustrons ci-dessous la dynamicité d'Amulet, et montrons la simplicité de l'ajout de comportements de manipulation directe.

Amulet permet une gestion dynamique des objets. En effet tout est représenté sous forme d'objets possédant des « slots » dynamiques. Chaque « slot » porte un nom et contient une valeur typée ou non. Il est possible de changer dynamiquement la valeur de tout slot, d'en changer le type, ou encore d'ajouter ou de supprimer des « slots » à un objet. Un « slot » peut ainsi contenir une valeur réelle à un instant et une chaîne de caractères à un autre instant. Le programme de la figure 2 montre un exemple de code Amulet.

```
1 Am_Object wind = Am_Window.Create();
2 wind.Set(Am_LEFT, 20);
3 Wind.Set(Am_TOP, 20);
4 Wind.Set(Am_WIDTH, 20);
5 Wind.Set(Am_HEIGHT, 20);
6 Wind.Add(chaine, « Bonjour »);
7 Am_Screen.Add_Part(wind);
```

Figure 2 : Exemple de « slot » dans Amulet

En ligne 1, l'objet *wind* est déclaré comme un objet *Am_Window*, une fenêtre. De la ligne 2 à 5, les slots de coordonnées et de taille sont modifiés. La ligne 6 crée un nouveau slot appelé *chaîne* dont le type est une chaîne de caractères. Enfin la ligne 7, permet d'afficher la fenêtre à l'écran.

Un deuxième intérêt majeur d'Amulet, pour nos besoins, est sa gestion de la manipulation directe. Il est ainsi possible d'associer à chaque objet des opérations, correspondant aux différentes actions typiques de la manipulation directe. Ces actions sont représentées sous forme d'objets appelés interacteurs. Par exemple on peut

associer à un rectangle l'opération *Am_Move_Grow*, qui correspond à une action de glisser/lâcher via la souris. Cet objet peut alors réagir à ce type d'action de la part de l'utilisateur. Le programme de la figure 3 montre un exemple d'implémentation d'un tel procédé.

```
1 Rect = Am_Rectangle.Create();
2 rm = Am_Move_Grow_Interactor.Create();
3 rect.Add_Part(rm);
```

Figure 3 : Exemple d'interacteur

rect et *rm* ont été définis comme des objets ; *rect* est un objet rectangle et *rm* un objet interacteur. Le programme indique que l'utilisateur a la possibilité de déplacer *rect*.

Le noyau fonctionnel

Compte tenu de notre choix de boîte à outil, nous avons été contraint, pour notre interface avec le noyau fonctionnel, d'utiliser le langage C^{++}.

Nous avons donc conçu une classe définissant complètement le jeu. Nous n'avons pas défini de sous-objet (les cases par exemple), afin de ne pas compliquer le noyau, même si une telle décomposition eût certainement été préférable.

En dehors du constructeur de la classe et d'une fonction d'initialisation (`initialise` qui permet de réinitialiser le jeu), nous avons défini les services suivants :

- la gestion automatique du joueur. Les joueurs sont représentés par les entiers 1 ou 2 : à l'initialisation, le jeu autorise toujours le joueur 1 à jouer, puis gère l'alternance jusqu'à la fin du jeu. Le nombre de coups joué est également géré par le noyau fonctionnel. Les fonctions qui gèrent cet aspect sont `etat_jeton()` et `nombre_coups()`
- le jeu lui-même. Il consiste à « jouer » dans une case donnée. La fonction `joue(i,j)` permet, en fonction du joueur actif, de modifier correctement le noyau fonctionnel.
- plusieurs fonctions permettent de rendre compte de l'état (forcément privé) du noyau fonctionnel. Il s'agit des fonctions `gagne()`, `fin()`, `joueur()`, `type_jeton(i,j)`.
- l'annulation : nous avons implémenté une annulation systématique des actions sur le noyau fonctionnel, sans limitation du nombre de retours autre que celle du jeu lui-même (on peut au maximum annuler neuf coups…).

La figure 4 montre l'ensemble du noyau fonctionnel avec les spécifications semi-formelles utilisées pour le décrire. Le modèle utilisé est à pré- (*nécessite*) et post-conditions (*entraîne*).

```
1   class tictactoe {
2     public :
3           tictactoe() ;                        // Constructeur de la classe TicTacToe
4           bool initialise() ;                  // Remise à zéro du TicTacToe
5           int nombre_coups() ;                 // Nombre de coups joués
6                 // valeur initiale = 0
7           bool gagne() ;                       // Un joueur a gagné !
8                 // valeur initiale = false
9           bool fin() ;                         // Partie finie
10                // valeur initiale = false
11          bool joueur() ;                      // Joueur 1 gagnant ?
12                // nécessite :   gagne() == true, fin() == false
13          int etat_jeton(int i, int j) ;       // Le joueur qui doit jouer
14                // nécessite :   i >= 1 && i <= 3 , j >= 1 && j <= 3
15                // valeur initiale = 1
16          int type_jeton(int i, int j);        // Renvoie le type de jeton en i et j
17                // nécessite :   i >= 1 && i <= 3 , j >= 1 && j <= 3
18                // valeur initiale = 0
19          bool joue(int i, int j);             // Joue en case i et j
20                // nécessite :   i >= 1 && i <= 3 , j >= 1 && j <= 3, type_jeton(i,j) == 0
21                // nécessite :   fin() == false, gagne == false
22                // entraîne :    type_jeton(i,j) = etat_jeton()
23                //               nombre_coup()++, changement de joueur
24          bool etat_precedent() ;              // Annule le dernier coup
25                // nécessite :   nombre_coup() > 0
26    private …
```

Figure 4 : Noyau fonctionnel du TicTacToe

DESCRIPTION DE L'APPLICATION

L'application que nous avons réalisée comprend deux modules aux rôles radicalement différents :

- Le premier module, que nous appellerons par la suite « **générateur** », extrait les informations du noyau fonctionnel (cf. figure 4) et génère un fichier C^{++}, destiné à réaliser l'interfaçage du noyau fonctionnel avec l'environnement interactif de construction. Cette phase est entièrement automatique. S'appuyant sur un fichier forcément correct du point de vue C^{++}, les erreurs potentielles se limitent aux description supplémentaires que nous avons ajoutées.
- Le deuxième module, appelé « **builder** », est créé en compilant le noyau fonctionnel avec le fichier d'interfaçage issu du module **générateur**. C'est un éditeur d'interface qui permettra de réaliser l'application interactive finale. L'utilisateur de ce **builder** est alors capable de tester le noyau fonctionnel en utilisant une présentation par défaut de ce dernier, mais aussi et surtout d'alterner phase de création et de test de l'application interactive finale.

La figure 5 montre une représentation schématique de l'application.

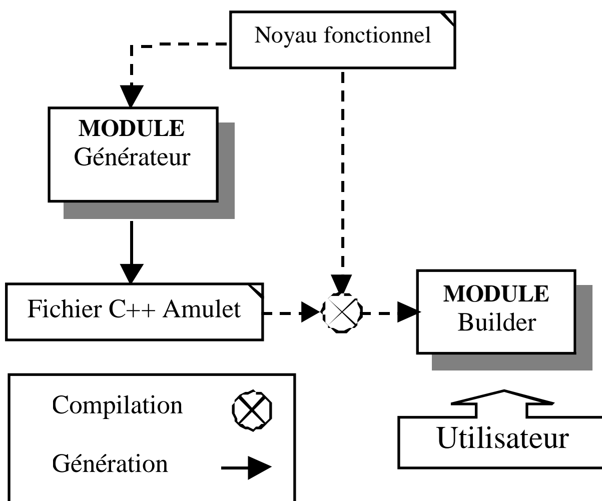

Figure 5 : Schéma général de l'application

Le module « Générateur »

Ce module a pour but de générer une interface interactive standard pour le noyau fonctionnel. Pour une classe donnée, il s'agit de fournir un moyen d'interagir avec chacune des fonctions publiques, et de représenter au mieux l'état du noyau fonctionnel.

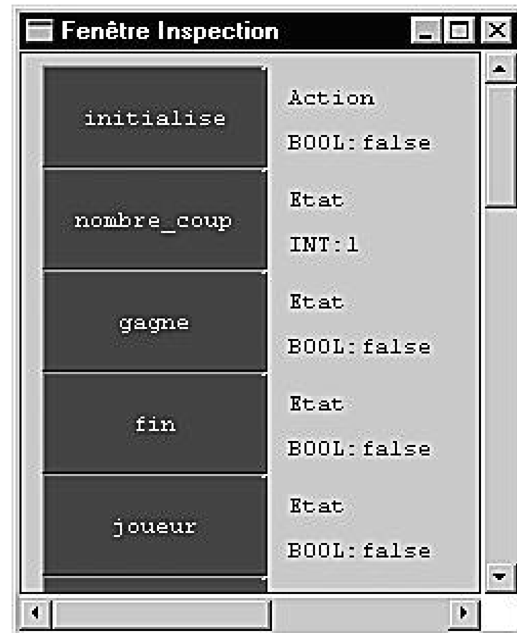

Figure 6 : Fenêtre d'inspection

Chaque fonction est associée à un bouton d'Amulet, qui permettra de l'activer interactivement. La représentation que nous avons choisie se fait dans une fenêtre que nous avons appelée fenêtre d'inspection, et qui est visible sur la figure 6 ; cette dernière montre une colonne de boutons dont le nom correspond à chaque fonction, et visualise à droite des boutons le type de résultat de retour de chaque fonction. Lorsque les fonctions n'ont pas de paramètre, la valeur courante de la fonction est également affichée. Pour faciliter l'identification du rôle des fonctions, nous avons caractérisé deux types de fonctions, qui sont également indiqués sur la fenêtre, les types « action » et « état ».

Les fonctions **actions** sont les fonctions qui modifient l'état du noyau fonctionnel. Par exemple, elles seront appelées par le contrôleur de dialogue au moment où l'utilisateur provoquera une action sur l'application finale. Une fonction action n'a pas pour vocation de renseigner l'état du noyau fonctionnel, donc la valeur de retour devra être obligatoirement de type booléen. Cette valeur indiquera à l'application graphique si l'exécution de la fonction s'est bien déroulée. Par convention une valeur *true* est une bonne exécution et *false* une mauvaise exécution de la fonction. La fonction `bool joue(int i, int j)`, qui place un jeton à l'emplacement i et j, est une fonction action.

Pour connaître l'état du noyau fonctionnel, l'application finale interrogera les fonctions **états**. L'information est retournée par la valeur de retour. Les fonctions états seront évaluées par l'application finale à chaque exécution d'une fonction action. La fonction `bool`

`fin()` est une fonction état qui indique si la partie en cours est finie.

Cette distinction entre fonctions actions et fonctions états n'est pas fondamentale à ce niveau d'analyse, mais nous montrerons dans les ouvertures de ce travail qu'elle nous permettra d'effectuer des vérifications intéressantes sur l'application finale.

Traduction des besoins spécifiques aux IHM

Nous reprenons ici les trois besoins spécifiques aux IHM identifiés par JD Fekete. Les choix effectués, en particulier avec la boîte à outils et le langage, ont imposé certaines solutions uniquement partielles.

La *fonction d'annulation* doit être implémentée dans le noyau fonctionnel, comme nous l'avons fait ; cela n'est cependant pas suffisant. En effet, pour permettre de conserver la propriété d'honnêteté de l'interface, il convient d'interroger et de visualiser chaque fonction d'état du noyau fonctionnel à chaque appel de la fonction d'annulation. Il serait plus efficace de pouvoir connaître les seuls éléments modifiés par une action, pour n'interroger que les fonctions concernant ces éléments. Cette solution imposerait une description supplémentaire dans le noyau fonctionnel.

Pour la *fonction de notification*, le problème est similaire. Comment détecter la fin du jeu ? l'interrogation de toutes les fonctions d'état après chaque action permet de contourner le problème. Il est clair que cette solution est viable dans notre étude de cas, mais ne peut être généralisée.

Pour la *prévention d'erreur*, la solution que nous proposons est plus générale. Parmi les informations disponibles dans les spécifications du noyau fonctionnel, les pré-conditions sont particulièrement intéressantes. Elles permettent d'implémenter directement la prévention d'erreurs, en fournissant au contrôleur de dialogue la possibilité d'interdire des actions si les conditions exprimées ne sont pas remplies. Ces pré-conditions sont des expressions booléennes C^{++}, qui peuvent donc être directement incluses dans le code du contrôleur de dialogue de l'application finale. Ceci permet de garantir que les actions de l'utilisateur ne pourront déclencher d'erreur dans le noyau fonctionnel.

À cette étape, une application interactive permettant d'activer les fonctions qui sont accessibles en fonction de l'état du noyau fonctionnel a été générée automatiquement. Cependant seules les fonctions sans paramètres sont activables de façon aussi simple qu'en appuyant sur un bouton. Lorsqu'il y a des paramètres, il faut pouvoir passer ces derniers à la fonction appelée. Pour cela, le générateur construit, pour chaque fonction possédant des paramètres, une boîte de dialogue permettant d'évaluer la fonction en lui donnant interactivement des paramètres. C'est ce que montre la figure 7. Lorsqu'une condition est présente sur les paramètres, la boîte de dialogue la vérifie préalablement à l'évaluation de l'action. Dans l'exemple, les paramètres doivent être compris entre 1 et 3.

Figure 7 : « Évalueur » d'actions

La génération de l'application interactive « standard » à partir de la description du noyau fonctionnel se fait donc de façon totalement automatique : pour extraire les données du noyau fonctionnel, le **générateur** utilise les outils Lex & Yacc. Pour chacun des deux programmes nous avons écrit une spécification qui regroupe un ensemble de formes auxquelles le fichier du noyau fonctionnel doit correspondre.

Le **générateur** récupère les informations de la spécification du noyau fonctionnel. Il commence par extraire les informations liées à la classe du noyau, et par la suite l'ensemble des fonctions. Pour ces dernières les données à récupérer sont la condition et la caractéristiques de la fonction. Le générateur étiquette la famille de la fonction et précise l'intervalle des paramètres. Ensuite il récupère le nom de la fonction, le type de la valeur de retour et éventuellement le ou les paramètres.

À ce stade, nous disposons d'un outil permettant de tester le noyau fonctionnel, et de vérifier s'il fonctionne correctement. Un mécanisme d'enregistrement des actions de l'utilisateur est en cours de réalisation pour permettre de créer interactivement des tests qui pourront ensuite être ré-exécutés après modification du noyau fonctionnel. La gestion de tests de non-régression s'en trouvera ainsi facilitée.

Baron, M. and Girard, P. Construction interactive d'application à partir du noyau fonctionnel, in *Proc. Ergonomie et informatique avancées (Ergo-IHM'2000)* (Biarritz, France, 3-6 octobre, 2000), ESTIA, pp. 85-93.

Le module « Builder »
Le rôle du **builder** est d'offrir au concepteur un outil pour générer facilement son application interactive à partir du code généré par le module **générateur**. Comme cette phase de génération a fourni un module permettant de tester l'application standard, il sera facile de garder cette exécution en parallèle, et d'éviter ainsi la phase de compilation (ou d'interprétation) des outils traditionnels.

Plutôt que de partir de la présentation réalisée automatiquement pour chercher à l'améliorer, nous avons choisi de fournir un espace de conception indépendant de la fenêtre d'inspection. Ceci permet de conserver une visualisation de l'état du modèle en cours de conception.

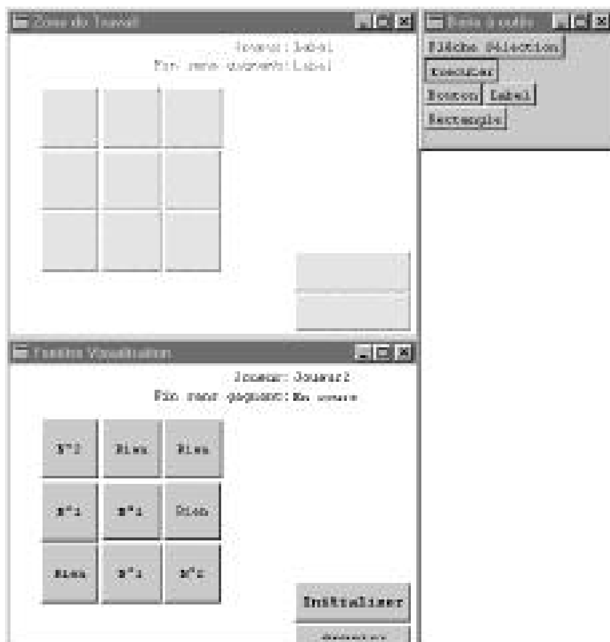

Figure 8 : Zone de travail et palette d'outils

Le **builder** est donc composé de 4 fenêtres, en plus de la barre de menu principale :

- La fenêtre d'inspection déjà décrite, qui demeure toujours active;
- Une zone de travail permettant de construire l'interface utilisateur en ajoutant des objets Amulet ;
- Une palette d'outils, associée à la zone de travail, permet de créer les objets d'interaction ; la figure 8 montre la zone de travail et une partie de la palette d'outils
- Une fenêtre de visualisation, qui représente l'application finale en cours de construction. Le passage de la phase de construction à la phase de test se fait simplement en cliquant dans la zone de travail (mode construction) ou dans l'une ou l'autre des fenêtres de visualisation ou d'inspection (phase de test), qui sont toutes deux connectées au même noyau fonctionnel. Un retour visuel permet au concepteur de connaître la fenêtre active, et donc de déduire la phase (conception ou test) active.

Mécanismes de « programmation »
La « programmation » de l'interface consiste, dans les termes d'Amulet, à affecter les fonctions du noyau fonctionnel aux « slots » des objets de l'interface. Deux mécanismes ont été développés.

Le premier consiste à ouvrir deux boîtes de dialogue en cliquant sur l'objet (clic droit). La première boîte de dialogue appelée « Interactions » définit l'interaction que l'utilisateur va réaliser sur l'objet. Sur chacune de ces interactions le concepteur choisira parmi les slots de l'objet, et parmi les fonctions du noyau fonctionnel, les associations nécessaires. Puis la deuxième boîte de dialogue appelé « Etats » définit la visualisation du résultat des actions, qui peut dépendre du résultat de l'appel d'une fonction état. Un exemple de la boîte de dialogue Etats est fourni à la figure 9.

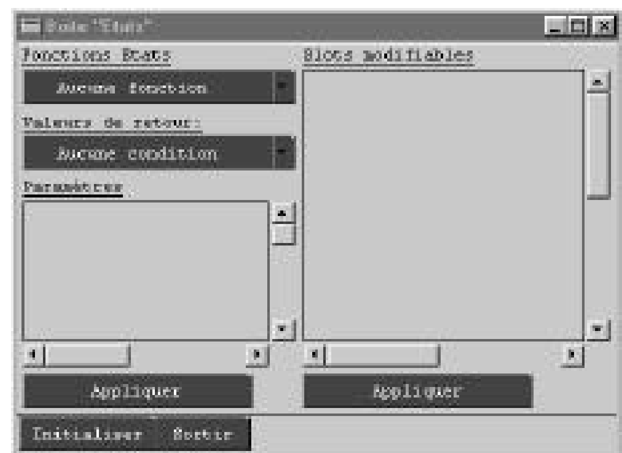

Figure 9 : Boîte de dialogue de « Etats »

Le deuxième mécanisme consiste à exploiter les possibilité de manipulation directe pour définir l'interaction. En mode création, le système permet de manipuler les objets d'interaction en démontrant l'interaction (utilisation de la programmation sur exemple). À chaque activation de slot prédéfini, le système demande à l'utilisateur ce qu'il souhaite réaliser. Un changement de visualisation ou le déclenchement d'une action peuvent ainsi être associés au slot en question.

AUTRES APPROCHES
Comme nous l'avons mentionné en introduction, nombreux sont les outils permettant de construire des applications graphiques interactives. Au-delà des boîtes à outils (généralement de bas-niveau) et des squelettes

d'application (de moins en moins utilisés), les SGIU ou Systèmes de Gestion d'Interface Utilisateur (UIMS en anglais, pour User Interface Management Systems) et les Systèmes Basés sur Modèles (MBS en anglais, pour Model-Based Systems) constituent les outils les plus en vogue aujourd'hui. Après avoir succinctement présenté quelques caractéristiques de ces systèmes en regard de notre approche, nous donnerons une autre voie, celle des environnements de programmation sur exemple (PbD en anglais, pour Programming by Demonstration), qui peuvent également répondre à certains de nos objectifs.

Constructeurs d'interfaces

Les constructeurs d'interface sont des outils permettant de définir l'aspect graphique de l'interface. Ils produisent un squelette d'application auquel il faut ajouter les appels à l'application. Employant généralement la manipulation directe, ils sont d'utilisation aisée, et permettent d'obtenir très rapidement des prototypes. Cependant, ils ne permettent pas de définir la dynamique du dialogue, qui doit être codée dans l'application. Ils n'autorisent pas non plus la visualisation des éléments de l'application, à moins d'utiliser une boîte à outils pour les coder dans l'application [7]. Ils permettent ainsi de dessiner aisément les différents écrans et boîtes de dialogue d'une application, mais pas de définir la succession de ces objets à l'écran, qui doit donc être codée par un développeur ayant connaissance de l'environnement.

Ce type de SGIU est bien au point, et maintenant intégré dans des produits commerciaux. Par exemple Visual Basic©, Visual C++© de Microsoft Corporation® ou Delphi© de Borland®, disposent de constructeurs d'interface directement intégrés à leurs environnements de programmation.

Les systèmes universitaires en cours de développement disposent de capacités plus étendues. Myers a ainsi poussé les techniques d'inférence à un point élevé dans ce domaine avec Peridot [8], puis GARNET [9] et GILT [4]. Ce dernier est ainsi à même de déduire du placement *approximatif* d'un objet graphique son aspect ainsi que sa position finale.

Aucun de ces systèmes ne permet cependant d'alterner les phases de construction et d'exécution comme le permet notre application. Au mieux, pour les langages interprétés comme le Basic, peut-on lancer une exécution alors que l'interface n'est pas complète, mais les phases d'exécution et de construction sont distinctes, et tout retour à l'environnement de construction entraîne la perte du contexte d'exécution.

D'autre part, la phase de programmation nécessite généralement l'ajout de petites portions dans divers endroits du code, par exemple en remplissant des squelettes de fonction. Par exemple, la prise en compte d'un bouton en Visual C++© impose l'écriture de code dans la fonction activée par un clic sur le bouton ; si l'on veut en plus que le bouton soit activé et désactivé selon le résultat d'une fonction du noyau fonctionnel, il faut intervenir au niveau du constructeur, puis remplir une nouvelle portion de code. Notre approche permet de centraliser l'ensemble de ces opérations à partir du bouton lui-même.

A notre niveau la programmation est facilitée par le mécanisme mis en place, ce qui conduit à réduire considérablement le cycle de développement. Il n'y a qu'à choisir le type d'interaction à placer sur l'objet et y associer une fonction du noyau fonctionnel simplement par manipulation visuelle. Par contre la plupart des outils de la famille SGUI, nécessitent un codage de l'application pour chaque événement de l'objet.

Des générateurs d'interfaces aux MBS

À l'opposé des constructeurs d'interfaces, on trouve l'approche par génération, qu'elle consiste à générer des esquisses d'interfaces susceptibles de modifications (UofA*UIMS, [13], ou MIKE [11]). ou à générer des interfaces définitives à partir de spécifications plus ou moins complexes (ITS [16], PDGen [2]).

Dans ce dernier cas, par exemple, PDGen génère un programme complet possédant une interface graphique pour éditer des données scientifiques à partir d'un fichier. Se basant sur une analyse du code C++, et plus particulièrement de la définition des classes, PDGen génère une interface permettant d'accéder directement aux données, de naviguer parmi elles, et de les modifier. Le seul but du programme étant cette accès aux données, il n'est ensuite possible que de sauvegarder les données modifiées dans un nouveau fichier. La principale différence avec notre approche est que la sémantique de l'application est figée. PDGen ne traite pas les fonctions membres des classes, et ne sait opérer que des actions limitées sur les données.

Pour vaincre cette difficulté, et étendre les possibilités des générateurs, la tentation est grande de toujours ajouter des éléments aux langages de spécification qui sont utilisés pour la première génération.

C'est ainsi que, depuis les années 1980, les langages de spécification des UIMS ont considérablement évolué, prenant en compte de plus en plus d'éléments, que ce soit du dialogue ou de la présentation, et permettant de générer des interfaces de plus en plus riches, de plus en plus complexes.

Les systèmes actuels utilisent par exemple un modèle des tâches de l'utilisateur, un modèle des données

manipulées par le système, un modèle de l'utilisateur, un modèle de la présentation… L'expression « Model Based System » fait référence à un ensemble d'outils de construction d'interface utilisant de tels modèles pour fournir une aide au développement de l'interface d'une application. Les maîtres mots sont *modèle* pour *générer* et *vérifier* le dialogue et l'interface d'une application [15].

Quoique séduisantes, ces approches tardent encore à trouver leurs utilisateurs, de par la complexité de leur utilisation, leur manque de complétude et la spécificité des outils généralement associés. Plutôt que de rechercher de nouveaux formalismes susceptibles de permettre une génération complète, nous préférons utiliser l'interface naturel des programmeurs, qui nous semble suffisamment riche en soi.

L'approche sur exemple (PbD)

Des travaux relativement récents dans le domaine de la programmation sur exemple sont à rapprocher de notre démarche. Il s'agit par exemple des systèmes KidSim/Cocoa [1, 14] ou Gamut [5, 6].

Ces systèmes fournissent des environnements de programmation sur exemple qui permettent de construire de petites applications interactives (des jeux dans ces deux cas), en démontrant un exemple du résultat attendu. Le système essaie de déduire des actions du « programmeur » une généralisation permettant de construire un programme.

Outre le fait que ces systèmes sont limités par leurs règles de déduction, ils souffrent d'un problème d'échelle, car il n'est pas possible ici de construire des noyaux fonctionnels indépendants : la réalisation sur exemple d'une application est une tâche qui devient très ardue à mesure que sa taille augmente. Ce reproche peut également être fait à notre solution, bien que nous ayons choisi une approche exempte de toute règle de déduction.

CONCLUSION ET PERSPECTIVES

Le travail que nous présentons ici n'en est qu'à sa première phase. Nous n'avons expérimenté la faisabilité de l'approche que sur un exemple limité.

Les données visualisées sont toutes de type élémentaire, et ne font pas intervenir d'espace de visualisation propre. Une étude approfondie des types de visualisation possibles en relation avec les widgets usuels est ici nécessaire. De même l'extension du noyau fonctionnel à des objets structurés, et la multiplication du nombre de classes induira des problèmes nouveaux. Quels modèles d'architecture d'application peuvent être utilisés ? Une approche objet semble naturelle, mais est-elle indispensable ?

Sur le plan de l'interface avec le noyau fonctionnel, il nous semble très intéressant d'étudier la possibilité de partir d'une spécification complètement formelle pour réaliser notre première génération. Ceci permettrait de concevoir un système qui garantirait que l'utilisateur ne viole pas les règles de l'application.

Un autre point digne d'intérêt est le contrôle de l'utilisation interactive du noyau fonctionnel : est-ce que l'interface utilise toutes les primitives du noyau fonctionnel ? Est-ce que tous les éléments définissant l'état du noyau fonctionnel interviennent dans la visualisation du système (en termes d'IHM, l'application est-elle honnête) ? La séparation fonction action / fonction état que nous avons décrite plus haut devient utile ici.

Nous avons vu comment l'utilisation des pré-conditions permettait de satisfaire au besoin de prévention d'erreur. les autres besoins fondamentaux décrits par JF Fekete peuvent-ils être pleinement satisfaits ? En existe-t-il d'autres ?

Enfin, l'alternance des phases de construction et de test qui est proposée ici mérite d'être validée par des utilisateurs pour vérifier son utilisabilité.

BIBLIOGRAPHIE

1. Cypher, A. et Smith, D.C. KidSim: End User Programming of Simulations. In *Proceedings of Human Factors in Computing Systems (CHI'95)* (7-11 May, Denver, Colorado), ACM/SIGCHI, 1995, pp. 27-36.
2. Engelson, V., Fritzson, D. et Fritzson, P. Automatic generation of user interfaces from data structure specifications and object-oriented application models. In *Proceedings of ECOOP96* (8-12 July 1996, Linz Austria), Springer Verlag, 1996, pp. 114-141.
3. Fekete, J.-D. Trois besoins pour les systèmes interactifs (à vérifier). In *Proceedings of Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'96)* (16-18 Septembre, Grenoble), Cépaduès, 1996, pp. 45-50.
4. Hashimoto, O. et Myers, B.A. Graphical Styles for Building User Interfaces by Demonstration. In *Proceedings of ACM Symposium on User Interface Software and Technology (UIST'92)* (15-18 November, Monterey, California), ACM/SIGCHI, 1992, pp. 117-124.
5. McDaniel, R.G. *Building Whole Applications Using Only Programming-by-Demonstration*. PhD Thesis : Carnegie Mellon University, 1999.

6. McDaniel, R.G. et Myers, B.A. Getting More Out Of Programming by Demonstration. In *Proceedings of Human Factors in Computing Systems (CHI'99)* (15-20 May, Pittsburg), ACM/SIGCHI, 1999, pp. 442-449.
7. Meinadier, J.P. *L'interface Utilisateur : pour une informatique plus conviviale*. Dunod Informatique, Paris, 1991.
8. Myers, B.A. *Creating User Interface by Demonstration*. Academic Press, 1988.
9. Myers, B.A., Giuse, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A. et Marchal, P. GARNET: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*. 23, 11 (1990), pp. 71-85.
10. Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A. et Doane, P. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering,*. 23, 6 (1997), pp. 347-365.
11. Olsen, R. Mike : the Menu Interaction Kontrol Environment. *ACM Transaction on Graphics*. 5, 3 (1986), pp. 318-344.
12. Pierra. *Les bases de la programmation et du Génie Logiciel*. Dunod informatique, Paris, 1991.
13. Singh, G. et Green, M. Automating the Lexical and Syntaxic Design of Graphical User Interfaces : the UoA*UIMS. *ACM Transaction on Graphics.* 10, 3 (1991), pp. 213-254.
14. Smith, D. et Cypher, A. KidSim : Child Constructible simulation. In *Proceedings of Imagina'95* Monte-Carlo, Février), 1995, pp. 87-99.
15. Szekely, P. Retrospective and challenge for Model Based Interface Development. Bodart, F. et Vanderdonckt, J. (Ed.). In *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Springer-Verlag, Namur, Belgium, 1996, pp. 1-27.
16. Wiecha, C., Bennet, W., Boies, S., Gould, J. et Greene, S. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*. 8, 3 (1990), pp. 204-236.