GESTION AUTOMATIQUE DE LA COHERENCE DE L'INTERFACE UTILISATEUR AVEC L'ÉTAT DE L'APPLICATION

David Julien,
Doctorant en Informatique
david.julien@lip6.fr, + 33 1 44 27 54 80

Mikal Ziane,
Maître de conférences (HDR) en Informatique
mikal.ziane@lip6.fr, + 33 1 44 27 87 46

Zahia Guessoum,
Maître de conférences (HDR) en Informatique
zahia.guessoum@lip6.fr, +33 1 44 27 87 43

Adresse professionnelle
Laboratoire d'Informatique de Paris 6, Pôle IA ★ 8 rue du Capitaine Scott ★ 75015 PARIS

Résumé : Malgré les progrès des outils de développement d'interfaces utilisateur, il est difficile de garantir que les données exposées dans une interface restent cohérentes avec les données de l'application. En effet, cette cohérence est encore souvent gérée à la main, et une erreur ou un oubli suffit à la compromettre. Nous proposons donc une solution pour automatiser la gestion de cette cohérence. Cette solution est fondée sur l'intégration des connaissances des concepteurs dans les outils de conception et d'exécution des interfaces. Le concepteur peut alors déléguer cette activité à ces outils et consacrer plus de temps à travailler sur la qualité de l'interface.

Summary : Despite improvement in user-interface development tools, guaranteeing that what is displayed accurately reflects the application's data is still a difficult task. It is however still handled manually by interface designers. We thus propose a solution to automate this task by integrating interface designers' relevant knowledge into support tools. Interface Designers will then be able to delegate this task to such tools and focus on the quality of the interface.

Mots clés : Interface utilisateur, Représentation des connaissances, Modèles conceptuels.

Gestion Automatique de la Cohérence de l'Interface Utilisateur avec l'État de l'Application

La conception de l'interface utilisateur représente à elle seule 48% du code et 50% du temps d'implémentation d'une application (Myers 95). Malgré le succès des outils de construction visuelle, le problème de la conception de l'interface est toujours d'actualité. En effet, ces outils permettent de définir facilement la présentation de l'interface mais offrent peu de solutions pour relier la présentation à l'application. Les relations entre l'application et la présentation sont donc essentiellement implémentées à la main par les concepteurs.

Notre objectif général est de faciliter la conception des interfaces utilisateurs. Pour cela, cette conception doit être automatisée en demandant au concepteur de décrire son interface sous la forme d'une spécification déclarative, en tout cas abstraite, et en générant le code de l'interface à partir de cette description. Les environnements fondés sur cette approche sont appelés MB-UIDE (*Model-Based User Interface Development Environment*) (Silva 00).

Nous nous sommes notamment intéressés à un problème récurrent : la gestion de la cohérence de l'interface avec l'état de l'application. Appelons *s* une donnée de l'application qui doit être reflétée dans l'interface. Appelons *o* une donnée de la présentation qui doit refléter *s*. Il existe donc une fonction f telle que $o = f(s)$. Le problème de la cohérence consiste à garantir que cette égalité soit maintenue.

Dans cet article, nous proposons une solution au problème de la cohérence dans le cadre des MB-UIDE et nous décrivons sa réalisation dans GOLIATH. GOLIATH (Julien et al. 04) est un environnement à base de modèles reposant sur l'idée que les connaissances des concepteurs doivent être intégrées dans les outils de conception et d'exécution. L'objectif de cette démarche est de faciliter la conception des interfaces utilisateur en éliminant la résolution manuelle et *ad hoc* de certains problèmes récurrents grâce à l'application automatique de techniques éprouvées utilisées par les experts en conception d'interface. L'utilisation de modèles déclaratifs permet de faire abstraction des détails d'implémentation et d'appliquer ces techniques quel que soit le langage d'implémentation de l'application et quelle que soit la bibliothèque de présentation utilisée.

Notre article est organisé comme suit. Dans la section 1 nous montrons pourquoi le problème de la cohérence n'est pas résolu dans les outils commerciaux actuels, puis en quoi consistent les approches à base de modèles. Dans la section 2 nous définissons plus précisément le problème de la cohérence et nous exposons deux techniques classiques pour le résoudre « manuellement ». Dans les sections 3 et 4 nous présentons GOLIATH, notre outil d'aide à la conception, et nous montrons comment les connaissances permettant de gérer la cohérence sont intégrées dans GOLIATH pour automatiser la gestion du problème de la cohérence. Notre article sera illustré par la création d'une interface pour un gestionnaire de carnets d'adresses dont le résultat est présenté à la section 5.

1 – CONTEXTE ACTUEL

Nous présentons à présent les principaux environnements permettant la conception d'interfaces utilisateurs et nous évaluons comment ils résolvent le problème de la cohérence.

1.1 – Les environnements commerciaux

La plupart des environnements de développement rapide tels que *Delphi* d'*Inprise*, *Visual Studio.NET* de *Microsoft* et *Eclipse* d'*IBM* permettent la construction d'interfaces suivant le paradigme WYSIWYG (« *What You See Is What You Get* »). La sélection des éléments constituant la présentation (appelés *widgets*) et leur positionnement dans l'interface s'effectuent de manière visuelle par *glisser-déposer*.

Toutefois, si ces environnements permettent une construction rapide de la présentation de l'interface, ils ne proposent pas de solutions permettant de décrire proprement les relations entre cette présentation et le noyau fonctionnel. Ces relations doivent donc être programmées à la main par le développeur de l'interface. La solution la plus courante consiste à modifier le code produit par le constructeur visuel pour y ajouter des appels aux fonctions de l'application, ou encore à modifier l'application elle-même pour décrire l'accès à la présentation. Ces techniques ne répondent donc pas au principe de séparation entre le noyau fonctionnel et la présentation. De plus, des limitations apparaissent également dans la construction de la présentation en elle-même. Chaque environnement est conçu pour fonctionner avec une boîte à outils en particulier, et l'approche ne s'applique qu'aux interfaces dites WIMP (« *Windows, Icon, Menu, Pointer* »).

Certains de ces environnements proposent des « *wizards* » (i.e. des assistants) facilitant la construction de l'interface. Dans *WinDev* de

PCSoft, par exemple, lorsque le concepteur ajoute un widget dans une fenêtre, l'outil interroge le concepteur pour notamment connaître l'origine des données à exposer dans ce widget (par lecture dans un fichier, par requête dans une base de données, par saisie manuelle). Il procède alors automatiquement à l'initialisation du widget. Lorsque la source de données est un fichier ou une table, *WinDev* permet de décrire facilement que la donnée exposée par un widget correspond à un/plusieurs champs de cette table ou encore que la donnée exposée dépend de la sélection effectuée dans un autre widget. Ces interfaces permettant de paramétrer les widgets sont plus conviviales que dans les autres environnements. Cependant, la puissance de *WinDev* repose sur un langage propriétaire pour l'application, ainsi que sur une bibliothèque de présentation propriétaire sur laquelle l'environnement a donc un contrôle total. Les solutions proposées ne sont donc pas génériques. De plus, *WinDev* ne s'intéresse pas aux relations entre l'application et la présentation. La plupart des mises à jour de la présentation doivent donc encore être décrites manuellement dans le code de l'application.

En dehors de ces environnements, les outils de conception d'application, comme ceux reposant sur UML, proposent rarement des solutions pour concevoir l'interface. En effet, UML ne traite pas explicitement de l'interface utilisateur de l'application. Certains outils, par exemple *Together Control Center de TogetherSoft*, proposent des solutions identiques aux environnements précédents. Seule *Genova*, une extension de *Rational Rose*, propose une autre solution. Elle consiste à décrire la présentation par l'intermédiaire d'un *modèle* reposant sur la notion d'AIO (*Abstract Interaction Object*) proposée par (Vanderdonckt et al. 93). À partir d'un guide de style et d'un ensemble de classes sélectionnées par le concepteur, Genova génère une présentation possible de l'interface utilisateur. Même si l'utilisation d'AIO permet une description de la présentation indépendante des boîtes à outils et donc une génération de la présentation pour différents langages de programmation (en l'occurrence Java, C++, VB, HTML), les modèles obtenus ne permettent qu'une génération partielle de l'interface. Le prototype généré doit ensuite être modifié par les développeurs afin d'obtenir la présentation finale. De plus, *Genova* s'intéresse uniquement à la partie présentation et ne traite à aucun moment les relations entre la présentation et le noyau fonctionnel. Elle ne traite donc pas non plus le problème de la cohérence.

Il n'existe donc pas actuellement d'outils commerciaux capables de prendre en charge toute la chaîne de conception d'une interface utilisateur. Les solutions proposées se limitent à la partie présentation (et souvent à une boîte à outils en particulier), et l'aide à la description des relations entre le noyau fonctionnel et cette présentation est pratiquement inexistante. C'est donc le concepteur qui doit lui-même développer les mécanismes liés à la gestion de la cohérence.

Pour faciliter la conception des interfaces utilisateurs, des environnements indépendants des langages de programmation et des toolkits ont été proposés par la recherche. Nous allons présenter ici les approches basées sur l'utilisation de modèles déclaratifs.

1.2 – Les environnements à base de modèles

Le modèle ARCH (Bass et al. 92) est l'un des principaux modèles d'architecture pour systèmes interactifs. Il divise une application en cinq composantes représentées sous la forme d'une voûte (voir figure 1) :

- **le noyau fonctionnel :** il contient les fonctions et les données de l'application ;
- **l'adaptateur au noyau fonctionnel :** il joue le rôle de médiateur entre le contrôleur de dialogue et le noyau fonctionnel. Il permet l'exécution des fonctions de l'application et prépare les données de l'application pour l'interface.
- **le contrôleur de dialogue :** il est responsable des relations entre la présentation et le noyau fonctionnel. Il décrit les fonctions à appeler dans le noyau fonctionnel, les données à exposer, la navigation entre les différentes parties de l'interface, etc.
- **l'adaptateur à la présentation :** il joue le rôle de médiateur entre le contrôleur de dialogue et la boîte à outils ;
- **la boîte à outils (ou *toolkit*) :** il définit les différents éléments de présentation (également appelés *widgets)* d'une bibliothèque de présentation.

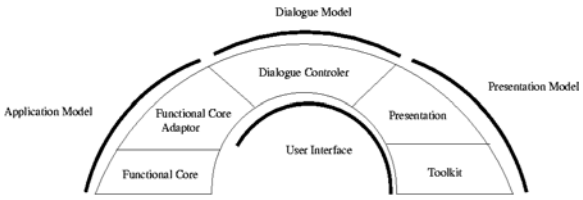

Figure 1. *ARCH et modèles d'interface*

De nombreuses démarches ont été proposées pour décrire ces cinq composantes (Myers et al. 00). Une des plus prometteuses est celle des environnements de développement appelés MB-UIDE (*Model-Based User Interface Development Environment*) (Silva 00). Ces environnements reposent sur l'utilisation d'un ensemble de modèles représentant de manière déclarative tous les aspects importants d'une interface, indépendamment des détails d'implémentation tels que les langages de programmation ou les boîtes à outils. L'interface utilisateur est ensuite générée automatiquement ou

semi-automatiquement. Les trois principaux modèles sont les suivants :

- **le modèle de l'application (ou du domaine)** décrit les fonctions et les données de l'application. Le choix du modèle dépend en partie de la nature du noyau fonctionnel pour lequel l'interface utilisateur est construite. Par exemple, Mastermind (Szekely et al 95) utilise une extension de l'IDL de Corba, tandis que Teallach (Griffiths et al. 01) utilise le modèle objet ODMG (Cattel et al. 00).
- l**e modèle de présentation** décrit les fonctionnalités des éléments de présentation proposés par les bibliothèques d'interface. Il permet également la description de nouveaux éléments à partir des éléments existants.
- l**e modèle de dialogue** décrit les liens entre le noyau fonctionnel de l'application et la présentation de l'interface. Il est construit à partir d'un modèle des tâches (Paternò 99) pour obtenir ensuite un modèle de dialogue opérationnel (Palanque 03). Le modèle des tâches décrit les tâches pouvant être exécutées par les utilisateurs, l'application, ou les deux. Une tâche peut être décomposée en sous-tâches, reliées par des contraintes temporelles telles que l'ordre d'exécution et le nombre d'itérations. Les tâches atomiques sont principalement des actions supportées par le noyau fonctionnel ou l'utilisateur.

Si les environnements à base de modèles permettent la description des interfaces indépendamment des détails d'implémentations, ces approches n'ont pas encore intégré de solution permettant de traiter de manière complètement automatique des problèmes récurrents telles que la gestion de la cohérence des données exposées par rapport aux données de l'application.

2 – COMMENT GÉRER LE PROBLÈME DE LA COHÉRENCE ?

Dans cette section nous définissons plus précisément le problème de la cohérence puis nous exposons deux techniques classiques pour gérer ce problème manuellement. Dans la section suivante nous montrons comment automatiser cette gestion.

2.1 – Le problème de la cohérence

Soit *s* une donnée de l'application qui doit être reflétée dans l'interface et *o* une donnée de la présentation qui doit refléter *s*. Il existe donc une fonction *f* telle que $o = f(s)$. Le problème de la cohérence consiste à garantir que cette égalité est maintenue.

Nous prenons aussi en compte le cas inverse où une donnée *s* de l'application dépend de *o* : il existe donc une fonction f telle que $s = f(o)$. Mais nous ne traitons pas ici du cas plus général où *s* et *o* peuvent tous les deux être modifiés « spontanément ».

Nous considérons de plus que le modèle de l'application décrit effectivement, éventuellement indirectement, la fonction *f*. Il est donc possible de recalculer *o* (ou *s*) en appelant des fonctions définies dans le modèle de l'application.

Autrement dit, la relation liant *o* à *s* est en pratique de la forme :

$$o = f^0\left(s_1^0, s_2^0, \ldots, s_i^0, \ldots, s_{n_0}^0\right)$$

avec $s_i^j = \begin{cases} \text{valeur constante} \\ \text{donnée de la présentation} \\ f^k\left(s_1^k, s_2^k, \ldots, s_i^k, \ldots, s_{n_k}^k\right) \text{où } 0 \le k < K \end{cases}$

Pour une fonction f^j, n_j correspond au nombre de paramètres de cette fonction et s_i^j correspond au paramètre d'indice *i* de cette fonction. K représente le nombre de fonctions intervenant dans la description de *s*.

Une contrainte satisfaite ne peut devenir insatisfaite que si un ou plusieurs des termes de la partie droite sont modifiés. Garantir la contrainte d'égalité consiste donc à réévaluer la partie droite quand un de ses sous-termes a changé.

Deux techniques sont souvent utilisées par les concepteurs d'interface. La première, qui correspond au *design pattern* observer (Gamma et al 94), consiste à enregistrer o auprès des données susceptibles d'être modifiées. Ces données « préviennent *o* » quand elles sont modifiées. La seconde technique consiste pour le concepteur d'interface à inférer lui-même, quand c'est possible, lorsqu'une donnée est modifiée. Typiquement le concepteur sait que l'appel de telle fonction avec tels arguments modifiera telle donnée.

2.2 – Le design pattern Observer

Un *design pattern* est une documentation du comportement d'un concepteur spécialisé décrivant un problème récurrent ainsi que sa solution. Le pattern *Observer* (voir figure 2) définit une dépendance entre un objet source (*subject*) vers plusieurs objets (*observers*), tel que lorsque l'objet source change d'état, toutes ses dépendances sont notifiées (*o* → *update*) pour une mise à jour (*subject* → *GetState*). Ce pattern s'applique lorsqu'il est nécessaire de changer d'autres objets quand l'objet source change d'état, ainsi que pour éviter un couplage fort entre un objet et ses observateurs.

L'utilisation du pattern Observer est donc intéressante pour garantir la cohérence des données exposées telle que nous l'avons définie dans la section précédente. En effet, si nous considérons que nos termes sont les données observées, les notifications sont alors les événements indiquant que le terme a été modifié.

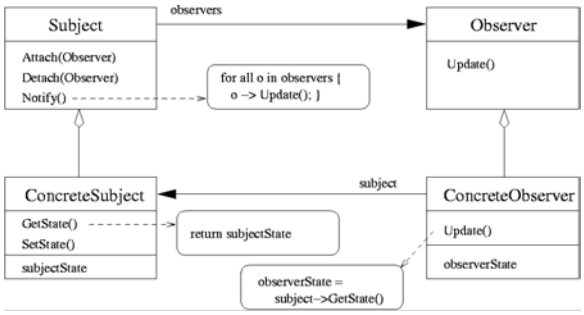

Figure 2. *Le design pattern Observer*.

Une application conçue avec le pattern *Observer* permet donc à l'interface de s'enregistrer auprès de l'application lorsqu'une donnée exposée dépend d'une donnée de l'application pour laquelle le pattern Observer est implémenté. Pendant la période d'exposition de la donnée, l'interface attend les notifications de l'application pour déterminer les termes modifiés et mettre à jour les données exposées. L'interface se désenregistre dès lors que la donnée n'est plus exposée ou qu'elle ne dépend plus de cette donnée de l'application.

2.3 – La détection d'effets de bord

Il est également possible d'identifier qu'une donnée de l'application a été modifiée en considérant que la plupart de ces modifications font suite à des appels de fonction provoquant des effets de bord (c'est-à-dire dans notre cas les fonctions modifiant des données de l'application).
Les événements indiquant qu'un terme est modifié sont dans ce cas les appels de fonction modifiant la donnée dans l'application.

Maintenant que nous savons comment gérer la cohérence des données exposées par rapport aux données de l'application, nous allons décrire comment ce mécanisme est intégré dans GOLIATH, notre environnement de conception.

3 – GOLIATH

GOLIATH a été conçu dans le but de faciliter la conception des interfaces utilisateur en général, et la description des relations entre le noyau fonctionnel et la présentation en particulier. Il est fondé sur une approche à base de modèles décrivant les différentes composantes de l'interface.
L'utilisation de connaissances est nécessaire à la gestion de ces modèles. Elles font le lien entre les modèles et l'implémentation et prennent en charge certaines tâches réalisées habituellement à la main par les concepteurs.

3.1 – Le modèle d'application

Le modèle d'application de GOLIATH est basé sur un langage plus simple que l'IDL utilisé par Mastermind. Notre modèle repose essentiellement sur la définition de types de données et de signatures de fonction.
Dans ce modèle, nous faisons la distinction entre les fonctions permettant d'obtenir des données de l'application et les fonctions dont l'exécution modifie les données de l'application, que nous appellerons fonctions à effets de bord et que nous identifierons par un '+' précédant leur déclaration.

Exemple 1 : La fonction *getContact* récupère un contact d'un carnet d'adresses. Cette fonction a deux arguments en entrée : le carnet d'adresses et la clé du contact ; elle retourne en sortie le contact correspondant.

```
getContact(IN AddressBook anAddressBook,
   IN ContactKey aContactKey,
   OUT Contact aContact)
```

Exemple 2 : La fonction *updateContact* met à jour un contact dans un carnet d'adresses. C'est une fonction à effets de bord car elle modifie le contenu du carnet ainsi que le contact.

```
+updateContact(
     IN AddressBook anAddressBook,
     IN ContactKey aContactKey,
     IN String newFirstName, ...)
```

Le modèle d'application peut être généré automatiquement en analysant le code source de l'application.

3.2 – Le modèle de présentation

Comme tous les modèles de présentation, le modèle de GOLIATH permet la description de nouveaux éléments de présentation à partir des éléments primitifs offerts par les boîtes à outils. Chaque élément de présentation (primitif ou composite) possède une partie interface qui définit les actions déclenchées, les données accessibles et les slots existant dans l'élément de présentation. Un slot définit un emplacement personnalisable au sein d'un élément de présentation. Il permet la définition de conteneurs (comme les fenêtres), ou de déléguer une partie du rendu à un élément externe.

Exemple 3 : Ci-dessous une partie de l'interface de l'élément de présentation permettant la mise à jour d'un contact. Cet élément définit notamment une action (*update*), déclenchée lorsque l'utilisateur confirme la mise à jour, et une donnée (*contact*), pour exposer et récupérer le contact à modifier.

actions = (update (…))
data = (contact :Contact (…))

Par ailleurs, chaque élément de présentation composite définit une liste de sous-éléments, un ensemble de liens entre son interface et l'interface

des sous-éléments, et un ensemble de comportements définissant des modifications dynamiques. Ces aspects sortent du cadre de cet article.
Le modèle de présentation peut être généré semi-automatiquement à partir du modèle de dialogue.

3.3 – Le modèle de dialogue

Un modèle de dialogue décrit les relations entre l'application et la présentation, et notamment l'exposition des données de l'application dans la présentation.
L'approche que nous proposons consiste à décomposer l'interface utilisateur en un ensemble de « conteneurs abstraits ». Chaque conteneur abstrait représente une partie indépendante de l'interface reliée à un élément de présentation. Il décrit 1) les signaux permettant de communiquer avec les autres conteneurs abstraits, 2) les opérations que l'utilisateur peut déclencher, 3) les vues exposant des données dans l'élément de présentation.

Les signaux de navigation servent principalement à décrire les relations entre les différents conteneurs abstraits. Certains signaux, comme l'activation ou la désactivation d'un conteneur abstrait, sont prédéfinis. De nouveaux signaux peuvent être définis par le concepteur en fonction des besoins de l'application.

Les opérations sont composées d'un appel de fonction ainsi que d'un ensemble de conditions décrivant leur déclenchement. Une condition correspond à : une action effectuée par l'utilisateur dans la présentation, la réception d'un signal de navigation, ou une signature de fonction et ses paramètres (l'opération est alors déclenchée si un appel à cette fonction est effectué). Une fois la fonction de l'opération exécutée, des signaux de navigation peuvent être envoyés.

Exemple 4 : L'opération updateContact décrit comment mettre à jour un contact dans le carnet d'adresses quand l'action *update* est déclenchée dans la présentation. *myAddressBook* et *contactKey* sont des variables locales du conteneur abstrait, tandis que *firstName* , *lastName*, … représentent des données décrites dans l'élément de présentation associé au conteneur :

operation updateContact (
  triggers = (update),
  function-call = updateContact(
    myAddressBook, contactKey,
    firstName, lastName, …),
  end-signals = ())

Les vues mettent en relation une donnée de l'application avec une donnée décrite dans l'élément de présentation associé au conteneur abstrait. Elles sont également composées d'un champ **update** décrivant les conditions entraînant la mise à jour de la vue. Ces conditions sont de la même nature que les conditions utilisées pour déclencher une opération. La gestion de la cohérence des données exposées dans la présentation consiste donc essentiellement à déterminer les conditions nécessaires à placer dans ce champ **update**.

Exemple 5 : Dans le conteneur abstrait exposant un contact d'un carnet, nous avons une vue sur un contact correspondant à une clé. Le contact retourné est associé à la donnée *contact* définie dans l'élément de présentation (voir exemple 3) afin d'être exposé à l'utilisateur.

view DisplayContact (
  update = (<*à determiner*>),
  data-term = aContact **from** getContact(
    myAddressBook, contactKey),
  pdata = (contact) …)

4 GESTION DE LA COHÉRENCE DES DONNÉES EXPOSÉES DANS GOLIATH

Nous avons vu à la section 2 que pour résoudre le problème de la cohérence, il s'agit de réévaluer la partie droite de la contrainte de cohérence $o = f(s)$ quand un est des sous-termes de s a été modifié.
Les conditions de modification peuvent être décrites par l'intermédiaire d'une déclaration de dépendance entre un terme et un événement E indiquant que ce terme est modifié :

$$\left.\begin{array}{c}\text{valeur constante}\\ \text{donnée de la présentation}\\ f(s_{i_1},\ldots,s_{i_n})\end{array}\right\} \text{dépend de } E(s_1,\ldots,s_n)$$

Autrement dit, une déclaration de dépendance décrit qu'un événement E, accompagné d'un ensemble de paramètres, indique soit la modification d'un terme atomique, soit la modification d'un terme portant sur une fonction f utilisant éventuellement un sous-ensemble des paramètres de E. Ces paramètres permettent d'identifier de manière plus ou moins précise quel terme a été modifié et donc quelle donnée doit être mise à jour dans la présentation. Il peut donc y avoir de « fausses alertes », à savoir qu'un événement nous amène à considérer qu'un terme est modifié alors que ce n'est pas le cas. Le nombre de ces fausses alertes diminue avec la précision des contraintes liant un événement à un terme. Cependant cette précision augmente le coût de la gestion de la cohérence. Il est donc nécessaire de moduler la précision des contraintes en fonction du contexte (nombre de données à exposer simultanément, fréquence des modifications, etc.)

pour ne pas saturer la présentation avec les mises à jour. Cet aspect n'est pas traité dans cet article.

4.1 Extension du modèle d'application

Le modèle d'application que nous avons décrit dans la section 3.2 est insuffisant pour décrire les informations nécessaires à la gestion de la cohérence.
Pour utiliser la technique du pattern *Observer*, nous enrichissons tout d'abord le modèle en identifiant les signatures correspondant à des notifications (par l'intermédiaire d'un '*'). Ensuite, nous définissons des relations permettant d'identifier les fonctions de (dés)enregistrement liées à une notification.

Exemple 6 : La modification d'un contact peut être signalée par la notification *contactUpdated*. Cette notification possède deux paramètres permettant d'identifier le contact concerné. La fonction *subscribeContactUpdated* permet de s'enregistrer auprès du noyau fonctionnel pour recevoir cette notification. À l'inverse, la fonction *unsubscribeContactUpdated* permet de se désenregistrer. Les déclarations **subscribes-to** (resp. **unsubscribes-from**) permettent d'identifier automatiquement comment s'enregistrer (resp. se désenregistrer) d'une notification.

```
*contactUpdated(
   OUT AddressBook anAddressBook,
   OUT ContactKey aContactKey)

subscribeContactUpdated(
   IN AddressBook anAddressBook,
   IN ContactKey aContactKey)
unsubscribeContactUpdated(
   IN AddressBook anAddressBook,
   IN ContactKey aContactKey)

subscribeContactUpdated(AB, CK)
   subscribes-to contactUpdated(AB, CK)
unsubscribeContactUpdated(AB, CK)
  unsubscribes-from contactUpdated(AB,CK)
```

Par ailleurs, pour identifier les données modifiées par un appel de fonction à effets de bord, ou dont la modification est signalée par la réception d'une notification, nous introduisions des déclarations de dépendance reliant les fonctions retournant les données de l'application avec les fonctions à effets de bord et les notifications.

Exemple 7 : La notification *contactUpdated* indique qu'un contact est mis à jour :

```
   contactUpdated(AB, CK)
impacts contact from getContact(AB, CK)
```

Exemple 8 : La fonction *updateContact* modifie le résultat de la fonction *getContact*, à condition que le carnet d'adresses et la clé utilisée pour récupérer le contact soient les mêmes que ceux utilisés pour la modification du contact. Le symbole '_' indique que la valeur associée à ce paramètre n'a pas d'importance.

```
   updateContact(AB, CK, _ , …)
impacts aContact from getContact(AB, CK)
```

Notons que les modifications qui ne sont pas dues à des appels de fonction et qui ne sont pas signalées par des notifications ne peuvent pas être détectées par cette technique. Ce cas de figure est pris en compte mais ne sera pas détaillé ici.

4.2 Transformation du modèle de dialogue

Maintenant que nous avons intégré les connaissances dans le modèle d'application, nous pouvons décrire les transformations qui sont appliquées automatiquement sur le modèle de dialogue. Le but de ces transformations est de produire un modèle permettant la génération d'une interface dont les vues sont mises à jour quand les données qu'elles exposent sont modifiées.
Une vue doit être mise à jour dans deux situations. D'une part lorsque la donnée obtenue par l'appel de fonction définie dans la vue est différente de la donnée obtenue lors d'un appel précédent. D'autre part lorsque les paramètres de l'appel de la fonction changent : ces modifications impliquent en effet qu'une autre donnée doit être exposée. Nous allons décrire les transformations à appliquer dans le premier cas. Pour le deuxième cas, il suffit de réitérer le processus pour chaque paramètre afin de déterminer ses conditions de modifications.

Prenons l'exemple d'un conteneur abstrait dont une vue est chargée d'exposer le contenu d'un contact. Le conteneur décrit par le concepteur est donc le suivant :

```
DisplayContainer (
 local-variables = (
  AddressBook myAddressBook,
  ContactKey contactKey)
 views = (
  view DisplayContact (
   update = (),
   data-term = contact from getContact(
     myAddressBook, contactKey),
   pdata = (contact) …)
 operations = (
  operation Close (
   triggers = ( close ),
   function-call = none,
   end-signals = ( unactivate )))   )
```

Nous allons maintenant voir les transformations nécessaires pour garantir que cette vue sera mise à jour si le contact exposé est modifié.

Cas d'un effet de bord

La technique liée à la détection d'effets de bord peut être choisie s'il existe une déclaration de dépendance reliant une fonction à effets de bord à la fonction *getContact*. Dans le cas de l'exemple 8, il s'agit de la fonction *updateContact*.

Grâce à la déclaration de dépendance, l'interface identifie pour quels paramètres de la fonction *updateContact* il sera nécessaire de mettre à jour la vue. En l'occurrence, la mise à jour est effectuée si le même carnet d'adresses et la même clé sont utilisés. Dans le cas de notre vue, il s'agit des variables locales *myAddressBook* et *contactKey*. Il suffit donc de modifier la vue en indiquant qu'elle est mise à jour lorsque la fonction *updateContact* est appelée avec ces paramètres :

```
view DisplayContact (
  update = (
    updateContact(
      myAddressBook, contactKey, _, …)
  ),
  data-term = aContact from getContact(
    myAddressBook, contactKey),
  pdata = (contact) …)
```

Cas du pattern *Observer*

La technique liée au pattern *Observer* peut être choisie s'il existe une déclaration de dépendance reliant une notification à la fonction *getContact*. Dans le cas de l'exemple 7, il s'agit de la notification *contactUpdated*. La démarche à suivre est la suivante :

1. Le système identifie la fonction permettant de s'enregistrer pour recevoir cette notification en cherchant dans le modèle d'application une relation de la forme « XXX subscribes-to contactUpdated(…) ». Le XXX trouvé (ici *subscribeContactUpdated*) correspond à la fonction d'enregistrement. Comme une donnée est exposée lorsque le conteneur abstrait de la vue est activé (réception du signal *activate*). Il suffit donc d'ajouter une opération dans le conteneur abstrait pour appeler la fonction d'enregistrement à la réception du signal *activate*.

La détermination des données à utiliser en paramètre de la fonction d'enregistrement s'effectue à partir de l'appel de fonction permettant de récupérer la donnée à exposer, et grâce aux contraintes sur les paramètres définies par la déclaration de dépendance et par l'identification de la fonction d'enregistrement.

```
operation DisplayContact_Subscribe (
  triggers = ( activate ),
  function-call = subscribeContactUpdated(
    myAddressBook, contactKey),
  end-signals = () )
```

2. La mise à jour de la donnée exposée est effectuée à chaque réception d'une notification *contactUpdated*. Les données attendues en paramètre de la notification sont déterminées à partir de l'appel retournant le contact et des contraintes définies dans la relation permettant d'identifier la fonction d'enregistrement.

```
view DisplayContact (
  update = (
    contactUpdated(myAddressBook,
                   contactKey)),
  data-term = aContact from getContact(
      myAddressBook, contactKey),
  pdata = (contact) …)
```

3. Une donnée n'est plus exposée lorsque le conteneur abstrait de la vue est désactivé (réception du signal *unactivate*). Il suffit donc d'ajouter une opération dans le conteneur abstrait pour se désenregistrer à la réception de ce signal. Le système identifie la fonction de désenregistrement et ses paramètres de la même manière que pour la fonction d'enregistrement.

```
operation DisplayContact_Unsubscribe (
  triggers = ( unactivate ),
  function-call =unsubscribeContactUpdated(
    myAddressBook, contactKey),
  end-signals = () )
```

5. RÉSULTAT

L'approche que nous venons de décrire a été implémentée dans un environnement baptisé GOLIATH, dont le noyau est implémenté en Caml (http://caml.inria.fr). GOLIATH a été testé avec des applications écrites en Caml et Java, ainsi qu'avec la boîte à outils Swing.

Cette approche a été validée par l'intermédiaire d'un certain nombre d'exemples comme la gestion d'un carnet d'adresses (voir figure 5). L'interface du carnet d'adresses repose sur quatre conteneurs abstraits : la fenêtre principale et les fenêtres de consultation, de modification et d'ajout. La fenêtre principale contient une vue sur la liste des contacts et celles de consultation/modification une vue sur un contact sélectionné dans la liste. Les fenêtres de modification/ajout permettent le déclenchement des fonctions updateContact / addContact.

À partir de cette description fournie par le concepteur, le modèle est complété par GOLIATH de manière à ajouter les informations permettant de mettre à jour automatiquement la fenêtre principale après chaque ajout/modification/suppression, ainsi que toutes les autres fenêtres de consultation affichant des contacts qui ont été modifiés ou supprimés.

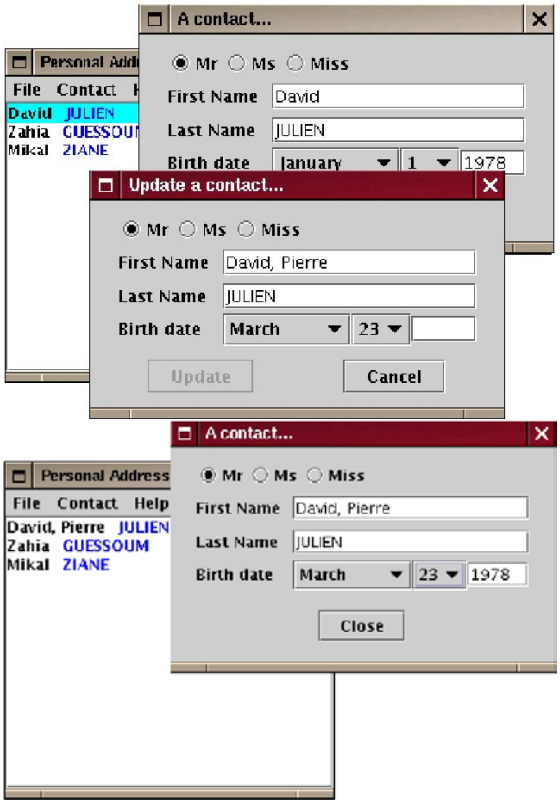

Figure 5. *En haut, l'interface avant modification. En bas, la même interface après mise à jour du contact. Les données ont été mises à jour automatiquement.*

6. CONCLUSIONS ET PERSPECTIVES

Dans cet article, nous avons montré comment deux techniques (le pattern *Observer* et l'identification des fonctions à effets de bord), utilisées habituellement à la main par les concepteurs d'interfaces utilisateur, pouvaient être utilisées automatiquement dans une démarche fondée sur des modèles. Notre environnement offre ainsi la possibilité de gérer automatiquement la cohérence de l'interface utilisateur avec les données de l'application. En effet, le concepteur se contente uniquement de décrire d'une part les données de l'application qu'il veut exposer et leur emplacement d'exposition, et d'autre part les fonctions de l'application à appeler.

Cette gestion de la cohérence peut encore être améliorée. Tout d'abord la précision des contraintes doit pouvoir être adaptée pendant l'exécution de manière à ne pas saturer la présentation avec des mises à jour lorsque de nombreuses données sont exposées ou lorsque la modification d'une donnée est trop fréquente. Ensuite nous devons également étudier le cas des vues bidirectionnelles, c'est-à-dire des vues où la donnée exposée peut être modifiée par l'utilisateur et par l'application.

D'autres connaissances ont également été intégrées au sein de GOLIATH. En utilisant un principe similaire à celui décrit dans cet article, GOLIATH détermine les actions autorisées ou non compte tenu de l'état de l'application. Il est ainsi capable de garantir automatiquement que l'utilisateur ne peut pas appeler des fonctions dont les pré-conditions ne sont pas vérifiées.

À moyen terme, la conception d'une interface avec GOLIATH consistera à définir le dialogue de l'interface. Le reste de l'interface sera généré semi-automatiquement via les connaissances disponibles dans les modèles. Ainsi, le concepteur pourra consacrer plus de temps à son rôle principal : la conception d'interfaces utilisables.

BIBLIOGRAPHIE

Bass L., Little R., Pellegrino R., Reed S., Seacord R., Sheppard S., Szczur M. R. (1992) *The UIMS Tool Developers' Workshop* : A Metamodel for the Runtime Architecture of an Interactive System. 24(1) :32-37.

Cattle R. G. G., Barry D. K., 2000, *The Object Databases Standard : ODMG 3.0*. Morgan Kaufmann Publishers.

Gamma E., Helm R., Johnson R., Vlissides J (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley.

Griffiths T., Barclay P. J., Paton N. W., McKirdy J., Kennedy J., Gray P. D., Coper R., Goble C. A., Da Silva P.P. (2001). Teallach: A Model-Based User Interface Development Environment for Object Databases. In Elsevier, editor, *Interacting With Computers*, volume 14, pp 31-68.

Julien D., Ziane M., Guessoum Z. (2004) GOLIATH: an Extensible Model-Based Environment to Develop User Interfaces. *In Proceedings of Computer Aided Design for User Interface IV (CADUI)*, Kluwer Academics Publishers, pp 95-104.

Myers B. A. (1995) User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64-103.

Myers B. A., Hudson S. E., Pausch R. (2000) Past, Present and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3-28.

Palanque, P., Bastide, R., Winckler, M. (2003) Automatic Generation of Interactive Systems: Why A Task Model is not Enough. In *Proceedings of Human-Computer Interaction Internationnal*, Heracklion, Crete.

Paternò F. (1999) *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag.

Pinheiro da Silva P. (2000) User Interface Declarative Models and Development Environments: A Survey. In Ph. Palanque and F. Paternò, editors. *Proceedings of DSV-IS2000*, volume 1946 of LNCS, pp 207-226, Limerick, Ireland.

Szekely P. A., Sukaviriya P., Castells P., Muthukumarasamy J., Salcher E. (1995) Declarative interface models for user interface construction tools: the Mastermind approach. *In Proceedings EHCI'95*, pp 120-150.

Vanderdonckt J., et Bodart, F. (1993) Encapsulating knowledge for intelligent automatic interaction objects selection. *In Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, pp 424-429.