Spécification du dialogue et génération d'interfaces à l'aide d'interacteurs à réseau de contrôle.

Gérard Tisseau, Hélène Giroire

Equipe SysDef - LIP6
Université Paris6, Boîte 169
4 Place Jussieu
F-75252 Paris Cedex 05
E-mail : Gerard.Tisseau@lip6.fr
E-mail : Helene.Giroire@lip6.fr
Tel : (33) 1 44 27 88 56

Jacques Duma

Lycée technique Jacquard
2 rue Bouret
75019 Paris
E-mail :dumajd@club-internet.fr

Françoise Le Calvez, Marie Urtasun

CRIP5
Université René Descartes
45 rue des Saints Pères
F-75270 Paris Cedex 06
E-mail : lecalvez@math-info.univ-paris5.fr
Tel : (33) 1 44 55 35 47

RESUME

Cet article présente d'abord un formalisme de spécification du dialogue d'une interface, apparenté aux réseaux de Petri, les interacteurs à réseaux de contrôle (IREC). Le réseau de contrôle d'un interacteur comporte des variables contenant des données et des commandes traitant ces données et les faisant circuler dans les variables en réponse aux événements d'affectation des variables. IREC offre des possibilités de composition par imbrication des interacteurs et de communication par partage de variables. Nous décrivons ensuite une architecture générique à objets, AGIREC, qui réifie les concepts du formalisme, qui les rend exécutables et qui permet de les relier aux widgets de la présentation. Nous présentons enfin un environnement interactif de développement d'interfaces, EDIREC, permettant d'éditer une spécification IREC et de la transformer en code exécutable fondé sur l'architecture AGIREC. EDIREC est intégré à VisualWorks et il a été produit à partir de lui-même par amorçage (bootstrap).

MOTS CLES : Interface Homme-Machine, spécification du dialogue, environnement de développement d'IHM.

INTRODUCTION

Les systèmes interactifs d'aide à l'apprentissage humain font jouer un rôle prépondérant à l'interface utilisateur. Celle-ci doit offrir à l'apprenant des activités au travers desquelles il pourra acquérir ou consolider des concepts et des démarches. Aux cours de ces activités, qui se présentent souvent sous forme de problèmes à résoudre, le système joue à la fois un rôle de guidage et d'évaluation, il doit laisser la possibilité à l'utilisateur de faire certaines erreurs (ce qui dans un contexte pédagogique, peut-être une bonne chose) mais sans conséquences néfastes, (l'action demandée par l'utilisateur n'est pas exécutée et l'erreur est expliquée). La réalisation de tels systèmes est difficile, car elle doit faire intervenir des compétences à la fois dans le domaine enseigné, en pédagogie et dans la conception d'interfaces. Trouver de bonnes activités pour enseigner un concept et concevoir de bonnes interfaces pour faire exercer cette activité à un apprenant sous le contrôle pédagogique d'un système est un processus encore hautement expérimental. Au cours de ce processus, de nombreuses interfaces doivent être réalisées, aussi bien pour tester différentes possibilités pour une même activité que pour proposer différentes activités couvrant un domaine d'étude. Il est alors indispensable de disposer d'outils facilitant la spécification, la réalisation, la modification et la réutilisation de ces interfaces.

De nombreux outils commerciaux, souvent désignés de manière générique sous le terme de Systèmes de Gestion d'Interfaces Utilisateurs (SGIU ou UIMS pour User Interface Management Systems), sont proposés actuellement pour faciliter la réalisation de systèmes interactifs [5]. La plupart d'entre eux concernent essentiellement la partie "présentation" du système : ils évitent au programmeur d'avoir à utiliser directement les primitives d'une boîte à outils ou d'un système de fenêtrage. Leur mise en œuvre est facilitée par l'existence d'éditeurs graphiques permettant de dessiner directement l'interface.

Ces outils rendent effectivement abordable la réalisation de la présentation d'un système interactif à des programmeurs non spécialistes des primitives graphiques et événementielles d'un environnement particulier, et il est avantageux de les utiliser. Cependant, ils laissent la gestion des autres aspects de l'interface à la charge du programmeur, en particulier la spécification et la réalisation du dialogue, comme par exemple la détermination des commandes qui doivent être rendues disponibles ou indisponibles suivant l'état du système. La plupart du temps, le programmeur doit incorporer ces aspects aux procédures de réaction aux événements et les mélanger ainsi à des appels au noyau fonctionnel du domaine. Or la gestion du dialogue est un aspect essentiel d'une interface pédagogique et elle doit pouvoir être traitée séparément, avec des outils appropriés permettant de spécifier le dialogue et d'engendrer une version exécutable de cette spécification. De nombreux travaux de recherche ont traité le problème du dialogue, en particulier pour définir des formalismes de spécification [4, 7]. La plupart de ces formalismes restent théoriques, mais certains d'entre eux sont exécutables [3, 2]. Cependant,

pour le moment, peu d'entre eux ont débouché sur des outils intégrés de génération d'interfaces accessibles sur différentes plates-formes de développement. C'est pour cette raison que nous avons été amenés à réaliser un outil de développement d'interfaces adapté à notre environnement de travail et à nos objectifs, dans le cadre d'un projet de système interactif d'aide à l'apprentissage humain. Dans ce projet nous créons de nombreuses interfaces "à manipulation indirecte" dont on peut trouver des exemples dans [9].

Dans cet article, nous décrivons d'abord le formalisme de spécification du dialogue que nous avons introduit, apparenté aux réseaux de Petri. Nous présentons ensuite une architecture à base de composants permettant de rendre exécutable une spécification. Puis nous décrivons l'environnement interactif que nous avons réalisé et qui permet d'éditer une spécification et d'engendrer le code exécutable des composants associés. Cet environnement présente la particularité d'avoir été spécifié et engendré à partir du formalisme lui-même par un processus d'amorçage (bootstrap).

LE FORMALISME DE SPECIFICATION IREC

Notre formalisme est inspiré des réseaux de Petri à objets et rejoint en cela le formalisme ICO [1]. Nous avons cependant développé notre propre variante, essentiellement dans un but de simplification et d'adaptation aux interfaces que nous considérons. Nous avons choisi ce type de représentation parce que c'est un des seuls qui permette de prendre en compte les données et leur circulation au même titre que les actions et leur séquencement.

Interacteurs, variables et commandes

L'unité de structuration du formalisme est appelé un *interacteur*. Ce terme est utilisé de différentes manières dans la littérature et il tend actuellement à désigner les composants graphiques de base d'une interface (widgets, contrôles). Certains l'utilisent cependant avec un sens plus générique et plus abstrait [6] et c'est ce que nous ferons dans cet article, en réservant le terme de *widget* pour les composants de base.

Un **interacteur** est destiné à modéliser le comportement d'une fenêtre d'interface ou d'une sous-fenêtre (une région rectangulaire incluse dans une fenêtre). Décrire le comportement revient à préciser à tout moment quelles actions sont applicables à quels objets, et quel est le résultat de l'application d'une action. Dans notre modèle, les entités responsables de l'exécution des actions s'appellent des *commandes* et les entités destinées à contenir des objets s'appellent des *variables*. Un interacteur contient des commandes et des variables, structurées en un graphe biparti du type réseau de Petri : les variables jouent un rôle analogue aux places et les commandes jouent un rôle analogue aux transitions. Nous appellerons ce graphe le *réseau de contrôle* de l'interacteur. La figure 1 représente un interacteur et son réseau de contrôle.

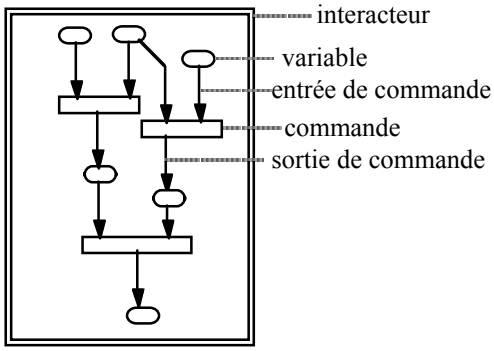

Figure 1 : Un interacteur et son réseau de contrôle

Les variables d'entrée d'une commande contiennent les objets nécessaires à l'exécution de l'action associée à la commande, soit parce que la commande les utilise ou les modifie pour produire des résultats (qui seront affectés aux variables de sortie), soit parce qu'ils interviennent dans des préconditions indiquant si l'exécution de la commande est autorisée.

Une **variable** peut contenir à tout moment une valeur, qui est un objet quelconque, ou aucune. C'est une restriction et une simplification par rapport aux réseaux de Petri à objets généraux, dont les places peuvent contenir plusieurs jetons étant eux-mêmes des n-uplets d'objets. Cette simplification permet de faire jouer un rôle important à deux sortes d'événements : la *validation* d'une variable (lorsqu'une variable sans valeur acquiert une valeur) et son *invalidation* (lorsqu'une variable qui a une valeur perd cette valeur). Pour une variable donnée d'un interacteur donné, ces événements peuvent avoir seulement trois causes : 1) l'utilisateur peut affecter une valeur à la variable par l'intermédiaire d'un widget (ce qui n'est possible que si la variable a été déclarée affectable par l'utilisateur), 2) l'interacteur lui-même peut affecter une valeur à la variable comme résultat de son fonctionnement interne et 3) un autre interacteur peut affecter une valeur à la variable (ce qui n'est possible que si celle-ci est déclarée être *partagée* par les deux interacteurs, ce qui sera expliqué plus loin).

Les variables directement affectables par l'utilisateur ont un rôle spécial : c'est uniquement à travers elles que l'interacteur est informé des actions de l'utilisateur sur la présentation. Pour cette raison, une commande ne peut avoir au plus qu'une de ces variables en entrée et aucune en sortie. Nous les désignerons dans la suite sous le terme de *déclencheurs* et nous réserverons le terme de variable aux autres. Un déclencheur est l'analogue d'une place utilisateur dans le formalisme ICO.

Une **commande** peut se trouver dans deux états : *disponible* ou *indisponible*. Elle répond aux événements concernant ses variables d'entrée de différentes façons suivant son état. Pour prendre une

décision, elle prend en compte différents éléments : les variables d'entrée et deux conditions booléennes jouant le rôle de *préconditions* : l'une s'appelle *autorisation* et l'autre *acceptation*. Chaque commande possède ces deux préconditions (par défaut elles sont toujours vraies). De même, chaque commande possède une *action* indiquant le traitement à effectuer lorsque la commande est effectivement déclenchée et que ce déclenchement est accepté. Cette action peut comporter des envois de message aux objets valeurs des variables d'entrée de la commande (ou valeur de son déclencheur) et des affectations de valeurs aux variables de sortie.

Par définition, une commande est *déclenchable* si ses variables d'entrée sont valides et si sa précondition d'autorisation est vérifiée. Les commandes qui n'ont pas de déclencheur exécutent automatiquement leur algorithme de déclenchement par réflexe sur chaque événement qui les rend déclenchables. Le déclenchement s'exprime par l'algorithme suivant :
"si acceptation vérifiée alors action sinon refus".
Lorsqu'une commande possédant un déclencheur devient déclenchable (indéclenchable), elle passe à l'état disponible (indisponible). Pour une telle commande, son déclenchement, envisageable lorsqu'elle est disponible, ne peut s'effectuer que sur l'événement "affectation du déclencheur" ce qui s'exprime par la règle suivante :
"Réaction quand état disponible :
Si déclenchable et affectation déclencheur alors
si acceptation vérifiée alors action sinon refus"

L'introduction de deux types de préconditions (autorisation et acceptation) résulte d'un besoin lié aux interfaces pédagogiques. L'autorisation permet de décider si on rend réactif le widget associé à une commande, alors que l'acceptation examine si une action de l'utilisateur sur ce widget (lorsqu'il est réactif) correspond à une demande valide dans le contexte pédagogique. Cela permet de régler le niveau de guidage et de protection de l'utilisateur : certaines actions sont rendues visiblement impossibles par le jeu des autorisations (les widgets sont grisés) alors que d'autres sont apparemment permises mais déclarées invalides lors d'une tentative de déclenchement injustifiée, par le jeu des acceptations. On laisse ainsi à l'utilisateur la possibilité de faire certaines erreurs (ce qui, dans un contexte pédagogique, peut être une bonne chose) mais sans conséquences néfastes (l'action n'est pas exécutée et l'erreur est expliquée).

Ce comportement générique d'une commande peut être complété par des traitements supplémentaires en fonction de deux propriétés que peut posséder la commande : elle peut être *consommatrice* ou non et elle peut être *invalidante* ou non. Lorsqu'une commande est consommatrice, elle rend invalides ses variables d'entrée après chaque déclenchement. C'est le comportement habituel des transitions d'un réseau de Petri. Mais il peut être pratique dans certains cas de ne pas consommer les entrées. Une commande invalidante rend invalides ses sorties dès qu'une de ses entrées devient invalide. Ce comportement n'est pas standard dans les réseaux de Petri. Nous l'avons introduit pour faciliter certains effets d'annulation lorsque la commande n'est pas consommatrice. L'idée est la suivante : la présence simultanée de valeurs en entrée et en sortie exprime souvent une relation entre elles, la sortie étant par exemple une fonction de l'entrée. Si l'entrée devient invalide, cette relation n'est plus vérifiée ou n'a plus de sens et il faut alors invalider la sortie pour rétablir la cohérence. C'est par exemple le cas lorsque l'entrée est une liste de valeurs et que la sortie est le résultat d'une sélection d'une valeur dans cette liste. Si l'entrée est invalidée, il n'y a plus de liste et on ne peut plus dire que la sortie est un élément de la liste d'entrée. On invalide alors la sortie, et cela se fait automatiquement si la commande est déclarée invalidante.

Mécanismes de structuration et de communication

Un inconvénient reconnu des réseaux de Petri est leur manque de possibilités de structuration, de composition et de coopération. Différentes solutions ont été proposées pour y remédier. Dans le formalisme ICO, par exemple, un Objet Coopératif X peut coopérer avec un autre objet Y par l'intermédiaire d'un jeton représentant Y présent dans une des places de X. Une transition de X peut alors adresser des messages à Y sous forme de demandes de service suivant un protocole client-serveur. Nous avons retenu une autre solution, avec une approche plus structurelle et utilisant un mode de communication par flux de données plutôt que par envoi de messages.

Dans notre formalisme, un interacteur peut en contenir d'autres, ce qui permet de définir une structure arborescente récursive dans laquelle un interacteur père peut avoir des interacteurs fils, repérés dans le contexte du père par des noms locaux indiquant leurs rôles par rapport au père. Cette décomposition permet la conception modulaire des interacteurs et leur réutilisation. De plus, elle est bien adaptée au mécanisme des sous-vues que proposent certains Systèmes de Gestion d'Interfaces Utilisateur, dans lesquels on peut définir indépendamment différentes vues (ou formulaires, ou canevas) et inclure une vue déjà définie dans une vue en cours de construction, et cela récursivement.

La communication entre interacteurs ne peut s'effectuer que par l'intermédiaire de *variables partagées*. Lorsqu'on inclut un interacteur fils dans un interacteur père, on déclare que certaines variables du père doivent être partagées avec certaines variables du fils. Sur la figure 3, la variable X du père est partagée avec la variable Y du fils F. X est le nom d'une variable dans le contexte du père, F est le nom d'un fils (c'est-à-dire d'un rôle) dans ce même contexte et Y est le nom d'une

variable dans le contexte du fils. Il s'agit ainsi d'une sorte de plan de branchement d'un composant à l'intérieur d'un autre, les variables jouant le rôle de bornes de branchement.

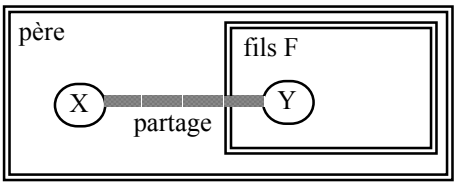

Figure 3 : Inclusion d'un fils et partage de variables

Du point de vue de leur comportement, dire que deux variables sont partagées revient à dire qu'elles sont identifiées physiquement pour ne plus former qu'une seule variable participant au réseau de contrôle du père aussi bien qu'à celui du fils. De cette façon, on peut ainsi construire modulairement un réseau complexe, en réutilisant des réseaux déjà définis. Par le jeu de partages à différents niveaux de la hiérarchie, une même variable peut finalement être partagée entre plusieurs générations successives (un père, un fils et un petit-fils par exemple) ou entre deux frères (par l'intermédiaire du père).

Ce type de communication relève plus du flux de données (data-flow) que de l'envoi de messages. En effet, la communication n'est pas dirigée : lorsqu'une variable partagée est affectée, elle émet un événement qui est pris en compte aussi bien par le père que par le fils, sans se soucier de l'origine de l'affectation. On peut donc concevoir un interacteur de manière indépendante et l'utiliser plus tard comme fils d'un autre interacteur, sans prévoir à l'avance à qui il enverra des messages.

États et transitions d'un interacteur

Dans un réseau de Petri, l'état du réseau est entièrement déterminé par le contenu des différentes places (le marquage), sans qu'il soit nécessaire de définir a priori une liste d'états possibles comme dans les automates de transition (transition networks), ce qui permet une représentation plus concise et augmente les possibilités de comportement dynamique. On constate cependant souvent dans les exemples que certaines places sont utilisées pour représenter en fait des états globaux bien définis, comme "début", "fin", "il y a une donnée à éditer" ou "modification en cours". C'est le cas en particulier dans les interfaces à but pédagogique que nous construisons. Pour prendre en compte l'apparition fréquente de certains motifs (patterns) de ce type, nous avons associé à chaque interacteur un automate de transition indiquant les états globaux qu'on retrouve dans tout interacteur et les transitions entre ces états. Cet automate complète le réseau de contrôle. Il sert en fait seulement d'abréviation, car il pourrait être traduit directement dans le réseau. Mais cela rendrait celui-ci plus complexe et cela serait fastidieux car il faudrait répéter la même structure dans chaque réseau. L'automate étant défini à part une fois pour toutes, chaque interacteur n'a plus à redéfinir cette structure commune, mais seulement à préciser les préconditions et les actions des transitions qui y figurent.

Un interacteur peut se trouver dans quatre états globaux : *invalide*, *prêt*, e*n cours* et *complet*. Ce choix traduit la vision selon laquelle le but principal d'un utilisateur utilisant un interacteur est de produire un résultat (créer ou éditer un ou plusieurs objets) et les états choisis indiquent des degrés d'avancement dans l'accomplissement de cette tâche.

La figure 4 montre les états et les transitions possibles. Une transition ne peut se produire que lors d'un événement de validation ou d'invalidation d'une variable. Le dessin de l'automate sépare ces deux cas pour une meilleure lisibilité.

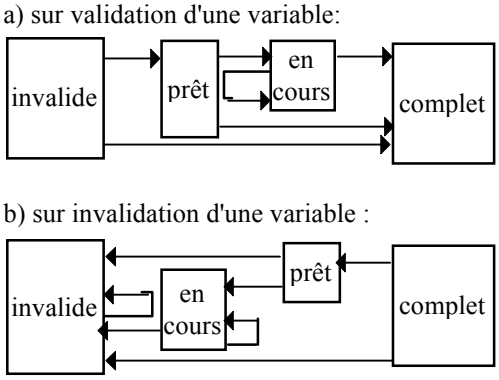

Figure 4 : Etats et transitions d'un interacteur

La détermination de l'état courant et des transitions à exécuter fait jouer un rôle particulier à certaines variables qui doivent être distinguées dans le réseau de contrôle. Pour distinguer ces variables, on a ajouté au formalisme la notion de *statut* d'une variable. Les statuts possibles sont : variable d'*entrée*, variable *interne* ou variable de *sortie*. Les variables d'entrée doivent être valides pour que l'interacteur puisse être utilisable et les variables de sortie contiennent les objets constituant le résultat de l'activité de l'interacteur. Lorsqu'un événement survient, l'interacteur choisit la transition qui le fait passer dans l'unique état dont la *condition d'activation* est vérifiée. La figure 5 montre les quatre conditions d'activation. L'une d'entre elles fait intervenir la *précondition d'autorisation* de l'interacteur. C'est une condition vraie par défaut, mais que chaque interacteur peut spécialiser.

Prêt : les variables d'entrée sont toutes valides, les variables de sortie ne sont pas toutes valides et la précondition d'autorisation est vérifiée. En cours : même condition que prêt. Complet : les variables d'entrée sont toutes valides et les variables de sortie aussi. Invalide : les conditions d'activation des autres états sont toutes fausses.

Figure 5 : Conditions d'activation des états

Tous les interacteurs possèdent donc le même comportement global ainsi défini par ces états et ces transitions, en plus de leur comportement interne défini par leur réseau de contrôle. Ce comportement global peut toutefois être personnalisé : le concepteur peut définir la précondition d'autorisation, ainsi que les actions d'*initialisation* à effectuer lors du passage dans chacun des quatre états. Ces actions consistent essentiellement à affecter des valeurs à des variables de l'interacteur. Ce choix de modélisation, qui restreint apparemment la liberté de conception, s'est révélé avantageux : d'une part, une partie du mécanisme est prédéfinie et n'a pas à être explicitée à chaque fois, et d'autre part cela assure une bonne cohérence des interfaces, fondée sur une vision assez naturelle (un interacteur sert à produire un résultat).

Liens avec la présentation

Jusqu'à présent, nous avons seulement décrit la spécification du dialogue. Précisons maintenant comment relier le dialogue à la présentation. Celle-ci est définie par un ensemble de widgets, chacun d'entre eux pouvant être associé soit à une commande, soit à une variable, soit à l'interacteur lui-même.

Un widget associé à une commande peut communiquer avec celle-ci en affectant une valeur au déclencheur de cette commande. C'est le cas par exemple d'un menu permettant de sélectionner une valeur dans une liste : la valeur sélectionnée est affectée au déclencheur, ce qui permet à la fois de signaler un événement et de transmettre une donnée. Inversement, la commande connaît le widget et lui demande de se rendre disponible ou indisponible à chaque fois qu'elle-même devient disponible ou indisponible. De plus, lorsqu'elle devient disponible, la commande peut transmettre au widget des informations concernant son contenu, comme par exemple la liste des valeurs apparaissant dans un menu (à partir d'une liste d'objets présente dans une des variables d'entrée de la commande). De même, un widget associé à une variable est actualisé en fonction du contenu de la variable à chaque affectation de cette variable. C'est le cas par exemple d'un champ de texte destiné à afficher la valeur d'une variable. Un widget associé à l'interacteur est utilisé uniquement pour l'affichage (il est appelé un *décor*) et peut être modifié par l'interacteur lors d'un changement d'état de celui-ci. Il peut par exemple être rendu disponible ou indisponible, visible ou invisible. C'est le cas par exemple d'une étiquette statique La notion de visibilité permet de spécifier une partie du retour de l'information

La visibilité ou l'invisibilité des widgets est spécifiée déclarativement : pour chaque composant d'un interacteur (commande, variable, interacteur fils ou décor), on indique pour chacun des quatre états de l'interacteur si le composant en question est visible ou invisible. A chaque changement d'état, l'interacteur se chargera automatiquement de rendre visibles ou invisibles les widgets concernés. On voit là un intérêt du choix de définir explicitement des états globaux pour un interacteur : on peut associer à ces états des informations déclaratives décrivant un aspect du comportement.

L'ARCHITECTURE AGIREC

Pour faciliter la mise en œuvre d'une spécification exprimée dans le formalisme IREC, nous avons défini une architecture logicielle à base d'objets (AGIREC), dans laquelle les concepts sont représentés par des classes. Les classes principales sont Interacteur, Commande et Variable. Ces classes s'insèrent dans la hiérarchie de la bibliothèque de classes de l'environnement de programmation VisualWorks, ce qui permet de profiter de la gestion des widgets et des événements. En fait, les concepts IREC ne sont pas tous réifiés. En particulier, l'automate de transition d'un interacteur n'est pas représenté sous forme d'objet mais sous forme de méthode.

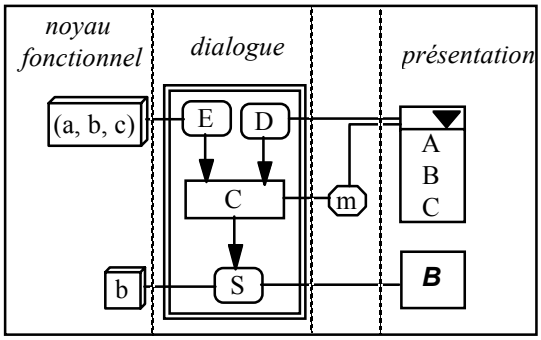

Figure 6 : Principe de l'architecture

La figure 6 montre le principe de l'architecture sur un exemple. Le dialogue est géré par un interacteur. La variable E contient un objet du domaine d'application, ici la collection (a, b, c). La commande C, de classe CommandeMenu, est liée à la présentation par l'intermédiaire du déclencheur D et d'un auxiliaire m dont le rôle est d'actualiser le contenu du menu (widget) en fonction du contenu de E. La figure montre l'état lorsque l'utilisateur a choisi l'option B dans le menu : la variable S a été affectée avec la valeur b, et un afficheur associé à S présente le résultat de ce choix avec une mise en forme appropriée. Le lien entre le dialogue et le noyau fonctionnel s'effectue par l'intermédiaire des variables qui contiennent des objets

du domaine et par les préconditions et actions des commandes et de l'interacteur, qui peuvent adresser des messages à ces objets.

Pour réaliser une interface, le programmeur doit définir une sous-classe de la classe Interacteur et fournir une méthode de création d'instances qui aura pour effet de créer et de mettre en place différentes instances des classes Commande et Variable, de manière à construire une structure reproduisant le réseau de contrôle, cette structure étant elle-même reliée aux widgets de la présentation. Un interacteur est relié à ses composants, c'est-à-dire ses variables, ses commandes et ses fils, qui sont eux-mêmes des instances de sous-classes d'Interacteur.

Une fois ces éléments mis en place, leur fonctionnement résulte des actions de l'utilisateur qui provoquent des événements émis par les widgets et transformés en affectations de valeurs aux déclencheurs des commandes. Le réseau fonctionne alors comme indiqué précédemment, en réaction aux affectations des variables. Les méthodes prédéfinies se chargent de provoquer les événements de contrôle (lors de l'affectation d'une variable) et de les transmettre aux objets concernés (commandes, interacteurs). Ces objets réagissent en invoquant les différentes méthodes correspondant à leurs préconditions et actions. Le comportement générique est ainsi inscrit dans les classes prédéfinies.

Pour adapter le fonctionnement générique à un interacteur particulier, c'est-à-dire à une sous-classe de la classe Interacteur, il faut spécialiser le code de certaines méthodes appelées par l'architecture générique. Pour chaque état, on fournit sa méthode d'initialisation. De plus, on fournit la précondition d'autorisation de l'état prêt.

Le comportement des variables n'a pas à être adapté : il résulte des méthodes prédéfinies dans la classe Variable et des liens des variables avec les commandes et les interacteurs.

Le comportement d'une commande doit être spécialisé en fournissant le code des méthodes de préconditions (autorisation et acceptation) et celui de la méthode d'action. Contrairement au cas des interacteurs, où chaque nouvel interacteur correspond à une nouvelle sous-classe, le programmeur ne doit pas définir une nouvelle sous-classe pour chaque nouvelle commande. Cela conduirait à introduire de très nombreuses sous-classes qui ne seraient pas réutilisables car elles n'auraient qu'une seule instance (la sémantique d'une commande est très dépendante de l'interacteur dans lequel elle est placée et de ses connexions). Mais faire ce choix demande de pouvoir programmer des méthodes instance par instance et non plus classe par classe. Certains langages à objets le permettent, en particulier ceux qui sont fondés sur la notion de *prototypes* comme Self [8]. Certains environnements de développement d'interfaces utilisent ce type de modèle objet, comme Amulet [6]. Le langage Smalltalk n'étant pas dans ce cas, nous utilisons la technique consistant à programmer le code de la méthode dans la classe de l'interacteur, et à installer dans une variable d'instance de la commande une référence à cette méthode (par son nom). Par exemple, la méthode d'action de la commande de nom "valider" s'appellera "valider_action", et c'est ce nom qui sera contenu dans la variable d'instance "action" de la commande.

Plusieurs classes de commande sont prédéfinies dans l'architecture générique, chacune d'entre elles étant adaptée à un certain type de widget. Il existe par exemple une classe pour représenter toute commande de choix d'une valeur par l'intermédiaire d'un menu. Ces classes disposent de méthodes d'initialisation pour relier la commande à un widget du type correspondant et pour assurer la communication avec ce widget d'une manière conforme à la spécification. Par exemple la commande rendra le widget indisponible quand elle-même deviendra indisponible. Une dizaine de classes de commande sont actuellement implémentées, et cela suffit pour les usages courants.

L'ENVIRONNEMENT DE DEVELOPPEMENT

Nous avons défini et réalisé un environnement de développement d'interfaces à réseaux de contrôle (EDIREC) qui permet d'éditer une spécification d'interface dans le formalisme IREC, qui engendre la quasi totalité du code exécutable de l'interface avec ses widgets dans l'architecture AGIREC et qui assiste le programmeur d'interface pour compléter ce code et le mettre au point. La figure 7 présente la fenêtre principale de EDIREC.

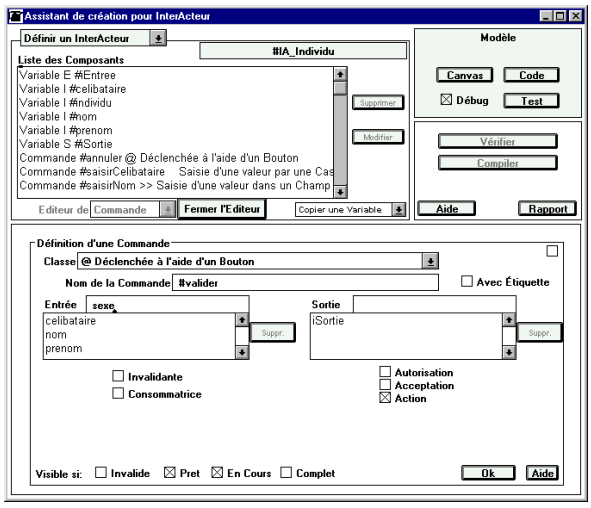

Figure 7 : Interface EDIREC

Cette fenêtre comporte trois grandes zones, deux concernant la spécification (la liste des composants et la zone d'édition d'un composant) et la troisième les

fonctions de génération et de mise au point du code Smalltalk de l'interface. La Figure 8 montre cette structure.

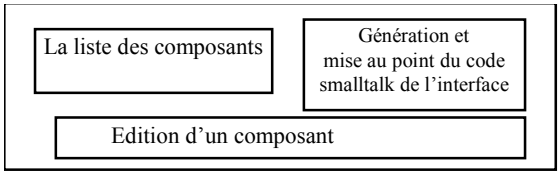

Figure 8 **:** Structure de la fenêtre d'EDIREC

La spécification

La figure 7 présente l'interface d'EDIREC au cours de la spécification d'une commande de validation. Cette commande est sous classe de la classe prédéfinie CommandeBouton. EDIREC a rempli certains champs par des valeurs par défaut propres à la CommandeBouton, le programmeur modifie et/ou complète.

La figure 9 présente la sous-fenêtre de la spécification d'un fils de l'interacteur en cours de création, lorsqu'on branche une variable du fils à une variable du père :

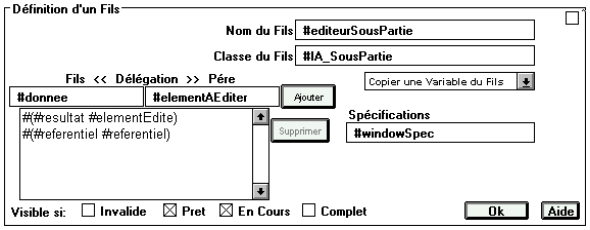

Figure 9 : Spécification d'un fils

Le développement du code

Après avoir terminé la spécification le programmeur demande sa vérification (bouton vérifier). EDIREC vérifie que toute les références à des composants correspondent bien à des composants définis, du bon type et sans doublon.

Lorsque la vérification a réussi, le programmeur demande la compilation (bouton compiler), EDIREC produit alors la quasi totalité du code ; il crée une sous-classe de la classe Interacteur dont la méthode de création d'instances engendre automatiquement toutes les instances des composants de la spécification, les instances des widgets et des objets auxiliaires nécessaires. EDIREC fournit un rapport après la compilation (bouton rapport), contenant les informations nécessaires au programmeur pour compléter le code : les noms (engendrés automatiquement par EDIREC) des méthodes à programmer et leurs rôles. Par exemple il peut comporter l'avertissement suivant : "ATTENTION Ne pas oublier de programmer #valider_Action, action de la commande #valider". Le programmeur peut accéder aux fenêtres usuelles d'édition de code de Smalltalk (bouton code) tout en gardant EDIREC disponible. Le code peut être immédiatement testé (bouton test) avec des facilités de mise au point.

Le programmeur doit configurer la présentation. Les widgets ont été créés automatiquement et placés arbitrairement lors de la compilation. Certaines de leurs propriétés ont été définies automatiquement de manière à les relier aux interacteurs, variables et commandes. Il reste à les dimensionner, les placer et éditer leurs propriétés graphiques. EDIREC étant entièrement intégré à l'environnement VisualWorks, il donne accès directement (bouton canvas) au Système de Gestion d'Interface Utilisateur de ce dernier.

Développer avec EDIREC

Le cycle de développement est résumé dans la figure 10. Les libellés *compiler*, *code*, *canvas, test* font référence aux boutons du même nom.

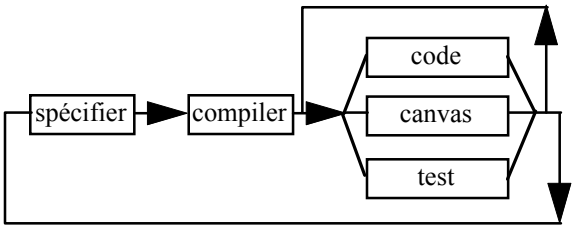

Figure 10 : Cycle de développement avec EDIREC

Cette démarche est fondée sur les modèles (model-based) plutôt que sur la visualisation : on ne commence pas par dessiner la présentation. Les widgets sont créés d'après la spécification. Par exemple, si on déclare qu'une variable doit être visible, alors un widget correspondant est créé. La méthode de compilation est adaptée au cycle de développement : la recompilation ne modifie pas les widgets déjà créés (s'ils restent nécessaires), ce qui permet de conserver leurs propriétés graphiques d'un cycle à l'autre. Il en va de même pour le code supplémentaire écrit par le programmeur.

La compilation sauvegarde la spécification dans le code même de la classe de l'interacteur. Cela permet de rééditer avec EDIREC tout interacteur qui a été créé avec EDIREC. Cela permet également de créer un nouvel interacteur par recopie d'un autre, soit totalement, soit partiellement (le réseau de contrôle seulement). Le programmeur peut ainsi isoler et réutiliser des schémas de programmation (design patterns).

Aussi bien EDIREC que les applications créées avec EDIREC sont portables, en raison de son intégration à VisualWorks qui possède cette même propriété. Nous avons fait fonctionner EDIREC et les interfaces créées sur trois plates-formes (Windows, Unix et Macintosh).

Applications

L'une des premières applications importantes créées avec EDIREC a été … EDIREC lui-même ! En effet, il s'agit d'une application interactive où l'interface joue un rôle important pour aider l'utilisateur dans sa tâche. A ce titre, sa réalisation est susceptible d'être facilitée par un environnement de développement comme EDIREC. Il a fallu bien sûr amorcer le processus : nous avons d'abord programmé une version EDIREC 0 à la main, en utilisant les possibilités offertes par l'architecture AGIREC, puis nous avons utilisé EDIREC 0 pour spécifier et développer une application équivalente EDIREC 1. La comparaison entre les efforts de développement de la version 0 et de la version 1 a suffi pour nous convaincre de l'utilité de l'outil EDIREC. Désormais, chaque nouvelle version n + 1 de EDIREC est produite avec la version n.

Une autre application en cours est la production d'interfaces pédagogiques pour un système d'aide à l'apprentissage humain [9].

CONCLUSION

Nous avons présenté le formalisme IREC de spécification du dialogue d'une interface, l'architecture AGIREC permettant de mettre en œuvre une telle spécification et l'environnement de développement EDIREC permettant d'éditer une spécification et d'engendrer automatiquement la plus grande partie du code de l'interface. IREC est un formalisme déclaratif apparenté aux réseaux de Petri, offrant des possibilités de composition par imbrication d'interacteurs avec partage de variables. AGIREC est une architecture générique à objets permettant au programmeur de rendre exécutable une spécification IREC en spécialisant des classes et des méthodes prédéfinies qui implémentent le comportement générique des interacteurs, variables et commandes. EDIREC est un environnement interactif de développement intégré à VisualWorks automatisant la plus grande partie de la production du code d'une interface à partir d'une spécification IREC. Il a été produit par amorçage avec lui-même (bootstrap).

Ces outils, initialement conçus pour des interfaces pédagogiques, ne sont en fait pas limités à ce cadre. Ils sont adaptés à la production d'interfaces dites "à manipulation indirecte". Ils permettent une expression déclarative fondée sur un modèle dans lequel les données et leur circulation jouent un rôle aussi important que les actions et leur séquencement. Ce modèle n'offre pas toutes les possibilités qu'on trouve dans un formalisme comme ICO, mais il n'a pas les mêmes buts. Nous avons surtout cherché à obtenir un ensemble d'outils opérationnels sur plusieurs plateformes pour expérimenter rapidement des interfaces à objectifs pédagogiques. Nous avons privilégié pour cela la simplicité d'expression pour les cas courants, les possibilités de réutilisation (aussi bien par composition que par recopie différentielle), un cycle de développement court et l'intégration à un environnement portable offrant de bons outils de création d'interfaces (VisualWorks).

EDIREC est un outil opérationnel qui est effectivement utilisé : nous l'utilisons de manière régulière pour construire les interfaces d'un système d'aide à l'apprentissage humain. Cette expérience contribue à son évaluation et à son évolution. Une amélioration souhaitable, par exemple, serait d'intégrer à EDIREC un éditeur graphique permettant de dessiner le réseau de contrôle.

BIBLIOGRAPHIE

1. Bastide R. & Palanque Ph. (1990) Petri nets with objects for the design, validation and prototyping of user-driven interfaces. In D. Diaper, D. Gilmore, G. Cockton, B. Shalckel (eds.), *Proceedings INTERACT'90*, Elsevier Science Publishers (North-Holland), pp. 625-631.
2. Bastide R. & Palanque Ph.(1995) A Petri Net Based Environment for the Design of Event-Driven Interfaces, ATPN'95, Torino, Italy *LNCS n°935*, pp. 66-83, Springer-Verlag.
3. Browne T., Davilla D., Rugaber S. & Stirewalt K. Using Declarative descriptions to Model User Interfaces with MASTERMIND. In *Formal Methods in Human-Computer Interaction* Palanque Ph. & Paternò F.(eds) Springer Verlag 1997.
4. Dix A. *Formal Methods for Interactive Systems*, Academic Press 1991.
5. Myers Brad A. User Interface Software Tools, *ACM Transactions on Computer Human Interaction*. 1995. **2** (1) pp. 64-103.
6. Myers Brad A, McDaniel Richard G., & Miller Robert C. The Amulet Prototype-Instance Framework. In *Object-Oriented Application Frameworks*, **3**,.M. Fayad and D. C. Schmidt (eds.). New york : John Wiley & Sons, 1999.
7. Palanque Ph. & Paternò F.(eds*).Formal Methods in Human-Computer Interaction* Springer Verlag 1997.
8. Ungar D. & Smith Randall B. Self : The Power of Simplicity, *SIGPLAN Notices*. 1987. **22** pp.241-247. ACM OOPSLA'87.
9. Le Calvez F., Urtasun M., Tisseau G., Giroire H.& Duma J. "Les machines à construire : des modèles d'interaction pour apprendre une méthode constructive de dénombrement" *EIAO'97*, Cachan, M. Baron, P. Mendelsohn, J.F. Nicaud (eds), Hermès, 1997, pp.49-60.