SPECIAL ISSUE OF THE 8TH MCSEAI'04
May 09 – 12 – 2004, Sousse - Tunisia

It is our great pleasure to welcome you to read this special issue of the eighth *Maghrebian Conference on Software Engineering and Artificial Intelligence* MCSEAI'04 that was held of the 9 to the 12 may 2004 in Sousse-Tunisia. This eighth edition intends to be a forum for the most recent research on the engineering of advanced information systems. Web and multimedia applications, natural language processing, as well as distributed intelligent systems.

As you will note, the 11 selected papers out of 51 presented (i.e., a selection rate of about 20%) have a high quality and they are agreeable to read.

These papers will highlight recent advances and solutions to problems raised by research in the domain of software engineering and artificial intelligence. Their authors are eminent specialists in the domains of information retrieval and indexing, and advanced design of multimedia applications. Many postgraduate students will find their answers to various, interesting question.

We would like to express our gratitude to the guest editor, the members of the scientific committee, to the additional reviewers and the session's chairmen of their precious collaboration:

Guest editor

Abdelmajid Ben Hamadou	***(ISIM, Sfax, Tunisie)***

Scientific committee

Guillaume Auriol	(LAAS, Toulouse, France)
Béchir Ayeb	(FS, Monastir, Tunisie)
Abdelmajid Ben Hamadou	(ISIM, Sfax, Tunisie)
Rafik Bouaziz	(FSEG, Sfax, Tunisie)
Faouzi Boufares	(LIPN, France)
Walid Mehdi	(ISIM, Sfax, Tunisie)
Mohamed Moncef Ben Khelifa	(STIC/AI Toulon, France)
Hadj Kacem Ahmed	(FSEG, Sfax, Tunisie)
Mohamed Mosbah	(LaBRI, Bordeaux, France)
Mohamed Jmaiel	(ENI, Sfax, Tunisie)
Faiez Gargouri	(ISIM, Sfax, Tunisie)
Hanene Ben Abdallah	(FSEG, Sfax, Tunisie)
Jamel Feki	(FSEG, Sfax, Tunisie)
Abdelaziz Abdellatif	(FS, Tunis, Tunisie)
Chahir Youssef	(GREYC, Caen, France)
Khalil Drira	(LAAS, Toulouse, France)
Zaia Alimazighi	(USTHB, Algerie)
Mounir Boukadoum	(U. Quebec, Canada)
Khaled Ghedira	(ENSI, Tunisie)
Mokhtar Sellami	(U. Badji Mokhtar, Allemagne)

Finally, we wish you a very good reading.

The editorial committee of ISDM.

UNE APPROCHE FORMELLE POUR L'INTEGRATION DES ASPECTS STRUCTURAUX ET COMPORTEMENTAUX DE REPRESENTATIONS CONCEPTUELLES

Nahla Haddar
Ingénieur en informatique
Nahla.Haddar@fsegs.rnu.tn

Faîez Gargouri
Maître de conférences à l'Institut Supérieur d'Informatique et de Multimédia de Sfax, Tunisie
Faiez.Gargouri@fsegs.rnu.tn

Abdelmajid Ben Hamadou
Professeur à l'Institut Supérieur d'Informatique et de Multimédia de Sfax, Tunisie
Abdelmajid.Benhamadou@isimsf.rnu.tn

Résumé : Dans cet article, nous présentons une approche formelle pour l'intégration de représentations conceptuelles et nous décrivons les différentes étapes du processus d'intégration. L'originalité de notre travail provient du fait que nous avons traité le problème d'intégration à l'aide d'un langage de spécification formel qui est l'Object-Z et que nous avons inclus aussi bien les aspects structuraux que ceux comportementaux des représentations conceptuelles dans le processus d'intégration.

Summary : In this paper, we present a formal approach for conceptual representations integration and we describe the different steps of the integration process. The originality of our approach comes from the fact that we have treated the integration problem using a formal specification language, Object-Z and from the inclusion of structural as well as behavioural aspects of conceptual representations in the integration process.

Mots clés : intégration conceptuelle, transformation, comparaison, règles d'intégration.

Une approche formelle pour l'intégration des aspects structuraux et comportementaux de représentations conceptuelles

Aujourd'hui, les évolutions incessantes de l'entreprise et de son environnement ont eu un impact sur son système d'information. Pour réussir ces évolutions, l'entreprise doit mettre en œuvre un système d'information plus complexe et plus intégré, traitant des informations différentes et en plus grande quantité. L'augmentation de la complexité et de la taille d'un SI le rend difficile à développer.

Face à la complexification des systèmes d'information, la réaction de la communauté informatique consiste à développer de nouvelles démarches et de nouveaux outils de conception. Ainsi ces dernières années plusieurs approches ont été proposées dans le but d'appréhender cette complexité. Parmi ces approches nous pouvons citer la conception distribuée de Rational (2003), la réutilisation conceptuelle Semmak (1998), les méthodes situationnelles Harmsen (1997), etc. Les outils de modélisation ont aussi évolué pour mettre en œuvre les approches déjà citées et dont on pense pouvoir améliorer la productivité des concepteurs et optimiser le temps de développement. En effet, les recherches actuelles se concentrent sur le développement d'outils coopératifs et souples dans le sens où ils supportent plusieurs méthodes et permettent l'emploi de plusieurs d'entre elles à la fois dans le processus de développement. Ces outils doivent permettre aussi le partage de la tâche de conception entre plusieurs groupes de concepteurs.

Le développement d'un outil proposant une ou plusieurs de ces facilités nous conduit à considérer le problème d'avoir un ensemble de représentations conceptuelles (RCs) exprimées éventuellement dans des formalismes différents (entité/association, orienté objet, etc.) et que nous devons intégrer en une seule représentation conceptuelle globale.

Dans ce contexte, notre approche d'intégration conceptuelle propose un processus d'intégration de RCs traitant les aspects structuraux et comportementaux des objets représentés. Cette approche est formelle et basée sur le langage de spécification formel Object-Z. Son originalité provient du fait que nous avons traité le problème d'intégration à l'aide d'un langage de spécification formel et que nous avons inclus aussi bien les aspects structuraux que ceux comportementaux des représentations conceptuelles dans le processus d'intégration.

Notre article est composé des sections suivantes : Dans la première section nous décrivons les différentes étapes du processus d'intégration. Chacune de ses étapes sera détaillée dans les trois sections qui suivent. Enfin dans la dernière section nous présentons nos conclusions et perspectives.

1 - APPROCHE D'INTEGRATION

Notre approche pour assembler plusieurs RCs est basée sur une démarche définie en trois étapes. Ces dernières sont déduites des travaux de recherche réalisés dans le domaine des bases de données, comme les étapes proposées dans Parent et Spaccapietra (1998), par exemple. Nous décomposons donc le processus d'intégration en trois phases : la traduction des RCs initiales, leur comparaison et l'intégration proprement dite Gargouri, Haddar et Ben Hamadou (2000). En premier lieu, les RCs initiales sont traduites dans un formalisme (ou modèle) commun (MC).Les RCs exprimées dans le MC peuvent avoir des éléments en commun ou sémantiquement différents mais ayant des représentations semblables. Elles peuvent aussi contenir des éléments sémantiquement identiques mais n'ayant pas la même représentation. La deuxième phase consiste alors à comparer les RCs afin de détecter leurs correspondances et les conflits possibles. Enfin, dans la troisième phase les conflits sont résolus et les RCs seront intégrées pour construire la RC globale qui rassemblera tous les éléments des RCs dans une seule représentation. La figure 1 illustre cette démarche.

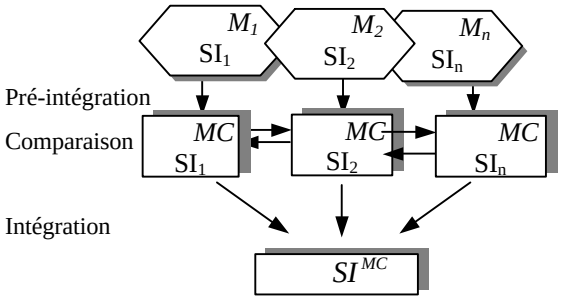


Figure 1: Etapes de l'intégration conceptuelle

Les sections suivantes explicitent les trois phases du processus d'intégration.

2 - ETAPE DE PRE-INTEGRATION

L'étape de pré-intégration consiste à transformer les RCs à intégrer dans un MC. Le MC que nous avons choisi est le langage UML. Ce choix est dû au fait qu'UML est un langage standard. De plus, UML est supporté par un grand nombre d'AGL et de méta-outils de conception. Pour réaliser la transformation

des RCs dans le MC UML, nous avons défini un cadre général qui permet l'expression de règles de mapping assurant le passage d'un modèle source, décrit dans un formalisme F1, dans un modèle cible équivalent exprimé dans un autre formalisme F2. La transformation entre modèles doit être définie au niveau des formalismes (i.e. au niveau des méta-modèles) puis appliquée au niveau des modèles pour qu'elle soit générale et facile à automatiser. Nous partons donc de deux méta-modèles F1 et F2 exprimés dans le langage standard de représentation de méta-modèles MOF de l'OMG (2001) pour définir une démarche qui aboutit à l'élaboration de règles de transformation entre F1 et F2. La figure 2 illustre notre cadre de transformation.

Les règles de transformation peuvent être exprimées de plusieurs façons différentes. Toutefois, il est important qu'elles soient définies à l'aide d'un langage de spécification formel car une définition formelle est toujours précise et non ambiguë. Le langage qu'il faut choisir doit supporter tous les concepts orientés objet tels que l'objet, l'identité d'objet, le comportement d'objet, la classe, l'héritage, la composition et le polymorphisme. Un très grand spectre de langages formels supporte ces concepts. Nous choisissons l'Object-Z car, d'une part, il offre les concepts orientés objet que nous avons déjà cités et qui permettent d'alléger et de simplifier la spécification, et d'autre part, il constitue une extension du langage Z, l'un des langages formels les plus appropriés pour la spécification des SI André et Vailly (2001). Une étape intermédiaire avant de spécifier les règles de transformation est nécessaire. Elle consiste à formaliser les modèles MOF de F1 et de F2 avec Object-Z. Ainsi, le processus de transformation devient aussi formel que possible. Sans cette formalisation l'approche de transformation devient difficile à réaliser.

Etant donné que les concepts utilisés par toutes les méthodes pour décrire leurs produits sont issus d'un nombre limité de formalismes, il est possible de regrouper ensemble les descriptions MOF de ces formalismes et de les formaliser à l'aide d'Object-Z. La spécification formelle en Object-Z d'un formalisme exprimé à l'aide de MOF peut être automatisée. Il s'agit de définir des règles de mapping entre le méta-modèle de MOF (exprimé lui-même en MOF) et celui de Object-Z et de les appliquer aux méta-modèles, pour obtenir une spécification formelle de ceux-ci. Les travaux de Kim et Carrigton (2000) sur le mapping formel entre les diagrammes de classes UML et les spécifications Object-Z montrent la faisabilité de cette formalisation. En effet, les deux auteurs donnent dans Kim et Carrigton (2000) des spécifications Object-Z du méta-modèle du diagramme de classes UML et du méta-modèle décrivant les concepts du langage Object-Z et les liens entre eux. Puis ils établissent des règles de transformation d'un diagramme de classes en une spécification Object-Z.

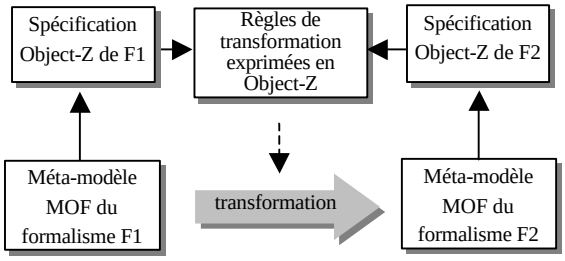


Figure 2: Cadre pour la transformation de RCs

Dans ce qui suit nous nous basons sur les spécifications formelles suivantes d'une classe UML et d'un diagramme de classes:

È*UMLClasse* _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®*ModelElement*
®È _
®®*diagramme* : *DiagrammeDeClasses*
®®*dEtats* : *DiagrammeDEtats*
®Ç _
®®A*a1*, *a2* : *attributs* ¥ *a1.nom* = *a2.nom*
®® Þ *a1* = *a2*
®®A*op1*, *op2* : *opérations* ¥ (*op1.nom* = *op2.nom*
®® ¦ #*op1.paramètres* = # *op2.paramètres*
®® ¦ A*i* : 1.. #*op1.paramètres* ¥
®® *op1.paramètres*(*i*).*type*=
®® *op2.paramètres*(*i*).*type*) Þ *op1* = *op2*
®®*self* e *diagramme.classes*
®Ð _
Ð _

L'attribut *diagramme* désigne le diagramme de classes qui contient la classe alors que l'attribut *dEtats* représente le diagramme d'états de la classe. Les invariants dans ce schéma signifient que les noms d'attributs dans une classe doivent être différents et que les opérations doivent avoir des signatures différentes.

È*DiagrammeDeClasses* _ _ _ _ _ _ _ _ _ _ _ _ _
®È _
®®*classes* : Fs*UMLClasse*
®®*assoc* : F*UMLAssociation*
®®*classesAssoc* : F*UMLClassAssoc*
®®*gen* : F*UMLGénéralisation*
®Ç _
®®A *c1*, *c2* : *classes* ¥ c1.*nom* = *c2.nom*
®® Þ *c1* = *c2*
®®U {*a* : *assoc* ¥ {*i*: *1..#a.pattes* ¥
®® (*a i*).*classeAttachée*}} z *classes*
®®U {*g* : *gen* ¥ {g.super, *g.sous*}} z *classes*
®®*classesAssoc* z *classes*
®Ð _
Ð _

Un diagramme de classes est caractérisé par ses classes, ses associations, ses classes d'association et ses liens de généralisation/spécialisation. Les

contraintes expriment que les classes impliquées dans une association, une classe d'association ou une généralisation doivent être des classes du diagramme.

3 - ETAPE DE COMPARIASON

Dans notre approche de conception par intégration, les RCs que nous voulons intégrer sont généralement construites indépendamment les unes des autres, mais elles font partie d'un même domaine. De ce fait, nous pouvons trouver des éléments (i.e. classes, attributs, méthodes) qui se répètent dans différentes représentations ou d'apercevoir des informations communes, mais qui sont modélisées de façons différentes. En plus, il est possible de rencontrer des éléments qui portent le même nom, mais qui sont sémantiquement différents. Pour que l'intégration de RCs puisse réussir, il faut que les similarités et les conflits entre les éléments de ces RCs soient détectés et résolus. Ainsi, une étude approfondie de la sémantique des RCs permet de les comparer et d'établir des correspondances entre leurs parties communes pour retrouver celles présentant des conflits.
Afin de déterminer les ressemblances sémantiques possibles entre les éléments des différentes RCs, nous devons nous intéresser non seulement à leur niveau statique mais aussi à celui dynamique. La prise en compte de ces deux aspects ensemble enrichie nos connaissances sur la sémantique des objets représentés. Ainsi, nous considérons à côté des diagrammes de classes les diagrammes d'états. Un diagramme de classes décrit des classes, les liens statiques entre elles, la structure de leurs objets et les messages qu'elles peuvent s'échanger. Tandis qu'un diagramme d'états d'une classe montre les états que peuvent prendre les objets de cette classe suite à l'invocation de ses méthodes. Ces réflexions nous conduisent à définir la sémantique du monde réel (RSW) d'une classe comme étant l'ensemble de tous les objets du monde réel dont les propriétés et le comportement sont représentés par cette classe. En se basant sur cette définition, nous nous proposons de déduire automatiquement les liens linguistiques entre les noms des éléments des diagrammes, les propriétés structurelles communes aux classes des différents modèles et les ressemblances entre les comportements de leurs objets. Il s'agit donc d'aller au delà des simples représentations pour capter leur sémantique. Plusieurs critères sont nécessaires : le critère linguistique, lié aux noms des éléments, le critère structurel, lié à la structure des objets des classes et le critère dynamique, lié au comportement de ces objets. L'analyse et la comparaison des RCs sont basées sur ces trois critères.

3.1 - Correspondances linguistiques

Les termes employés dans un domaine pour désigner les éléments de RCs sont la source de plusieurs conflits (confits de nom, conflits sémantiques, …). La détection de ces conflits ne peut être réalisée que si nous avons une description des termes et des concepts du domaine et des relations qui peuvent exister entre eux indépendamment de toute RC. En d'autres termes, il est nécessaire d'avoir une conceptualisation de la terminologie du domaine appelé aussi ontologie, permettant d'en déduire automatiquement les correspondances linguistiques entre les termes.
La spécification formelle de l'ontologie que nous proposons est exprimée en Object-Z. Elle définit une représentation des constituants de l'ontologie qui sont les termes, les concepts et des liens entre eux. Nous distinguons deux types de lien : des liens entre les concepts et des liens entre les concepts et les termes. Les concepts sont reliés par des liens conceptuels traduisant des liens sémantiques comme la synonymie, l'homonymie, l'hypernymie, etc.. Des liens d'interprétation entre termes et concepts permettent d'exprimer le sens des termes.
Afin de spécifier notre ontologie en Object-Z, nous définissons un type libre que nous appelons *[Chaîne]* à partir duquel toutes les chaînes de caractères sont spécifiées.
L'ontologie est conçue pour couvrir les concepts et les termes d'un domaine. Pour optimiser la fonction de recherche de concepts ou de termes nous distinguons plusieurs sous-domaines d'un domaine. Chacun d'eux possède des concepts spécifiques.

```
È_ Ontologie_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®È_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®®sousDomaine : P SousDomaine
®®concept : P Concept
®®terme : P Terme
®Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®®A sd : sousDomaine ¥ sd.concept z concept
®®A c : concept ¥ c.super z concept
®®A c : concept ¥ c.sous z concept
®®A c : concept ¥ c.composant z concept
®®A c : concept ¥ c.composé z concept
®®A c : concept ¥ c.terme z terme
®®A t : terme ¥ t.concept z concept
®Đ_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Đ_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Pour exprimer les liens sémantiques entre les termes utilisés dans un domaine, nous définissons les fonctions *identique*, *synonyme*, *gen_spec* et *composition* qui testent respectivement si deux termes sont identiques, synonymes, ou s'il existe un lien de généralisation/spécialisation ou de composition entre eux. La fonction *Homonyme* vérifie si un terme peut être utilisé pour désigner des concepts totalement différents.

```
®identique : Terme x Terme f B
Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®A t1, t2 : Terme ¥ identique (t1, t2)
®                    Û t1.nom = t2.nom
```

Deux termes sont synonymes s'ils désignent un même concept. La fonction *synonyme* vérifie donc l'existence d'une identité sémantique entre les termes.

® *synonyme* : *Terme* x *Terme* f **B**
Ç_ _
®A *t1, t2* : *Terme* ¥ *synonyme* (*t1, t2*)
® Û ! *identique*(*t1, t2*) ¦ ∃ *c*: *Concept* ¥
® *c*e *t1.concept* ¦ *c* e *t2.concept*

Un terme peut être utilisé de façon homonyme s'il existe deux concepts différents désignés par ce terme.

® *homonyme* : *Terme* f **B**
Ç_ _
®A *t* : *Terme* ¥ *homonyme* (*t*) Û
® ∃ *c1, c2*: *Concept* ¥ *c1* ë *c2* ¦ *c1*e *t.concept*
® ¦ *c2* e *t.concept*

Deux termes sont liés par un lien de composition si les concepts qu'ils désignent le sont aussi.

® *composition* : *Terme* x *Terme* f **B**
Ç_ _
®A *t1, t2* : *Terme* ¥ *composition* (*t1, t2*) Û
® ∃ *c1, c2*: *Concept* ¥ *c1* e *c2.composants* ¦
® *t1*e *c1.désignation* ¦ *t2* e *c2.désignation*

De même, la fonction *gen_spec*, appliquée à deux termes donne la valeur vrai si ces deux termes désignent des concepts liés par un lien d'hypernymie. Cette fonction teste si un terme t1 est un nom générique par rapport à t2 qui est un terme spécifique.

® *gen_spec* : *Terme* x *Terme* f **B**
Ç_ _
®A *t1, t2* : *Terme* ¥ *gen_spec* (*t1, t2*) Û
® ∃ *c1, c2*: *Concept* ¥ *c1* e *c2.super* ¦
® *t1*e *c1.désignation* ¦ *t2* e *c2.désignation*

Exemples :
Soient R1 et R2 deux représentations conceptuelles. Nous avons :
synonyme(R1.Individu, R2.Personne)
composition(R1.voiture, R2.moteur)
gen_spec(R1.Personne, R2.Employé)

3.2 - Correspondances structurelles

L'étude de la structure d'une classe repose sur la séparation entre sa structure locale et sa structure globale. La structure locale d'une classe est définie, par l'ensemble de ses attributs. Elle consiste donc en la description des propriétés qui sont propres à ses objets. Au contraire, la structure globale de la classe est décrite par l'ensemble des liens d'agrégation et de composition qui lient la classe à chacun de ses agrégats (et composants). La structure globale d'une classe se définit ainsi comme un assemblage d'agrégats (et de composants).
Cette section présente des métriques permettant de déterminer les correspondances entre deux classes aussi bien au niveau de leurs structures locales (au niveau de leurs attributs) qu'au niveau de leurs structures globales.

Comparaison des attributs de classes
La comparaison entre attributs de classes permet la détection de liens sémantiques entre ces classes. En effet, deux classes qui ont plusieurs attributs en commun doivent avoir un lien sémantique entre elles. La correspondance entre attributs de classes est déterminée par comparaison des ensembles d'attributs des classes.
Etant donné qu'une classe peut avoir des liens de généralisation /spécialisation avec d'autres classes, l'ensemble de tous ses attributs, qui forme sa structure locale, ne se limite pas à ceux qui sont spécifiques à la classe, mais inclue aussi ceux qui sont hérités. Il est donc nécessaire, avant de calculer la correspondance entre les attributs de deux classes, de chercher les ensembles complets de leurs attributs. Afin de formaliser cela, nous définissons une fonction Object-Z sur les classes d'un diagramme de classes qui renvoie l'ensemble de toutes les superclasses d'une classe UML du diagramme.

®*parents : UMLClasse* f P *UMLClasse*
®*ancêtres : UMLClasse* f P *UMLClasse*
Ç_ _
®A*c : UMLClasse* ¥
® *parents*(*c*) = {*g : c.diagramme.gen* | *g.sub=c* ¥
® *g.super*}
® *ancêtres* (*c*) = *parents* (*c*) U (U {*x : parents*(*c*) ¥
® *ancêtres* (*x*)})

Cette fonction nous permet de retrouver tous les attributs d'une classe y compris ceux qui sont hérités des superclasses.

® *allAttributs : UMLClasse* f P *UMLAttribut*
Ç_ _
®A *c : UMLClasse* ¥ *allAttributs (c)=*
® *c.attributs U (U {x : ancêtres(c)* ¥ *x.attributs})*

La comparaison des attributs de deux classes consiste à considérer les attributs de l'une d'elles et de chercher pour chacun d'entre eux si son nom est sémantiquement lié à un attribut de l'autre classe. A chaque correspondance trouvée, nous attribuons une pondération. Les valeurs des pondérations dépendent du lien sémantique qui existe entre les noms des attributs. Ainsi, nous avons cinq pondérations possibles :

p_0 = 0 : attribuée aux attributs dont les noms n'ont aucun lien sémantique entre eux,

p_1 : attribuée à deux attributs *a* et *b* tel que gen-spec(*a*.nom, *b*.nom),

p_2 : octroyée à *a* et *b* tel que composition (a.nom, b.nom),

p_3 : accordée à *a* et *b* tel que synonyme(*a*.nom, *b*.nom) et

p_4 = 1 : affectée à *a* et *b* tel que identique (*a*.nom, *b*.nom).

Les valeurs des poids p_1, p_2 et p_3 sont choisies par l'utilisateur en respectant toutefois les contraintes suivants :

$\forall i, j \in \{0, 1, .., 4\} \mid i \neq j \Rightarrow p_i \neq p_j$.

$0 = p_0 < p_1 < p_2 < p_3 < p_4 = 1$.

Une fonction p_a fournit la pondération à attribuer à un couple d'attributs suivant le lien sémantique entre leurs noms.

Certains termes sont fréquemment employés dans les représentations conceptuelles pour désigner des attributs de classes. C'est le cas, par exemple, des mots *nom*, *désignation* ou *libellé*. La comparaison de ces attributs n'a pas un grand intérêt, car ceux-ci ne sont pas très significatifs. Nous regroupons ces mots dans un même ensemble que nous appelons *ensemble des attributs non significatifs A*. Et dans le processus de comparaison, nous vérifions si les attributs à comparer sont significatifs ou non et nous en tenons compte dans l'attribution des pondérations. Ainsi, toute pondération concernant des attributs dont les noms sont des mots de l'ensemble *A* sera multipliée par un coefficient α_p ($0<\alpha_p<1$) appelé *coefficient de pertinence*. La fonction *p* suivante précise dans quel cas il faut appliquer ce coefficient.

p : Terme x Terme → {a_p, 1}

∀ a, b : Terme • p (a, b) = 1 ⇔ a ∉ A ∨ b ∉ A
∀ a, b : Terme • p (a, b) = a_p ⇔ a ∈ A ∧ b ∈ A

Lorsqu'une correspondance de noms entre deux attributs existe, nous comparons leurs types pour vérifier s'ils sont compatibles. Deux types sont compatibles s'il y a un lien d'inclusion entre eux. Toute pondération différente de 0 concernant des attributs de type incompatible sera multipliée par un coefficient α_c positif et inférieur à 1 appelé *coefficient de compatibilité*. La fonction *c* suivante teste si deux types sont compatibles ou non.

c : Terme x Terme → {a_c, 1}

∀ s, t : Terme • c (s, t) = 1 ⇔ (s,t) ∈ C
∀ s, t : Terme • c (s, t) = a_c ⇔ (s,t) ∉ C

Soient deux ensembles d'attributs X et Y tels que la cardinalité de X est inférieure à celle de Y. Nous comparons chaque attribut de X à tous ceux de Y et nous retenons le maximum des pondérations que nous pouvons obtenir. Ensuite, nous calculons la somme $\sigma_a(X, Y)$ de toutes les pondérations obtenues. La fonction σ_a est définie en Object-Z comme suit :

s_a: ℙ UMLAttribut × ℙ UMLAttribut → ℝ
∀ X: ℙ UMLAttribut • s_a(0, X) = 0 ∧ s_a(X, 0) = 0
∀ X, Y: ℙ UMLAttribut | X ≠ ∅ ∧ Y ≠ ∅ ∧
#X ≠ #Y • (∃ x : X • s_a (X, Y) =
max {y : Y • p(x.nom, y.nom) c(x.type.nom,*
y.type.nom) p_a(x.nom, y.nom)} + s_a (X \{x}, Y))*
∀ X, Y: ℙ UMLAttribut | X ≠ ∅ ∧ Y ≠ ∅ ∧
#X > #Y • s_a (X, Y) = s_a (Y, X)

Comparaison des structures globales

Dans cette section, nous considérons la structure globale des classes et nous définissons les fonctions qui permettent d'établir des correspondances entre elles.

La recherche des correspondances entre les structures globales de deux classes consiste à comparer les structures de leurs objets, et à déterminer la relation qui peut exister entre elles. Pour ce faire, il est nécessaire de retrouver, pour chacune de ces classes, toutes celles qui y sont reliées par un lien d'agrégation ou de composition. Ainsi, nous définissons une fonction Object-Z qui permet de rechercher pour un diagramme de classes donné et une classe de ce diagramme, toutes les classes ayant un lien d'agrégation ou de composition avec cette classe. Cette fonction est définie comme suit :

composants : UMLClasse → ℙ (UMLClasse×
TypeAgrégation ×UMLCardinalité)
∀ c : UMLClasse • composants (c) =
{x : c.diagramme.classes; m: UMLCardinalité;
t : TypeAgrégation | ∃ a: c.diagramme.assoc •
#a.pattes = 2 ∧
a.pattes(1).classeAttachée = c ∧
a.pattes(2).classeAttachées = x ∧
a.pattes(1).agrégation ≠ aucune ∧
t = a.pattes(1). agrégation ∧
a.pattes(2).cardinalité = m }

Chaque composant est un triplet formé par la classe composant, le type de lien qui la relie à la classe composé (agrégation ou composition) et la cardinalité qui se trouve du côté du composant.

La structure des objets d'une classe peut être définit uniquement par la classe elle-même ou, si la classe possède des superclasses, la définition de cette structure est déduite de la classe et de ces classes ancêtres. La fonction suivante permet de retrouver tous les composants d'une classe appartenant à une hiérarchie d'héritage.

allComposants : UMLClasse → ℙ (UMLClasse×
TypeAgrégation ×P_1ℕ)
∀ c : UMLClasse • allComposants (c) =
composant (c) ∪ (∪{x : ancêtre(D, c)•

®*composants(x)})*

Les ensembles de composants des deux classes à comparer ayant été trouvés, il s'agit ensuite de les comparer et de calculer le degré de similitude entre eux. Pour ce faire, nous comparons, pour une classe, chaque triplet représentant un composant à tous ceux de l'autre classe et une pondération est calculée. Puis le maximum de toutes ces pondérations est retenu pour notre triplet. Pour chaque couple de triplet, la pondération est calculée comme suit: à tout lien linguistique entre les noms des classes nous attribuons une pondération dont la valeur est renvoyée par la fonction π_a définie précédemment. Cette pondération est multipliée par un coefficient a_g si les types de liens dans les deux triplets ne sont pas les mêmes. Le produit peut être aussi multiplié par un deuxième coefficient a_m dans le cas où les cardinalités figurant dans les deux triplets ne coïncident pas.
Afin de formaliser la fonction de calcul de la somme des pondérations, nous donnons d'abord les fonctions *g* et *m* qui définissent les conditions dans lesquelles les coefficients respectifs α_g et α_m doivent être appliqués.

® *g : TypeAgrégation* x *TypeAgrégation* f *{a_g, 1}*
Ç_ _
®A *a, b : TypeAgrégation* ¥ *g (a, b) = 1* Û *a = b*
®A *a, b : TypeAgrégation* ¥ *g (a, b) =* a_g Û *a* ë *b*

®*m*:*UMLCardinalité* x *UMLCardinalité* f {a_m, 1}
Ç_ _
®A *s, t : UMLCardinalité* ¥ *m (s, t) = 1* Û
® *s.minimale = t.minimale* ¦
® *s.maximale = t.maximale*
®A *s, t : UMLCardinalité* ¥ *m (s, t) =* a_m Û
® *s.minimale* ë *t.minimale* v
® *s.maximale* ë *t.maximale*

Ensuite, la fonction s_{gl} ci-dessous calcule la somme des pondérations entre les triplets.

®s_{gl} : P *((UMLClasse*x *TypeAgrégation* x
® *UMLCardinalité)* d2) f R
Ç_ _
®A *X:* P *(UMLClasse*x *TypeAgrégation* x
® *UMLCardinalité)* ¥ s_{gl}(0, *X*) = 0 ¦ s_{gl}(*X*, 0) = 0
®A *X, Y: P (UMLClasse*x *TypeAgrégation x*
® *UMLCardinalité)* | *X* ë *0* ¦*Y*ë *0* ¦ *#X* ø *#Y* ¥
® (E *x: X* ¥ s_a *(X, Y) = max {y : Y* ¥
® p_a*(x(1).nom, y(1).nom) * g(x(2), y(2)) **
® *m(x(3), y(3)) }+* s_{gl} *(X \{x}, Y))*
®A *X, Y: P (UMLClasse*x *TypeAgrégation x*
® *UMLCardinalité)* | *X* ë *0* ¦*Y*ë *0* ¦
® *#X* > *#Y*¥ s_{gl}*(X, Y)* = s_{gl}*(Y, X)*

Rapport sémantique de deux classes
Pour mesurer le lien sémantique entre deux classes, nous additionnons la somme des pondérations données aux attributs avec celle trouvée pour les composants des classes. Puis nous calculons le rapport de la valeur obtenue avec la somme des cardinalités minimales des ensembles d'attributs et de composants. Ce rapport est défini par la fonction r_s suivante.

®r_s *: UMLClasse* x *UMLClasse* f R
Ç_ _
®A *a, b: UMLClasse* ¥ r_s*(a, b) =*
®s_a*(allAttributs(a), allAttributs(b)) +*
® s_{gl} *(allComposants(a), allCompopsants(b)) /*
® *(min(#allAttributs(a),#allAttributs(b))+*
® *min(#allComposants(a), #allCompopsants(b)))*
Le rapport sémantique r_s prend ses valeurs dans l'intervalle [0, 1] comme nous pouvons le constater facilement d'après la définition de cette fonction et de celle de s_a et de s_{gl}. Ainsi, à travers la valeur de ce rapport, nous pouvons déterminer le lien sémantique qui existe entre deux classes. Plusieurs cas peuvent se présenter.

- Une valeur 0 de ce rapport signifie qu'il n'existe aucune correspondance sémantique entre les structures des deux classes sous-jacentes.
- La valeur 1 traduit le fait que la structure de l'une des classes est incluse dans celle de l'autre.
- Une valeur supérieure à p_2 traduit un lien sémantique fort entre les structures.
- Une valeur inférieure ou égale à p_2 traduit un lien sémantique faible entre les structures.

A chacun de ces cas nous faisons correspondre une relation binaire. Ainsi la relation suivante appelée équivalence sémantique traduit le fait que les deux classes sous-jacentes ont le même nombre d'attributs et un rapport sémantique égal à 1.

®_ º$_s$_: P*(UMLClasse* x *UMLClasse)*
Ç_ _
®A *a,b : UMLClasse* ¥ *a* º$_s$ *b* Û r_s *(a, b) = 1* ¦
® *# allAttributs (a) = # allAttributs (b)* ¦
® *# allComposants (a) = # allComposants (b)*

Lorsque le rapport sémantique de deux classes vaut 1 et que les classes n'ont pas le même nombre d'attributs ou de composants, alors il s'agit d'une relation d'inclusion sémantique. Cette relation est définie comme suit :

®_ ,$_s$_: P*(UMLClasse* x *UMLClasse)*
Ç_ _
®A*a, b : UMLClasse* ¥ *a* ,$_s$ *b* Û r_s *(a, b) = 1* ¦
® *# allAttributs (a)* < *# allAttributs (b)* ¦
® *# allComposants (a)* < *# allComposants (b)*

Dans le cas où le rapport sémantique de deux classes est supérieur à p_2 cela signifie qu'il y a forcément des attributs (ou des composants) dont les noms sont identiques ou lié par un lien de synonymie ou de composition. Lorsqu'un tel cas se présente, nous avons une relation d'intersection

sémantique forte. Cette relation est définie comme suit :

$\mathcal{R}\_\ \acute{e}_{ss}\ \_ : \mathcal{P}(UMLClasse \times UMLClasse)$
$\forall a, b : UMLClasse \bullet a\ \acute{e}_{ss}\ b \Leftrightarrow 1 > r_s(a, b) > p_2$

Quand le rapport sémantique des attributs est inférieur à p_2, mais différent de 0, cela veut dire que les classes sous-jacentes ont peu d'attributs et de composants sémantiquement liés. Dans ce cas nous avons une relation d'intersection sémantique faible. Sa définition est donnée par le schéma Object-Z suivant.

$\mathcal{R}\_\ \acute{e}_{ws}\ \_ : \mathcal{P}(UMLClasse \times UMLClasse)$
$\forall a, b : UMLClasse \bullet a\ \acute{e}_{ws}\ b \Leftrightarrow 0 < r_s(a, b) \neq p_2$

Lorsque les attributs et les composants de deux classes n'ont aucun lien sémantique entre eux, nous parlons alors d'une disjonction sémantique. Elle est définie par:

$\mathcal{R}\_\ <>_s\ \_ : \mathcal{P}(UMLClasse \times UMLClasse)$
$\forall a, b : UMLClasse \bullet a\ \ddot{e}_s\ b \Leftrightarrow r_s(a, b) = 0$

Correspondances entre structure locale et structure globale

La comparaison entre la structure globale d'une classe et la structure locale d'une autre permet de détecter certains conflits structurels. Rappelons qu'un conflit structurel se définit par le fait que la même information est représentée par des éléments de modélisation différents dans des représentations conceptuelles différentes.

Afin de pouvoir localiser ce type de conflits, nous confrontons la structure locale d'une classe et la structure globale d'une autre et nous vérifions s'il y a une identité ou une synonymie entre le nom d'un attribut de la première et un nom de composant de la seconde. La fonction *y* suivante réalise cette confrontation.

$\mathcal{R}\phi : \mathcal{P}(UMLClass \times UMLClasse) \rightarrow \mathbb{B}$
$\forall a, b : UMLClasse \bullet \phi(a, b) = true \Leftrightarrow$
$\exists x : allAttributs(a) \bullet \exists y : allComposants(b) \bullet$
$(x.nom = y.nom) \vee synonyme(x.nom, y.nom)$

La présence d'un conflit structurel entre deux classes a et b se traduit donc par une valeur *true* de la fonction ϕ appliquée à ces deux classes.

3.3 - Correspondances dynamiques

L'aspect dynamique d'une classes est représenté en UML par les opérations définies dans la classe et par son diagramme d'états qui décrit le comportement de ses objets face aux événements qu'ils peuvent subir tout au long de leur cycle de vie. La recherche des correspondances dynamiques entre classes se fait donc en deux étapes. Dans la première étape, nous comparons les opérations définies dans les deux classes. Puis nous cherchons les relations entre leurs diagrammes d'états.

Comparaison des opérations

Au niveau conceptuel, les opérations ne sont pas entièrement définies. Les informations que nous avons d'une opération se limitent au nom de la méthode auquel peuvent s'ajouter une liste de paramètres et le type de retour. La comparaison des méthodes de deux classes C1 et C2 concerne donc uniquement les noms des opérations, leurs paramètres et le type qu'elles retournent.

Identiquement au processus de comparaison des attributs, nous commençons par chercher l'ensemble de toutes les méthodes (spécifiques et hérités) de chaque classe et nous définissons une métrique appelée rapport sémantique entre opérations que nous notons r_m (C1, C2) et qui nous renseigne sur le lien sémantique entre les opérations des classes.

$allOpérations : UMLClasse \rightarrow \mathcal{P}\ UMLOpération$
$\forall c : UMLClasse \bullet allOpérations(c) =$
$c.opérations \cup \{p : UMLOpération \mid$
$\exists x : ancêtres(c) \bullet p \in x.opérations \bullet$
$\forall q \in c.opérations \bullet p.nom \neq q.nom\}$

La comparaison des opérations de deux classes consiste à considérer les opérations de l'une d'elles et de chercher pour chacune d'entre elles si son nom est identique ou synonyme de celui d'une opération de l'autre. Pour chaque correspondance trouvée nous comparons la liste des paramètres et les types retournés. Nous attribuons le poids 1 à tout couple d'opérations dont le nombre de paramètres, leurs types et le type retourné sont identiques. Dans le cas inverse la pondération est 0.

La fonction suivante π_m fournit la pondération à attribuer à un couple d'opérations suivant le lien sémantique entre leurs noms.

$p_m : UMLOpération \times UMLOpération \rightarrow \{0, 1\}$
$\forall x, y : UMLOpération \bullet p_m(x, y) = 1 \Leftrightarrow$
$(identique(x.nom, y.nom) \vee$
$synonyme(x.nom, y.nom)) \bullet$
$(\#x.paramètres = \#y.paramètres) \bullet$
$(\forall p : x.paramètres \bullet (\exists q : y.paramètres \bullet$
$p(2).type = q(2).type \bullet$
$p(2).direction = q(2).direction))$
$\forall x, y : UMLOpération \bullet p_m(x, y) = 0 \Leftrightarrow$
$\neg (synonyme(x.nom, y.nom) \vee$
$identique(x.nom, y.nom)) \vee (\#x.paramètres \neq$
$\#y.paramètres) \vee (\exists p : x.paramètres \bullet$
$(\forall q : y.paramètres \bullet p(2).type \neq q(2).type \vee$
$p(2).direction \neq q(2).direction))$

Certains termes sont fréquemment employés dans les représentations conceptuelles pour désigner des

opérations. Tel est le cas des mots *ajouter*, *annuler* ou *modifier*. La comparaison de ces attributs n'a pas un grand intérêt car ils ne sont pas très significatifs. Nous regroupons ces mots dans un même ensemble que nous appelons ensemble des opérations non significatives que nous notons *M*. Et dans le processus de comparaison nous vérifions si les opérations à comparer sont significatives ou non et nous en tenons compte dans la comparaison. Ainsi toute pondération concernant des opérations dont les noms sont des mots de l'ensemble *M* sera multipliée par le coefficient de pertinence a_p.

Pour mesurer le lien sémantique entre les comportements des objets de deux classes, nous définissons une fonction que nous appelons rapport sémantique des opérations et que notons r_m. Soient deux ensembles d'opérations X et Y tels que la cardinalité de X est inférieure à celle de Y. Nous comparons chaque opération de X à toutes celles de Y et nous retenons le maximum des pondérations que nous pouvons obtenir. Ensuite nous calculons la somme s_m(X, Y) de toutes les pondérations obtenues et nous la divisons par la cardinalité de l'ensemble X pour avoir enfin le rapport r_m (X,Y). Les fonctions s_m et r_m sont définies comme suit:

s_m: P UMLOpération x P UMLOpération → R
r_m : UMLClasse x UMLClasse → R

∀ X: P UMLOpération • $s_m(0, X) = 0$ ∧
$s_m(X, 0) = 0$
∀ X, Y: P UMLOpération | X ≠ 0 ∧ Y ≠ 0 •
∃ a : X • s_m (X, Y) = max {b : Y •
p (a.nom, b.nom) * p_m (a, b)}+ s_a (X \{a}, Y)
∀a, b : UMLClasse •
#allOpérations(a) ≠ #allOpérations(b) ⇒
$r_m(a, b) = s_m$(allOpérations(a),allOpérations(b)) /
#allOpérations(a)
∀a, b : UMLClasse •
#allOpérations(a) > #allOpérations(b) ⇒
$r_m(a , b) = r_m(b, a)$

Le rapport sémantique des opérations r_m prend ses valeurs dans l'intervalle [0, 1] comme nous pouvons le constater facilement d'après la définition de cette fonction. Ainsi, à travers la valeur de ce rapport, nous pouvons déterminer le lien sémantique entre les comportements des deux classes.

La valeur 0 indique une dissemblance totale entre les noms de méthodes des deux classes a et b alors qu'une valeur 1 signifie une inclusion complète de l'ensemble des opérations de a dans l'ensemble des opérations de b. Si r_m est supérieur à 0,5, alors plus que la moitié des méthodes de a sont des méthodes de b.

Comparaison des diagrammes d'états

Le comportement des objets de la classe se représente en UML au moyen d'un diagramme d'états. Il s'agit d'un graphe d'états – transitions dont les nœuds représentent les états que peuvent prendre les objets durant leur cycle de vie. Le graphe comporte des nœuds particuliers, le nœud de départ et les nœuds d'arrivée. Une transition de ce graphe est étiquetée par un événement de type callEvent selon le méta-modèle d'UML défini par l'OMG (2003). Un événement de type callEvent n'est rien d'autre qu'un appel d'une opération de la classe sous-jacente. Ce diagramme est appelé aussi machine à états protocole (protocol state machine) par l'OMG (2003). Dans la suite, l'appellation "diagramme d'états" désigne toujours le diagramme d'états protocole.

La spécification formelle d'un diagramme d'états est donnée par la classe Object-Z suivante :

DiagrammeDEtats
ModelElement
top : sEtat ©
étatDeSousDiagramme : P EtatDeSousDiagramme
transitions : P Transition ©
D
chemins : P *Chemin*
{*self*} = *top. diagrammeDEtats*
∀*t : transition* • *self* = *t. DiagrammeDEtats*
∀ *s: EtatDeSousDiagramme* •
self = s.sousDiagramme
top = *EtatComposite*
top.conteneur = 0
top.sortant = 0
∀ *c* : *chemins* • *self = c.diagrammeDEtats*

Un diagramme d'états m est inclus dans un diagramme d'états n si et seulement si toute transition de m a son équivalent dans n, tout chemin de m a son équivalent dans n et tout *étatDeSousDiagramme* de m est équivalent à un *étatDeSousDiagramme* de n.

$\ddot{o}_{dyn}$ö:P(*DiagrammeDEtats*x*DiagrammeDEtats*)
∀*m,n : DiagrammeDEtats* • $m \ddot{o}_{dyn} n$ ⇔
m.top - n.top ∧
(∀*t :m.transitions* • (∃ *s: n.transitions* • *t - s*)) ∧
(∀c : m.chemins • *(∃ d: n.chemins* • *c - d)) ∧*
(∀e : m.étatDeSousDiagramme •
(∃ f: n.étatDeSousDiagramme • *e - f* ∧
e.sousDiagramme $\ddot{o}_{dyn}$ *f.sousDiagramme))*

Deux diagrammes d'états sont équivalents si chacune est incluse dans l'autre.

$\ddot{o}-_{dyn}$ö:P(*DiagrammeDEtats*x*DiagrammeDEtats*)
∀*m,n : DiagrammeDEtats* • $m -_{dyn} n$ ⇔
$m \ddot{o}_{dyn} n$ ∧ $n \ddot{o}_{dyn} m$

Il existe une intersection sémantique entre deux diagrammes d'états lorsque ces deux diagrammes ont des transitions en commun.

®_é$_{dyn}$_:P (*DiagrammeDEtats* x
® *DiagrammeDEtats*)
Ç_ _
®A *m, n :* DiagrammeDEtats ¥ *m* é$_{dyn}$ *n* Û
®*m* ë *0* ¦ *n* ë *0* ¦ *!* m ,$_{dyn}$ *n* ¦ *!* n ,$_{dyn}$ *m* ¦
®E *s* z *m.transitions*ÎA*e* : *s*ÎE *f* : *n.transitions*Î *e* - *f*

Nous avons une disjonction sémantique entre deux diagrammes d'états m et n quand aucune transition de m ne correspond à un transition de n.

®ö<>$_{dyn}$ö: P (*DiagrammeDEtats* x
® *DiagrammeDEtats*)
Ç_ _
®A*m,n : DiagrammeDEtats* ¥ *m* <>$_{dyn}$ *n* Û
® *m* ë *0* ¦ *n* ë *0* ¦ *!* m ,$_{dyn}$ *n* ¦ *!* n ,$_{dyn}$ *m* ¦
® *!* m é$_{dyn}$ *n*

3.4 - Processus de comparaison

Le processus de comparaison se base sur les critères que nous avons déjà explicités dans les sections précédentes. Ces critères sont considérés selon un ordre de priorité : linguistique, structurel et dynamique. L'association de ces différents critères permet d'avoir une comparaison complète traitant tous les aspects possibles.

La comparaison de deux classes débute par la lecture de leurs noms et la recherche dans l'ontologie du domaine du lien sémantique entre eux. Lorsqu'un tel lien existe, nous passons à la comparaison structurelle traitant les attributs et les composants des classes. Ces derniers sont comparés suivant leur nom et leur type. Cette comparaison se fait en même temps que le calcul du rapport sémantique. La valeur de ce rapport traduit l'acuité du lien sémantique entre les classes en question.

Si le rapport sémantique est très faible entre les deux classes, alors leur comparaison s'arrête à ce niveau. Dans le cas inverse, leurs comportement est étudié. Cette étude consiste à calculer le rapport sémantique des opérations r_m et à passer à la comparaison des diagrammes d'états si ce rapport est assez important. L'utilisateur est sollicité pour définir le seuil au dessus duquel la comparaison des diagrammes d'états peut être effectuée. Ce seuil est noté æ. Lors de la comparaison des opérations celles qui sont communes ou synonymes sont marquées.

Les résultats de la comparaison sont enregistrés dans une table, que nous baptisons Table de comparaison, pour les utiliser de l'étape d'intégration. L'algorithme de comparaison est donc le suivant :

Début
lire les définitions des classes *X* et *Y*
comparer les noms de *X* et de *Y*
Si (*p0* < p_a(*X.nom, Y.nom*) et p_a(*X.nom, Y.nom*) <= *p4)*
Alors calculer r_s (X, Y)
SI (p_a(*X.nom, Y.nom*) ë *p4* v (*X* é$_{ss}$ *Y*) v (*X* ,$_s$ *Y*) v (*X* -$_s$ *Y*))
Alors calculer *y*(*X, Y*)
calculer r_m (*X, Y*)
Si (r_m (*X, Y*)ùæ ¦ *X.dEtats* ë ∅ ¦ *Y.dEtats* ë ∅)
Alors déterminer lien entre diagrammes d'états de *X* et de *Y*
FinSi
FinSi
FinSi
inscrire résultats dans la table de comparaison et arrêter la comparaison de *X* et *Y*
Fin

Afin de pouvoir spécifier la table de comparaison et la présentation des informations qu'elle contient, quelques définitions sont nécessaires. Ainsi, nous définissons une première fonction *getLienLinguistique* qui traduit le lien linguistique entre deux noms en une chaîne de caractères pour en simplifier la présentation. Cette chaîne de caractères est l'une des valeurs de l'ensemble *LienLinguistique* défini ci-après.

LienLinguistique ::= *IDENT* | *SYN* | *HYPER_AB* | *HYPER_BA* | *COMP_AB* | *COMP_BA*| *AUCUN.*

La fonction *getLienLinguistique* est définie de la manière suivante :

®*getLienLinguistique : UMLClasse* x *UMLClasse*
® f *LienLinguistique*
Ç_ _
®A*a, b : UMLClasse* ¥ *identique (a.nom, b.nom)* Û
® *getLienLinguistique(a, b) = IDENT*
®A*a, b : UMLClasse* ¥ *synonyme(a.nom, b.nom)* Û
® *getLienLinguistique(a, b) = SYN*
®A*a, b : UMLClasse* ¥ *gen_spec(a.nom, b.nom)* Û
® *getLienLinguistique(a, b) = HYPER_AB*
®A*a, b : UMLClasse* ¥ *gen_spec (b.nom, a.nom)* Û
® *getLienLinguistique(a, b) = HYPER_BA*
®A*a, b :UMLClasse*¥ *composition(a.nom, b.nom)*
® Û *getLienLinguistique(a, b) = COMP_AB*
®A*a, b : UMLClasse* ¥ *composition (b.nom, a.nom)*
® Û *getLienLinguistique(a, b) = COMP_BA*
®A*a, b:UMLClasse* ¥ *! identique (a.nom, b.nom)* ¦
® *! synonyme(a.nom, b.nom)* ¦
® *! gen_spec(a.nom, b.nom)* ¦
® *! gen_spec (b.nom, a.nom)* ¦
® *! composition(a.nom, b.nom)* ¦
® *! composition (b.nom, a.nom)* Û
® *getLienLinguistique(a, b) = AUCUN*

Nous définissons aussi une deuxième fonction *getLienStructurel* qui renvoie, suivant le lien sémantique qui existe entre les classes, l'une des chaînes de l'ensemble *LienStructurel* défini par :

LienStructurel ::= *EQU* | *INC_AB* | *INC_BA* | *INTER_FO* | *INTER_FA* | *DISJ.*

La fonction *getLienStructurel* a la définition suivante :

getLienStructurel : UMLClasse x UMLClasse
LienStructurel x R

∀a, b : UMLClasse • a ≈s b
getLienStructurel (a, b).1 = EQU
∀a, b : UMLClasse • a ⊃s b
getLienStructurel (a, b).1 = INC_AB
∀a, b : UMLClasse • b ⊃s a
getLienStructurel (a, b).1= INC_BA
∀a, b : UMLClasse • a ∩ss b
getLienStructurel (a, b).1= INTER_FO
∀a, b : UMLClasse • a ∩ws b
getLienStructurel (a, b).1 = INTER_FA
∀a, b : UMLClasse • a ⊥s b
getLienStructurel (a, b).1= DISJ

Nous définissons enfin une troisième méthode *getLienDynamique* qui renvoie, suivant le lien qui existe entre les diagrammes d'états des classes, l'une des chaînes contenues dans l'ensemble *LienDynamique*. Cet ensemble est un type libre défini par :

LienDynamique ::= DEQU | *DINC_AB* | *DINC_BA* | *DINTER* | *DDISJ.*

La fonction *getLienDynamique* est alors décrite par

getLienDynamique : UMLClasse x UMLClasse
f LienDynamique x R

∀a, b : UMLClasse • a ≈dyn b
getLienDynamique(a, b) = DEQU
∀a, b : UMLClasse • a ⊃dyn b
getLienDynamique (a, b) = DINC_AB
∀a, b : UMLClasse • b ⊃dyn a
getLienDynamique (a, b) = DINC_BA
∀a, b : UMLClasse • a ∩dyn b
getLienDynamique (a, b) = DINTER
∀a, b : UMLClasse • a ⊥dyn b
getLienDynamique (a, b) = DDISJ

Par ailleurs, nous définissons pour chaque paire de RCs A et B une table *TableComp* qui doit contenir les résultats de la comparaison. Chaque entrée *TableComp*[i, j] de cette table est un objet de la classe *Cellule* définie ci-dessous. Une cellule est caractérisée par une référence *ref* et un contenu *contenu*.

La table est donc composée d'un ensemble de cellules correspondant chacune à deux classes A[i] et B[j] provenant respectivement des RC A et B. La cellule ayant la référence (i, j) est un tableau de 7 éléments appartenant au type *TableRes* défini par :

TableRes==*LienLinguistique* x *LienStructurel* x **B** x**R** x *LienDynamique* x R x **B**

et qui contient, dans l'ordre, le lien linguistique entre le nom de A[i] et celui de B[j], le lien sémantique entre ces deux classes, l'existence ou non de conflits structurels, la valeur du rapport des opérations, le lien dynamique de ces deux classes, un score calculé à partir des résultats précédents et une valeur booléenne qui servira au marquage des cellules dans la phase d'intégration. Le score mesure le rapprochement sémantique entre les classes A[i] et B[j]. Sa formule de calcul est la suivante :

$score = (b1*p_a(\text{A[i].nom, B[j].nom}) + b2*r_s(\text{A[i], B[j]}) + b3*p_{dyn}(\text{A[i].dEtats, B[j].dEtats})) / (b1+b2+b3)$,

où *b1*, *b2* et *b3* sont des coefficients définis afin d'offrir à l'utilisateur la possibilité de créer un ordre de priorité entre les critères. Par exemple, si *b1*=1, *b2*= 2 et *b3* =1, alors le critère structurel devient plus important, dans la comparaison, que ceux linguistique et dynamique.
La modification des priorités des trois critères est intéressante permet, aussi, de réaliser des tests permettant d'étudier, par exemple, l'influence de chacun de ces critères sur le résultat de l'intégration.
La définition d'une Cellule est donnée par le schéma Object-Z suivant :

Cellule
ref : N x N
contenu : TableRes

La table *TableComp* résultant de la comparaison de deux RC A et B forme une partition. Chaque élément de la partition est aussi une table. Les entêtes de lignes d'une table de la partition sont formés de toutes les classes d'une hiérarchie d'héritage dans A, tandis que les entêtes de colonnes forment des classes d'une hiérarchie d'héritage de B. La représentation de la table de comparaison sous forme d'une partition nous aidera à optimiser le processus d'intégration.

TableComp
ligne, colonne : iseq UMLClasse
cellules : P Cellule
partition : P Table

```
®Ç _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®®(1) #cellules = #ligne * #colonne
®®(2)Ac : cellules ¥ E i : dom ligne ¥
®®                    E j : dom colonne ¥ c.ref = (i, j)
®®(3) #partition ø #cellules
®®(4)Ap, q: partition¥ p.cellules I q.cellules= 0 ¦
®®              ran p.ligne I ran q.ligne= 0 ¦
®®              ran p.colonne I ran q.colonne= 0
®®(5) cellules = U {p: partition ¥ p.cellules}
®®(6) ran ligne = U {p: partition ¥ ran p.ligne}
®®(7)ran colonne=U {p:partition¥ ran p.colonne}
®Ð _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Ð _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

L'invariant *(1)* dans la spécification de la classe *TableComp* signifie que le nombre de cellules dans une table doit être toujours égal au produit du nombre d'entêtes de lignes par le nombre d'entêtes de colonnes. Chaque cellule dans la table a comme référence, selon l'invariant *(2)*, le numéro de la ligne et celui de la colonne où elle se trouve. La condition *(3)* vérifie que le nombre d'éléments de la partition est égal au plus au nombre de cellules. La partition est formée de tables disjointes (invariant *(4)*) dont l'union forme la table globale (invariants *(5), (6) et (7)*).

Une table de comparaison est calculée par une opération de la classe *TableComp* que nous notons *calculerTabComp*.

```
È calculerTabComp _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®D (ligne, colonne, partition, cellules)
®A?, B? : DiagrammesDeClasses
Ç _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®(1) ran ligne' = A?.classes
®(2) ran colonne' = B?.classes
®(3)Ap: partition' ¥ A i, j : 1..# p.ligne¥ ië j Þ
®    E g: A?.gen¥ (g.super= p.ligne i ¦
®    g.sub= p.ligne j) v (g.super= p.ligne j ¦
®                        g.sub= p.ligne i)
®(4)Ap: .partition' ¥ A i, j : 1..# p.colonne¥ ië j Þ
®         E g: A?.gen¥ (g.super= p.colonne i ¦
®      g.sub= p.colonne j) v (g.super= p.colonne j ¦
®                              g.sub= p.colonne i)
® (5)Ac : cellules' ¥ c.contenu.1=
®getLienLinguistique(ligne(c.ref.1),
®                                  colonne(c.ref.2))
®(6)Ac : cellules' ¥ c.contenu.1= AUCUN Þ
®       c.contenu.2 = DISJ ¦ c.contenu.3= false ¦
®       c.contenu.4=0 ¦ c.contenu.5= DDISJ
®(7)Ac :cellules' ¥ c.contenu.2 = getLienStructurel(
®        r_s(ligne(c.ref.1),colonne(c.ref.2)))
®(8)Ac : cellules' ¥ c.contenu.2 = INTER_FA v
®       c.contenu.2=DISJ Þ
®       c.contenu.4=0 ¦ c.contenu.5= DDISJ
®(9)Ac : cellules' ¥ c.contenu.2 = EQU Þ
®           c.contenu.3= false
®(10)Ac : cellules' ¥ c.contenu.2 ë EQU Þ
®  c.contenu.3=ø(ligne (c.ref.1), colonne(c.ref.2))
®(11)Ac : cellules' ¥ c.contenu.2 ë INTER_FA ¦
®           c.contenu.2 ë DISJ Þ
®(cellules i j).4= r_m(ligne(c.ref.1), colonne(c.ref.2))
®(12)Ac : cellules' ¥ c.contenu.4 ù æ Þ
®      c.contenu.5 = getLienDynamique(
®                  ligne (c.ref.1), colonne(c.ref.2))
®(13)Ac : cellules' ¥ c.contenu.6 = (b1*
®    p_a(ligne(c.ref.1).nom, colonne(c.ref.2).nom)+
®    b2*r_s(ligne(c.ref.1), colonne(c.ref.2)) +b3*
®p_dyn(ligne(c.ref.1).dEtats,colonne(c.ref.2).dEtats))
®       / ( b1+b2+b3)
Ð _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

L'opération *calculerTableComp* attribue aux entêtes des lignes de la table de comparaison les classes du diagramme A, tandis que celles des colonnes reçoivent les classes de B selon les post-conditions *(1)* et *(2)*. De plus, les entêtes des lignes sont organisées de façon à avoir dans chaque table de la partition de la table globale des entêtes de lignes qui constituent les classes d'une même hiérarchie d'héritage dans A (post-condition *(3)*). Une post-condition analogue est valable pour les entêtes des colonnes (post-condition *(4)*). Les prédicats (5)-(13) montrent comment calculer le contenu des cellules de la table à l'aide des fonctions *getLienLinguistique, getLienStructurel* et *getLienDynamique*.

	B1	B2	B3	B4
A1	SYN	HYPER_BA	COMP_AB	AUCUN
	INTER_FA	INTER_FO	DISJ	DISJ
	false	true	false	false
	0	0.5	0	0
	DDISJ	DINTER	DDISJ	DDISJ
	#	#	#	#
	false	false	false	false
A2	HYPER_AB	AUCUN	SYN	AUCUN
	INC_AB	DISJ	INC_AB	DISJ
	false	false	true	false
	0	0	1	0
	DDISJ	DDISJ	DEQU	DDISJ
	#	#	#	#
	false	false	false	false
A3	COMP_AB	SYN	HYPER_AB	HYPER_AB
	INTER_FA	DISJ	INTER_FO	INC_AB
	false	false	false	true
	0	0	0.2	0.75
	DDISJ	DDISJ	DDISJ	DEQU
	#	#	#	#
	false	false	false	false
A4	AUCUN	IDENT	AUCUN	AUCUN
	DISJ	EQU	DISJ	DISJ
	false	false	false	false
	0	0.75	0	0
	DDISJ	DINC_AB	DDISJ	DDISJ
	#	#	#	#
	false	false	false	false

Figure 3.Exemple de Table de comparaison

La table de la figure 3 est un exemple de résultat renvoyé par l'algorithme de comparaison. Les

représentations conceptuelles qui sont comparées contiennent respectivement les classes A1, A2, A3, A4 et B1, B2, B3, B4.
La comparaison des classes A3 et B4, par exemple, fournit les informations suivantes :

- Le nom de la classe A3 est une généralisation de celui de B4 (Hypernymie),
- La structure de B4 est incluse dans celle de A3 puisque nous avons une inclusion sémantique.
- Il y a deux conflits structurels entre A3 et B4.
- Le rapport des opérations vaut 0.75.
- Les diagrammes d'états sont équivalents.

Cette comparaison montre que les classes A3 et B4 sont sémantiquement très proches ce qui donne la possibilité de les fusionner ou de créer un lien de généralisation spécialisation entre elles lors de leur intégration dans la RC globale.

4 - ETAPE D'INTEGRATION

L'intégration de RCs consiste à prendre deux à deux les représentations conceptuelles et à créer une nouvelle représentation conceptuelle bien formée et cohérente contenant toutes les informations des RCs d'origine. L'intégration est guidée par la table de comparaison qui montre toutes les correspondances et tous les conflits détectés pendant la phase de comparaison. Afin de résoudre ces conflits, l'élaboration d'un ensemble de règles est nécessaire. Ces règles permettent de guider le processus d'intégration en indiquant les actions à entreprendre et les conditions sous lesquelles ces actions sont réalisées. Pour simplifier la spécification de ces règles, nous commençons par définir un catalogue de fonctions décrivant les actions pouvant être appliquées par le processus d'intégration de RCs.

4.1 - Catalogue des actions

Dans cette section, nous présentons un certain nombre d'actions qui sont utilisées dans les règles d'intégration. Elles sont toutes définies en Object-Z. et regroupées dans les sections suivantes selon l'élément de RC auquel elles sont appliquées.

Opérations sur les attributs

Les opérations appliquées aux attributs sont des fonctions qui seront appelées lors de l'intégration d deux classes. Il peut s'agir de copier un attribut ou un ensemble d'attributs, de fusionner ou de trouver l'intersection de deux ensembles d'attributs.

Copier un attribut
La copie d'attribut est nécessaire lorsque un ou plusieurs attributs d'une classe de RP source doivent être reproduites dans une classe de la RP intégrée. La fonction copierAttribut est définie comme suit :

® *copierAttribut*: *UMLAttribut* f P *UMLAttribut*
Ç _
®A*a*: *UMLAttribut* ¥ *copierAttribut* (*a*) =
®{*b*: *UMLAttribut* | *a* ë *b* ¦ *a.nom* = *b.nom* ¦
® *a.type* = *b.type*}

Une copie d'un attribut est tout autre attribut ayant le même nom et le même type que le premier.

Copier un ensemble d'attributs
La fonction de copie d'un ensemble d'attribut remplace plusieurs appels consécutifs de la fonction *copierAttribut* définie ci-dessus. Nous l'introduisons pour simplifier les fonctions faisant appel à la copie de plusieurs attributs à la fois. Cette fonction est définie comme suit :

® *copierAtts* : P *UMLAttribut* f P *UMLAttribut*
Ç _
®A*a, b*: P *UMLAttribut* ¥ *b* e *copierAtts* (*a*) Û
®#*a* = #*b* ¦ (A *x*: *a* ¥ E *y* : *b* ¥ *y* e *copierAttribut*(*a*))

La fonction *copierAtts* appliquée à un ensemble d'attribut renvoie un nouvel ensemble ne contenant que des copies de tous les élément du premier.

Fusionner des attributs
La fonction fusionnerAttributs permet de fusionner deux ensembles d'attributs. Elle sera appliquée lors de la fusion de deux classes dont les objets sont similaires.
®*fusionnerAttributs* :P*UMLAttribut*xP*UMLAttribut*
® f P (P *UMLAttribut*)
Ç _
®A*a, b, c*: P *UMLAttribut* ¥
® *c* e *fusionnerAttributs* (*a, b*) Û
®(*1*) #*c* = #*a* + #*b* - #{*u* : *b* | E *v*: *a* ¥
® *u.nom* = *v.nom* ¦ *c*(*u.type, v.type*) = *1*} ¦
®(*2*) A *z, t* : *c* ¥ *z.nom* ë *t.nom* ¦
® ! *synomyme*(*z.nom, t.nom*)) ¦
®(*3*) (A*x* : *a* ¥ *x* ä {*u* : *a* | E *v*: *b* ¥ *u.nom* = *v.nom* ¦
® *c*(*u.type, v.type*) = a_c } Þ
® E *y* : *c* ¥ *x.nom* = *y.nom* ¦ *y.type* z *x.type*) ¦
®(*4*) (A*x* : *b* ¥ *x* ä {*u* : *b* | E *v*: *a* ¥ *u.nom* = *v.nom* ¦
® *c*(*u.type, v.type*) = a_c } Þ
® E *y* : *c* ¥ *x.nom* = *y.nom* ¦ *y.type* z*x.type*) ¦
®(*5*)A*x* : *a*;A*y* : *b* ¥ *x.nom* = *y.nom* ¦
® *c*(*x.type*,*y.type*) = a_c Þ
® E *z, t* : *c* ¥ *z.nom* = *x.nom* ¦ *z.type* = *x.type* ¦
® *t.type* =*y.type* ¦ *t.nom* ä {*u* : *x* ¥ *u.nom*}U
® {*v* : *y* ¥ *v.nom*}

Si les ensembles d'attributs à fusionner ont des éléments ayant le même nom et des types compatibles, une seule copie de ces éléments sera ajoutée à un ensemble d'attributs résultat, notamment celle de l'élément ayant le type le plus grand. Cette condition est contrôlée par les prédicats (2), (3) et (4). De plus, si deux attributs ont le même nom est des types non compatibles alors l'un d'eux sera renommé. Le prédicat (5) garantit le respect de cette propriété. Enfin, le

prédicat (1) ajouté à (2), (3), (4) et (5) permet de vérifier que l'ensemble d'attributs résultat contient une copie pour chaque attribut des ensembles de départ et n'en ajoute pas d'autres.

Intersection d'ensembles d'attributs

L'intersection de deux ensembles d'attributs sert à grouper ensemble les attributs qui leurs sont communs. Cette fonction est nécessaire pour factoriser les attributs communs à deux classes dans une même classe générique.

```
® intersectionAttributs : P UMLAttribut x
®          P UMLAttribut   f  P (P UMLAttribut )
Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®Aa, b, c : P UMLAttribut ¥
®          c e  intersectionAttributs (a, b)  Û
®(1) (Ax : c ¥ E  y, z : UMLAttribut ¥ y e a ¦ z e b ¦
®    (x.nom = y.nom v  synonyme(x.nom, y.nom)) ¦
®    (x.nom = z.nom v synonyme(x.nom, y.nom)) ¦
®   c(x.type, y.type) =  1 ¦ c(x.type, z.type) =  1) ¦
®(2) (Ax: a; Ay: b  (x.nom = y.nom v
®    synonyme(x.nom, y.nom)) ¦
®   c(x.type, y.type) = 1  Þ E z : c ¥ z.nom = x.nom ¦
®   c(x.type, z.type) =  1) ¦
®(3) (Az, t : c ¥ z.nom ë t.nom ¦
®              ! synomyme(z.nom, t.nom))
```

La fonction d'intersection de deux ensembles d'attributs a et b crée dans un ensemble d'attributs résultat une seule copie de chaque attribut de a dont le nom est identique ou synonyme à un autre de b et dont les types sont aussi identiques ou compatibles. Les prédicats (1) et (3) vérifient cette post-condition. Le prédicat (2) assure que tout attribut commun à a et b possède une copie dans l'ensemble d'attributs résultat.

Opérations sur les opérations de classes

Les fonctions utilisées pour manipuler les opérations de classes sont analogues à celles appliquées aux attributs. Ainsi nous définissons les fonctions copierOpération, copierOps, fusionnerOps et intersectionOps.

Copier une opération

Lors de l'intégration de deux classes, les opérations définies dans les classes doivent être copier dans la RC globale. Le rôle de la fonction *copierOpération* est d'effectuer cette copie.

```
® copierOpération: UMLOpération  f  P
®                                  UMLOpération
Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®Aa: UMLOpération ¥ copierOpération (a) =
®{b:UMLOpération | a ë b ¦
®a.nom = b.nom ¦ a.paramètres = #b.paramètres ¦
®(A i : 1.. #a.paramètres ¥ a.paramètres(i).nom =
®b.paramètres(i).nom ¦ a.paramètres(i).type =
®b.paramètres(i).type ¦ a.paramètres(i).direction=
®b.paramètres(i).direction)
```

La fonction *copierOpération* appliquée à une opération *Op* renvoie tous les objets de la classe *Opération* dont le nom, le nombre de paramètres, leur type et leur direction (entrée, sortie ou entrée-sortie) coï ncident avec ceux de *Op*.

Copier un ensemble d'opérations

Afin de simplifier les fonctions faisant appel plusieurs fois à la fonction *copierOpération* et afin de rendre plus lisibles ces fonctions, nous avons ajouté une fonction qui permet de copier tout un ensemble d'opérations.

```
® copierOps: P UMLOpération  f
®                              P(P UMLOpération)
Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®Aa, b: P UMLOpération ¥ b e copierOps (a)  Û
®  #a=#b ¦ A x: a ¥ E y : b ¥ y e copierOpération(a)
```

Fusionner des opérations

La fusion de deux classes implique la fusion de leurs ensembles d'attributs et d'opérations. La fusion des opérations est définie comme suit :

```
®fusionnerOps: PUMLOpération x
®          PUMLOpération  f  P(PUMLOpération)
Ç_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®Aa, b, c : P UMLOpération ¥
®                  ce fusionnerOps (a, b)   Û
®(1)(Ax : c ¥ (E  y : UMLOpération ¥ (ye a v ye b) ¦
®    (x.nom = y.nom v synonyme (x.nom, y.nom)) ¦
®   #x.paramètres = #y.paramètres ) ¦
®   (Ap: ran x.paramètres ¥ (E q: ran y.paramètres ¥
®    p.type = q.type  ¦  p.direction =q.direction))) ¦
®(2) (A z, t : c ¥ (z.nom = t.nom ¦ #z.paramètres =
®     # tparamètres ¦ Ai : 1.. #z.paramètres ¥
®     z.paramètres(i).nom =  t.paramètres(i).nom ¦
®    z.paramètres(i).type = t.paramètres(i).type)  Þ
®     z = t) ¦
® (3) (A x : a U b ¥ (E y : c ¥ (x.nom = y.nom ) ¦
®       #x.paramètres = #y.paramètres ) ¦
®   (Ap: ran x.paramètres ¥ (E q: ran y.paramètres ¥
®    p.type = q.type  ¦  p.direction = q.direction)))
```

L'opération *fusionnerOpérations* part de deux ensembles d'opérations pour construire un nouveau. Ce nouvel ensemble doit contenir, d'après (3), des copies de toutes les opérations des ensembles d'origine. L'opération ne permet pas les duplications dans l'ensemble résultat comme nous pouvons le déduire de (4). Par contre les redéfinitions sont acceptées, d'après (1). La condition (4) assure qu'il n'y a pas de synonymie entre les noms d'opérations dans le résultat.

Intersection d'ensembles d'opérations

L'intersection de deux ensembles d'opérations permet de regrouper les éléments qui se répètent dans ces deux ensembles. Cette opération est utile lorsque nous factorisons deux classes

sémantiquement similaires et qui ont des opérations en communs.

® *intersectionOps* : P *UMLOpération* x
® P *UMLOpération* f P(P*UMLOpération*)
Ç_ _
® A*a, b, c* : P *UMLOpération* ¥
® c e *intersectionOps* (*a, b*) Û
®(*1*) (A*x* : *c* ¥ (E *y, z* : *UMLOpération* ¥ (*y* e *a* ¦
® *z* e *b*) ¦
® (*x.nom = y.nom* v *synomyme*(*x.nom, y.nom*)) ¦
® (*x.nom = z.nom* v *synomyme*(*x.nom, z.nom*)) ¦
®(*2*) *#x.paramètres = #y.paramètres* ¦
® *#x.paramètres = #z.paramètres* ¦
®(*3*)(A*p*: *ran x.paramètres* ¥ (E *q*:*ran y.paramètres*¥
® *p.type = q.type* ¦ *p.direction = q.direction*)) ¦
®(*4*) (A*p*: *ran x.paramètres*¥(E *q*: *ran z.paramètres*¥
® *p.type = q.type* ¦ *p.direction = q.direction*)) ¦
®(*5*) (A *z, t* : *c* ¥ ! *synonyme*(*z.nom = t.nom*))

La fonction *intersectionOpérations* reçoit en entrée deux ensembles d'opérations et renvoie des ensembles contenant chacun des copies de toutes les opérations qui se répètent dans les ensembles de départ ou qui ont des noms synonymes, le même type et la même direction des paramètres.

Opération sur les diagrammes d'états

Les actions que le processus d'intégration peut réaliser sur les diagrammes d'états sont la copie et la fusion. La copie de diagramme d'états traite le cas où le diagramme d'état d'une classe source doit être copié intégralement dans une classe cible. Tandis que la fusion s'applique généralement quand il s'agit de fusionner deux classes. Dans ce cas les diagrammes d'états peuvent être aussi fusionnés.

Copier un diagramme d'états

Les cas où l'opération de copie d'un diagramme d'états est appliquée sont les suivants :

- quand une classe entière doit être copiée dans la RC globale. Dans ce cas le diagramme d'états sera lui aussi copié,
- ou quand deux classes sont fusionnées et que le diagramme d'états de l'une est inclus dans celui de l'autre.

® *copierDEtats* : *DiagrammeDEtats* f
® *P DiagrammeDEtats*
Ç_ _
®A *m, n*: *DiagrammeDEtats* ¥
® *n* e *copierDEtats* (*m*) Û m -$_{dyn}$ n

Un diagramme d'états est une copie d'un autre si les deux sont équivalents au sens de la relation d'équivalence entre diagrammes d'états que nous avons définie dans la section 5.2.

Fusionner deux diagrammes d'états

La fusion de deux diagrammes d'états repose sur un algorithme de fusion proposé par Elkoutbi (2000) dans un contexte différent du notre, notamment dans le contexte de construction de diagramme d'états d'une classe à partir de cas d'utilisation. Cet algorithme peut être adapté à notre cas en omettant certaines étapes qui ne sont pas nécessaires pour notre problématique. Nous modifions aussi la terminologie utilisée dans Elkoutbi (2000) et qui ne fait pas parti du standard UML.

Etant donné que les diagrammes d'états sont composés d'états et de transitions, la fusion de deux diagrammes d'états consiste à fusionner les états et les transitions. Ainsi, deux fonctions sont nécessaire, l'une pour fusionner les états et l'autre pour fusionner les transitions. La fonction de fusion de diagrammes d'états *fusionnerDEtats* fait donc appel à ces deux fonctions f.

Opérations sur les classes

Les opérations que nous pouvons effectuer sur deux classes lors du processus d'intégration sont : la copie, la fusion, la factorisation, la généralisation et la composition. Ces quatre opérations seront décrites dans les sections suivantes.

Copier une classe

La fonction *copierClasse* permet de générer à partir d'une classe, une autre classe ayant exactement les mêmes attributs et les mêmes opérations. Cette fonction est utilisée, par exemple, pour copier dans le diagramme de classes global toutes les classes des représentations conceptuelles d'origines qui doivent être ajoutées au diagramme sans aucune modification.

® *copierClasse* : *DiagrammeDeClasses* x
® *sUMLClasse* f *P sUMLClasse*
Ç_ _
®A*a*: *sUMLClasse ;* A *D*: *DiagarammeDeClasses* ¥
®*copierClasse* (*a*)={*b*: *UMLClasse*| *b.nom= a.nom*
® ¦ *# b.attributs = #a.attributs* ¦
® *# b.opérations = #a.opérations* ¦
®*b.attributs* = {*x*: *UMLAttribut* | E *y* : *a.attributs* ¥
®*x* e *copierAttribut* (*y*)} ¦ *b.opérations* =
® {*x*:*UMLOpération* | E *y*: *a.opérations* ¥
® *x* e *copierOpération* (*y*)} ¦ *b.diagramme = D*}

La fonction *copierClasse* copie tous les attributs et toutes les opérations de la classe source dans la classe destination.

Fusionner deux classes

L'opération fusionnerClasses permet de fusionner deux classes de représentations conceptuelles différentes en une nouvelle classe dans la représentation conceptuelle en cours de construction. La nouvelle classe générée contient la fusion des attributs et des méthodes des classes d'origine. La fusion des attributs et celles des méthodes telles qu'elles sont définies dans les sections correspondantes évitent la duplication, dans la classe générée, des éléments communs aux deux classes. La fusion de deux classes est généralement pratiquée lorsque les noms sont

identiques ou synonymes, si leurs sémantiques sont très proches.
Suivant le diagramme d'état choisi pour la classe résultat de la fusion, nous définissons trois fonction de dusion de classes.

® *fusionnerClasses1* : *DiagrammeDeClasses* x
®s*UMLClasse* x s*UMLClasse* f P s*UMLClasse*
Ç_ _
®A *a, b, c* :s*UMLClasse;*
® A*D*: *DiagarammeDeClasses*¥
®*c* e *fusionnerClasses1* (*a, b*) Û
® *c.nom =a.nom* ¦ *c.attributs* e
®*fusionnerAttributs*(*a.attributs, b.attributs*) ¦
®*c.opérations* e *fusionnerOps*(*a.opérations,*
®*b.opérations*) ¦*c.dEtats*e *fusionnerDEtats*(*a, b*) ¦
®*c.diagramme=D*

La fonction *fusionnerClasses1* fusionne les attributs, les opérations et les diagrammes d'états des classes d'origines a et b. La nouvelle classe résultant de la fusion porte le nom de a. Ce choix est tout à fait arbitraire et se justifie par le fait que la fusion n'est adoptée que pour les classes ayant des sémantiques similaires.

® *fusionnerClasses2* : *DiagrammeDeClasses* x
®s*UMLClasse* x s*UMLClasse* f P s*UMLClasse*
Ç_ _
®A *a, b, c* : s*UMLClasse ;*
® A *D*: *DiagarammeDeClasses* ¥
®*c* e *fusionnerClasses2* (*a, b*) Û
®*c.nom =a.nom* ¦ *c.attributs* e
® *fusionnerAttributs*(*a.attributs, b.attributs*) ¦
®c.opérations e *fusionnerOps*(*a.opérations,*
®*b.opérations*) ¦ c.*dEtats*e *copierDEtats*(a) ¦
®*c.diagramme=D*

® *fusionnerClasses3* : *DiagrammeDeClasses* x
® s*UMLClasse* x s*UMLClasse* f P s*UMLClasse*
Ç_ _
®A *a, b, c* : s*UMLClasse ;*
® A *D*: *DiagarammeDeClasses* ¥
®*c* e *fusionnerClasses3* (*a, b*) Û
®*c.nom =b.nom* ¦ *c.attributs* e
® *fusionnerAttributs*(*a.attributs, b.attributs*) ¦
®c.opérations e *fusionnerOps*(*a.opérations,*
® *b.opérations*) ¦ c.dEtatse copierDEtats(b) ¦
® *c.diagramme=D*

La fusion des classes peut ne pas être accompagnée d'une fusion des diagrammes d'états des classes d'origine, mais plutôt de la copie de l'un de ces diagrammes dans la classe fusionnée. Ceci est notamment le cas lorsque nous avons un lien d'inclusion entre les diagrammes d'états. Les fonctions *fusionnerClasses2* et *fusionnerClasses3* ci-dessus traitent ces cas.

Factoriser deux classes

La factorisation de classes est utile dans le cas où deux classes sont sémantiquement proches et ont plusieurs attributs et méthodes en commun. Il est donc possible de regrouper leurs propriétés communes dans une nouvelle classe abstraite et de relier les classes d'origine à la nouvelle classe par un lien de généralisation spécialisation.

® *factoriserClasses* : *DiagrammeDeClasses* x
®s*UMLClasse* x s*UMLClasse* f P(s*UMLClasse*x
®s*UMLClasse* x s*UMLClasse x*
®*UMLGénéralisation x UMLGénéralisation*)
Ç_ _
®A*a, b, c, d* : s*UMLClasse; g, h* :
®*UMLGénéralisation*; *D*: *DiagarammeDeClasses*¥
® (*c, d, e, g, h*)e *factoriserClasses* (*a, b*) Û
®(*1*) (*identique*(*a.nom, b.nom*) Þ E *s*: *Terme*¥
® *c.nom= s* ¦ sä {*x*: *a.diagramme.classes*¥
® *x.nom*}U {*y*:*b.diaramme.classes*¥ *y.nom*}U
® {*z*:*D.classes*¥ *z.nom*}) ¦ *b.nom = d.nom* ¦
® (*! identique*(*a.nom, b.nom*) Þ *c.nom= a.nom*) ¦
®(E *t*: *Terme*¥ *e.nom= t* ¦ të *c.nom* ¦ të *d.nom*)) ¦
®(*2*) *c.attributs* e *copierAtts*(*a.attributs*)¦
® *c.attributs* ä *intersectionAttributs*(*a,b*) ¦
®(*3*) *c.opérations* e *copierOps*(*a.opérations*) ¦
® *c.opérations* ä *intersectionOpss*(*a,b*) ¦
® *c.diagramme=D* ¦
®(*4*) *d.attributs* e *copierAtts*(*b.attributs*)¦
® *d.attributs* ä *intersectionAttributs*(*a,b*) ¦
®(*5*) *d.opérations* e *copierOps*(*b.opérations*) ¦
® *d.opérations* ä *intersectionOps*(*a,b*) ¦
® *d.diagramme=D* ¦
®(*6*) *e.attributs* e *intersectionAttributs*(*a, b*) ¦
®(*7*) *e.opérations* e *intersectionOps*(*a, b*) ¦
®(*8*)*e.dEtats*e*copierDEtats*(*a.dEtats*)¦*c.dEtats= 0*
® ¦ *d. dEtats* e *copierDEtats* (*b. dEtats*) ¦
®(9) *g.super = e* ¦ *g.sous = c* ¦ *h.super = e* ¦
® *h.sous = d* ¦
®(10) e.nom ä{*x*: *a.diagramme. classes*¥ *x.nom*}U
® {*y*:*b.diaramme.classes*¥ *y.nom*} U
®{*z*:*D.classes*¥ *z.nom*}) ¦ *e.diagramme=D*

Le rôle de la fonction *factoriserClasses* définie ci-dessus est de renvoyer trois classes : la classe contenant les propriétés communes et deux autres classes représentant des copies des classes opérandes desquelles on a supprimé les propriétés communes. Elle renvoie aussi deux liens de généralisaion/spécialisation entre la classe contenant les propriétés communes et chacune des classes restantes.
Le prédicat (1) dans la définition de la fonction signifie que la première classes du 5-uplet résultat doit avoir un nom différent de celui de a, que la deuxième classe porte le même nom que b, et que le nom de la troisième classe est obtenu par la concaténation des noms de a et b à l'aide de la fonction *concat*. Les prédicats (2) jusqu'à (5) traduisent le fait que les classes c et d contiennent respectivement les attributs et les opérations des classes a et b à l'exception de ceux qui leurs sont communs. Les prédicats (6) et (7) vérifient que la classe e ne contient que les attributs et les opérations communes aux classes a et b. Enfin, le

prédicat (9) impose que g et h soient des liens de généralisation/ spécialisation respectivement entre e et c, et entre e et d .

Créer un lien de composition entre deux classes

Dans certain cas, la création d'un lien de composition entre deux classes s'avère nécessaire, car elle permet d'éviter les redondances et de mieux structurer les éléments du diagramme de classes intégré. Les deux cas suivants sont des situations où lien de composition devient utile :

Les noms des classes à fusionner sont liés par un lien de composition et les attributs de la classes dont le nom représente le composant se répètent dans la classe dont le nom est le composite.

Les noms des classes à fusionner sont liés par un lien de composition et un attribut de la classes dont le nom représente le composite porte le même nom que la classe dont le nom est le composant.

Pour chacune de ces situation nous appliquons respectivement les fonctions *composer1* et *composer2*.

®*composer1*: *DiagrammeDeClasses* x*sUMLClasse*
® x *sUMLClasse* f *P* (*sUMLClasse* x
®*sUMLClasse* x*sUMLAssoc*)
Ç_ _
®*A a, b, c, d* : *sUMLClasse; As* : UMLAssoc *;*
® *A D*: *DiagarammeDeClasses* ¥
® (*c, d, s*)e *composer1*(*a, b*) Û
® *c.nom = a.nom* ¦ *c.attributs* e
® *copierAtts*(*a.attributs*)\ *copierAtts*(*b.attributs*) ¦
® *c.opérations* e *copierOps*(*a.opérations*) ¦
®*c.dEtats*e *copier* (*a.dEtats*) ¦ *c.diagramme=D* ¦
®*d*e *copierClasse*(*b*)¦
®*d.dEtats*e*copierD*(*b.dEtats*) ¦ *d.diagramme=D* ¦
®#*s.pattes =2* ¦ *s.pattes*(*1*).*classe = c* ¦
®*s.pattes*(*2*).*classe = d* ¦*s.agrégation = agrégat*

La fonction *composer1* relie deux classes a et b à tout triplet composé d'une copie de la classe a ne contenant pas les attributs de b, d'une copie de b et d'un lien de composition entre ces deux copies.

®*composer2*: *DiagrammeDeClasses* x
®*sUMLClasse x sUMLClasse* f *P* (*sUMLClasse*
® *x sUMLClasse xsUMLAssoc*)
Ç_ _
®*A a, b, c, d* : *sUMLClasse; As* : *UMLAssoc ;*
® *A D*: *DiagarammeDeClasses* ¥
® (*c, d, s*) e *composer2*(*a, b*) Û *c.nom = a.nom* ¦
® *c.attributs* e {*X*: *copierAtts*(*a.attributs*) |
® *A x* : *X* ¥ *x.nom* ë *b.nom* }
® *c.opérations* e *copierOps*(*a.opérations*) ¦
® *c. DiagrammeDEtats* e *copierD* (*a. dEtats*) ¦
® *c.diagramme=D* ¦ *d* e *copierClasse*(*b*) ¦
®*d.dEtats*e *copierD*(*b.dEtats*) ¦ *d.diagramme=D* ¦
®#*s.pattes = 2* ¦ *s.pattes*(*1*).*classe = c* ¦
®*s.pattes*(*2*).*classe = d* ¦ *s.agrégation = agrégat*

Chaque couple de classes (a, b) a pour image l'ensemble des triplets composés d'une copie de la classe a ne contenant pas d'attribut de même nom que b, d'une copie de b et d'un lien de composition entre ces deux copies.

Généraliser une classe

La généralisation d'une classe est possible dans le cas où les deux RCs contiennent des classes sémantiquement très proches, mais qui ont des structures différentes. C'est par exemple le cas lorsque le nom d'un classe *a* est un hypernyme du nom d'une classe *b* et que les attributs de *a* son inclus dans ceux de *b*.

®*généraliserClasse*: *DiagrammeDeClasses* x
®*sUMLClasse*x*sUMLClasse* f *P*(*sUMLClasse* x
®*sUMLClasse* x *UMLGénéralisation*)
Ç_ _
®*Ag*: *UMLGénéralisation; Aa, b, c, d* :
®*UMLClasse; A D*: *DiagarammeDeClasses* ¥
®(*c, d, g*) e *généraliserClasse*(*a, b*) Û
® *c.nom = a.nom* ¦
® *c.attributs* e *copierAtts*(*a.attributs*) ¦
®*c.opérations* e *copierOps*(*a.opérations*) ¦
®*c.dEtats* e *copierDEtats*(*a*) ¦
® *c.diagramme=D* ¦ *d.nom = b.nom* ¦
® *d.attributs* e *copierAtts*(*b.attributs*)\
®*copierAtts*(*a.attributs*) ¦
®*d.opérations* e *copierOps*(*b.opérations*) \
®*copierOps*(*a.opération*) ¦
®*d.dEtats* e *copierDEtats*(*b*) ¦ *d.diagramme=D* ¦
®*g.super = c* ¦ *g.sous = d* ¦
®*copierAtts*(*a.attributs*)\ *copierAtts*(*b.attributs*)ë *0*
® Þ *g.commentaire = concChaînes*({*MASQUER*}U
®{*x*: *copierAtts*(*a.attributs*)*copierAtts*(*b.attributs*)¥
®*x.nom*})

La fonction *généraliserClasse* construit à partir de deux classes *a* et *b* un généralisation spécialisation *(c, d, g)* dans laquelle *c* est la classe généralisée, *d* est la classe spécialisée et *g* est le lien de généralisation/spécialisation entre *c* et *d*.

La classe *c* représente une copie de la classe *a* tandis que *d* est une copie de la classe *b* de laquelle nous avons omis tous les attributs et les opérations qui se répètent dans *a*. Si les classes *a* et *b* ont des diagrammes d'états, alors ses diagrammes sont copiés respectivement dans *c* et *d*.

4.2 - Diagramme fusionné

Les diagrammes de classes fusionnés sont des diagrammes particuliers puisqu'ils sont issus de la fusion d'autre diagrammes de classes. Pour les distinguer, nous les regroupons dans une classe particulière que nous nommons DiagrammeFusionné et qui hérite de la classe DiagrammeDeClasses.

È*DiagrammeFusionné*_ _ _ _ _ _ _ _ _ _ _ _ _ _
®*DiagrammeDeClasses*
®È_ _
®®*A, B* : *DiagrammeDeClasses*
®®*T* : *TableComp*
®®*L1* : *iseq* (*N x N x UMLClasse*)

®®*L2* : *iseq*(*UMLClasse*x*UMLClasse*
®® x*UMLCLasse*)
®®*L3* : *iseq* (*UMLClasse*x*UMLClasse*)
®®*L4* : *iseq*(*UMLClasse*x*UMLClasse*)
®Ç_ _
®®(*1*) (*A*(*x, y, z*): *ran L2* ¥ (*x* ä *classes* ¦
®® *y* ä *classes* ¦ *z* e *classes*)
®®(*2*) (*Ax* : *ran L2; a* : *UMLClasse* ¥ *a* = *x.1* Þ
®® *Ay* : *ran L2* ¥ (*y.1*).*nom* ë *a.nom*) ¦
®® (*Ax* : *ran L2; a* : *UMLClasse* ¥ *a* = *x.2* Þ
®® *Ay* : *ran L2* ¥ (*y.2*).*nom* ë *a.nom*)
®® (*Ax* : *ran L2; a* : *UMLClasse* ¥ *a* = *x.3* Þ
®® *Ay* : *ran L2* ¥ (*y.3*).*nom* ë *a.nom*)
®®(*3*) *A*(*x, y*): *ran L3* ¥ (*x* e *T.ligne* ¦
®® *y* e *T.colonne*v *x* e *T.colonne* ¦ *y* e *T.ligne*
®®(*4*)*A*(*x, y*): *ran L3; p, q* : *T.partition* ¥
®® *x* e (*ran p.ligne* U *ran p.colonne*) ¦
®® *y* e (*ran q.ligne* U *ran q.colonne*) Þ
®® ! (E *w, z* : *UMLClasse* ¥ (*w, z*) e ran *L3* ¦
®® *w* e (*ran q.ligne* U *ran q.colonne*) ¦
®® *z* ä (*ran p.ligne* U *ran p.colonne*) v
®® *z* e (*ran p.ligne* U *ran p.colonne*) ¦
®® *w* ä (*ran q.ligne* U *ran q.colonne*))
®®(*5*)*A x, y*:*UMLClasse*¥ (*x, y*) e *ran L4* Û
®® (*x*e *AUB* ¦ *y* e *classes*
®® *As, t* :*ran L4* ¥ *s.2*ë *t.2*
®®(*6*)*Ai,j* : *N* ; *a* : *UMLClasse*¥ (*i,j, a*) e *ran L1* Þ
®® *i* ø *#A.classes* ¦ *j*ø *#B.classes*
®Ð_ _
Ð_ _

Un diagramme fusionné est caractérisé par six attributs comme nous pouvons l'observer dans la spécification Object-Z présentée ci-dessus. Les deux premiers attributs, A et B, sont les diagrammes de classes à partir desquels le diagramme intégré est construit. Le troisième attribut, T, est la table de comparaison qui montre le lien entre les classes de A et deB et qui guide leur intégration. Les quatre derniers attributs L1, L2, L3 et L4 sont des listes utilisées pour garder une trace des actions appliquées aux classes de A et de B et les modifications qu'elles ont subies au cours de leur intégration. Ainsi, ces liste sont définies comme suit :

– L1 contient toutes les nouvelles classes obtenues par factorisation de deux classes provenant respectivement des diagrammes A et B. Il s'agit de classes abstraites dont le rôle est d'éviter les redondances dans certaines classes.

– L2 regroupe toutes les classes qui ont été fusionnées en une seule classe. Toute classe ne peut participer qu'une seule fois au plus à une opération de fusion. Cette condition traduite par l'invariant (2) dans la spécification de la classe *DiagrammeFusionné*, prévient la perte d'informations contenues dans les diagrammes *A* et *B* pouvant provenir de la fusion d'une même classe de l'un des diagrammes avec deux classes de l'autre.

– L3 contient toutes les relations de généralisation/spécialisation établies entre des classes de A et de B.

– L4 regroupe des couples de classes représentant les classes qui ont été renommées et leurs copie portant le nouveau nom.

4.3 - Règles d'intégration

Les règles d'intégration proposent des solutions aux problèmes rencontrés au cours de l'intégration de deux diagrammes de classes. Une règle d'intégration s'applique à une paire de classes choisies de diagrammes différents. Elle spécifie une conjonction de conditions dans lesquelles la règle peut être appliquée. Ces conditions concernent le lien entre le nom des classes, leurs structures, leurs méthodes et leurs diagrammes d'états. Plusieurs solutions peuvent être proposées par une règle. Une solution indique comment les classes en question seront ajoutées au diagramme intégré et qu'elles actions ces classes doivent subir.
Afin de retrouver toutes les règles d'intégration des classes possibles, nous avons commencé par inventorier toutes les combinaisons entre les valeurs des liens linguistiques, sémantiques et dynamiques qui peuvent exister ensemble. Puis nous avons défini pour chaque combinaison la ou les solutions qui semblent être cohérentes avec la situation sous-jacente. Nous avons ainsi défini un système de 24 règles. Elles sont définies par des opérations dans la classe *DiagrammeFusionné*. Dans ce qui suit nous donnons un exemple de ces règles.

Exemple

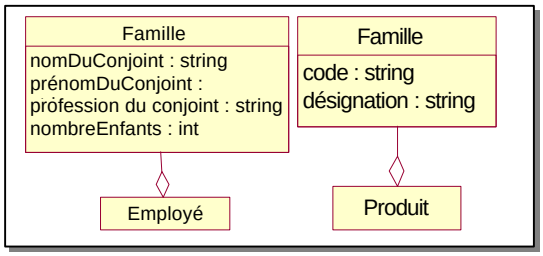


Figure 4. Exemple d'application de la règle 15.

Si deux classes ont des noms synonymes ou identiques mais l'intersection entre leurs structures est faible ou vide alors la solution proposée par la règle 1 est de copier les deux classes telles qu'elles sont dans le diagramme intégré mais il faut renommer l'une d'elles. La définition formelle de cette règle est :
*Règle15*ê *Règle15' ; renommer[c?\a?]*
avec
È*RègleC15'* _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
®D (*classes*)
®*x*? : *Cellule*
®*i*!, *j*!: *N*
®*c*! : *UMLClasse*
®*d*! : *UMLClasse*

Ç_ _
®(1) *x?e T.cellules*
® *x?.contenu.7*
®(2) *x.contenu.1e {IDENT, SYN}* ¦
® *x.contenu.2 e {INTER_FA, DISJ}*
®(3) *c! e copierClasse (self, x?.ref.1)*
® *d! e renommerClasse (self, x?.ref.2)*
® *i! = x?.ref.1*
® *j! = x?.ref.2*
®(4) *classes' = (classes \ {ligne (x?.ref.1),*
® *colonne (x?.ref.2)}) U {c!, d!}*
Ð_ _

Dans la figure 4, nous avons deux classes portant le même nom Famille. L'intersection entre la structure de ces deux classes est vide, ce qui laisse conclure que les classes sont sémantiquement très différents. En appliquant la règle 15, nous obtenons dans le diagramme intégré une copie de chacune de ces classes.

4.4 - Processus d'intégration

Le processus d'intégration de deux RCs A et B est un processus incrémental, c'est à dire qu'il part d'un diagramme de classes vide et le construit par ajout successif de classes de A et de B. Il est composé des étapes suivantes.

Etape1

Elle consiste à retrouver toutes les classes *propres* des diagrammes A et B et à les copier dans le diagramme intégré. Une classe *propre* du diagramme de classes A (respectivement B) est une classe qui n'a aucun lien sémantique avec les classes du diagramme B (respectivement A). Elle est reconnue par l'existence d'un score nul dans toutes les cellules appartenant à la ligne (respectivement la colonne) dont elle constitue l'entête.

Etape 2

Rappelons d'abord qu'une partition dans la table de comparaison correspond à deux hiérarchies d'héritage appartenant chacune à un diagramme de classes des RCs d'origine. La deuxième étape de notre processus consiste à marquer dans chaque partition la cellule de chaque ligne de la partition contenant le maximum des scores (ce maximum doit être différent de 0). La classe de l'entête de colonne correspondant à cette cellule constitue, à l'égard de la classe de l'entête de ligne correspondant à la même cellule, la classe qui lui est sémantiquement la plus proche dans toute la hiérarchie d'héritage formant la ligne de la partition. Si deux cellules contiennent un maximum une seule sera choisie arbitrairement.

Pour chaque cellule marquée, nous devons sélectionner, dans l'ensemble des règles de la première catégorie, celle dont la pré-condition correspond au contenu de la cellule. L'utilisateur a alors le choix entre plusieurs solutions : la (ou) les solution(s) proposé(es) par la règle dont la pré-condition coï ncide avec le contenu de la cellule traitée, ou la solution préconisée par la règle 16 et qui permet de copier les classes dans le diagramme intégré sans aucune modification.

Ainsi, pour chaque cellule marquée une solution est choisie, puis la cellule est démarquée et la table de comparaison est mise à jour.

Etape 3

Elle consiste à marquer les cellules dont le lien linguistique entre les noms des classes correspondantes est une composition. Il est donc possible de créer un lien d'agrégation entre ces classes.

Etape 4

Dans cette étape, toutes les classes des représentations conceptuelles A et B qui ne sont pas encore ajoutées sont intégrées. Les listes L2, L3 et L4 sont consultées pour trouver ces classes.

Etape 5

Tous les liens de généralisation/spécialisation des diagrammes d'origine, ainsi que ceux se trouvant dans les liste L1 et L3 sont ajoutés au diagramme en cours de construction.

Etape 6

Dans cette étape nous devons ajouter toutes les associations et les classes d'associations au diagramme résultat. Les associations dont les pattes ne sont pas attachée à des classes d'associations sont les premières à intégrer. Puis, toutes les classes d'association sont ajoutées.

Enfin, nous intégrons les associations restantes.

Bien entendu, il est possible de rencontrer certains conflits pendant cette intégration tels que les conflits de noms ou les conflit de cardinalité. Ces conflits seront résolus grâce aux règles d'intégration appropriées.

Notons que le processus de fusion décrit ci-dessus reçoit en entrée deux RC uniquement. Pour l'étendre à un nombre quelconque de RC, nous adoptons la stratégie à échelle étudiée entre autres dans Batini (1986). Elle consiste à fusionner d'abord deux RC. Le résultat de cette première fusion est à son tour fusionné avec une troisième RC et ainsi de suite, jusqu'à ce que toutes les RC soient fusionnées. Cette stratégie peut être résumée par la figure 5.

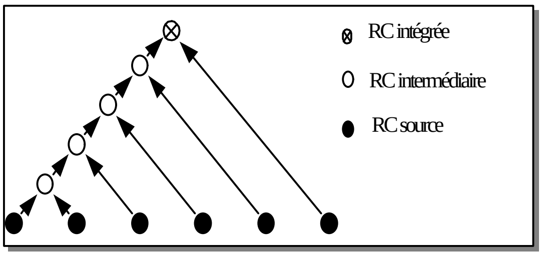


Figure 5. Stratégie à échelle.

Il est facile de constater que si le nombre de RC est égal à n, alors le nombre d'intégrations à effectuer est n-1 intégrations. Les deux RC avec lesquelles nous débutons l'intégration et l'ordre d'intégration

des autres RC sont définis par l'utilisateur. Ces choix peuvent être guidés par l'outil d'intégration en se basant sur le nombre de noms de classes qu'elles ont en commun. Les RC ayant un nombre élevé de classes en commun sont les premières à intégrer.

5 - CONCLUSION

Dans cet article, nous avons présenté une approche pour la fusion de représentations conceptuelles qui s'appuie sur Object-Z, un langage de spécification formel puissant et bien adapté à notre problématique.
Les travaux futurs concernent l'implémentation d'un outil de fusion et l'intégration de cet outil dans un méta-outil de conception. De plus certains aspects doivent être pris en compte dans nos prochains travaux comme par exemple l'intégration des contraintes.

BIBLIOGRAPHIE

Batini, C., Lenzerini, M., Navathe, S., A Comparative analysis of methodologies for database schema integration. ACM Computing surveys, vol.18 n°4, 1986.

Elkoutbi, M. (2000), "Ingénierie des interfaces usager à l'aide du prototypage et des méthodes formelles", *Ph.D. en informatique de la faculté des arts et des sciences, Université de Montréal, Canada*.

Gargouri, F., Haddar, N., Ducateau, C.F., Gargouri, W., « An integrated modelling approach for complex applications and distributed information systems». *International Journal of Production Economics* (64). 331-344. 2000.

Haddar, N., Gargouri, F., Ben Hamadou, A., «Model integration: the comparison step». *International conference of the UK Academy for Information System. Leeds Metropolitan University Leeds-UK. Avril 2002*.

Haddar, N., Gargouri, F., Ben Hamadou, A., « Integration of Object-Z class diagrams specifications». *Proc of IEEE Conference System Man Cybernetics, Hammamet, Tunisie, 6-9 Octobre, 2002*.

Harmsen, A. F., *Situational Method Engineering*. Moret Ernst & Young , 1997.

Kim, S.K., Carrington, D.(2000) A formal mapping between UML models and Object-Z specifications. Software verification center, departement of computer science, the university of Queensland, Australia. Technical report 00-03 January 2000.

OMG (2001), Meta-Object Facility (MOF) 1.3.1 Specification, Object Management Group, Document formal/2001-11-02.

Parent, C. and Spaccapietra, S., (1998) Issues and approaches of database integration. CACM, vol 41, n° 5, pp166–178.

Semmak F.(1998) Réutilisation de composants de domaine dans la conception des systèmes d'information, thèse de doctorat de l'université Paris I.

Smith, G., *The object-Z specification language*, Kluwer academic publishers, 2000.

Rational (2003) Rational Inc. Documentation UML. *http://www.rational.com.*

COMPRESSION DES IMAGES MEDICALES FIXES PAR RESEAU DE NEURONES

Nacéra Benamrane,
Maître de conférences en Informatique
nabenamrane@yahoo.com, + 213 41 41 53 22

Zakaria Benahmed Daho
Etudiant doctorant
z-daho@lycos.com

Jun Shen,
Professeur en Informatique
shen@egid.u-bordeaux.fr +33 5 57 12 10 26

Adresse professionnelle
Université des Sciences et de la Technologie d'Oran, Départ d'Informatique ★ BP 1505 ★
EL-Mnaouer 31000 Oran, Algérie

Résumé : Dans cet article, nous proposons une technique de compression des images fixes basée sur le réseau de Kohonen. Pour palier à l'effet de blocs « ou de pixellisation » causé par les taux de compressions élevés, les blocs de l'image à compresser sont classifiés, selon leurs degrés d'activités, avant de les présenter au réseau de Kohonen pour une quantification vectorielle. Cette classification réalise une discrimination entre les blocs de faibles activités et les blocs contours de hautes activités. Les blocs homogènes et les blocs contours sont codés séparément en utilisant des codebooks différents. Par cette technique, nous avons obtenu une amélioration sensible de la qualité visuelle des images médicales reconstruites tout en maintenant un taux de compression élevé

Summary: The image blocks to be compressed are classified according to their activity degree before presenting them to Kohonen's network for a vector quantization. This classification allows discrimination between the blocks with high activity (edge blocks) and blocks with law activity. The blocks of high activity are divided into small blocks of 2x2 pixels. Blocks of high and law activity are coded separately with different codebooks. We have obtained a noticeable improvement of visual quality of all the rebuild medical images while keeping an important compression rate.

Mots clés : Compression, Images médicales, Réseau de Kohonen, Quantification vectorielle, Classification.

Compression des images médicales fixes par réseau de neurones

Dans de nombreux domaines, l'image numérisée remplace les images analogiques classiques. Dans le domaine médical, l'utilisation des images radiographies, ultrasonores, IRM, … pose un grand problème de stockage et d'archivage. Par exemple ; un hôpital de 200 lits, produit en moyenne chaque année 875 Go de données images. En plus du problème de stockage, si de telles images doivent être transmises via un réseau, la durée de la transmission est souvent trop longue. Pour palier à tous ces problèmes, la compression de ces images devient une opération nécessaire et impérative. Le but principal de la compression des images est de réduire la quantité de bits nécessaire pour les décrire tout en gardant un aspect visuel acceptable des images reconstruites.

La compression des images fixes a été réalisée par plusieurs techniques parmi les plus connues : le JPEG qui est une méthode avec perte standardisé par ISO en août 1990, son principe est détaillé par Guitter (1995), le codage en sous bandes proposé par Woods (86) et Nezit-Gerek (2000), la transformée en Ondelette proposée par Mallat (1989), le JPEG 2000 qui est une nouvelle norme développée par ISO en 2000, son principe est détaillé par Ordóñez et al(2003) ect … Dans le contexte particulier de l'imagerie médicale, Le duff a proposé une méthode utilisant séparément l'ondelettes et le JPEG pour la compression de séquences spatiales d'IRM cérébrales, les résultats obtenus par ondelettes sont supérieurs en terme de qualité des images reconstruites. Ordóñez et al (2003) ont proposé une technique basée sur le JPEG 2000 pour compresser des images tumeurs de cerveau en vue d'une indexation basée sur les informations spectrale et spatiale.

Ces méthodes effectuent la compression en réalisant une quantification scalaire *(QS)* sur les valeurs obtenues après transformations. L'inconvénient de la quantification scalaire est qu'elle ne permet pas d'exploiter la corrélation spatiale qui existe entre les différents pixels de l'image. Une autre façon plus intéressante pour réaliser la compression est de coder non pas les valeurs individuellement les unes après les autres, mais de coder un ensemble de valeurs simultanément. Cette manière de procéder est appelée «quantification vectorielle » *(QV)*.

La quantification vectorielle (*Vector Quantization*) a été utilisée avec succès pour le codage du signal vocal par Linde (80) ainsi que pour la compression des images fixes par Nasrabasi (1988), Gerso (1982) et Ramamurthi (1986). Pour réaliser la quantification vectorielle plusieurs systèmes ont été proposés par Ramamurthi (1986) et Gray (1984). Ces systèmes se basent sur l'algorithme de Lynd, Buzo et Gray (1980).

Les approches utilisant les réseaux de neurones artificiels pour le traitement intelligent des données semblent être très prometteuses, ceci est essentiellement dû à leurs structures offrant des possibilités de calculs parallèles ainsi que l'utilisation du processus d'apprentissage permettant au réseau de s'adapter sur les données à traiter. Les nouvelles techniques basées sur les réseaux de neurones comme outils de compression ont été proposées par Jiang (1999), Robert (1995) et Stanley et al (1990). Le réseau de Kohonen (1990) est un réseau de neurones particulier, il peut être utilisé comme quantificateur vectoriel pour les images.

Cet article traite la méthode de compression basée sur le réseau de neurones de Kohonen pour des images médicales. Ce papier comprend quatre sections. Dans la section 2, le réseau Kohonen est présenté ainsi que son utilisation pour réaliser la quantification vectorielle. Dans la section 3, il sera présenté les résultats de compression obtenus sur des images médicales. Le problème produit par des blocs de taille importante sera aussi mis en évidence. Une amélioration de la méthode sera proposée dans la section 4. Enfin, la dernière section est consacrée pour la conclusion et les perspectives.

1 - QV PAR LE RESEAU DE KOHONEN

Le réseau de Kohonen (1990) (le *Self-Organizing Feature Map*) est un réseau à compétition (Jodouin, 1994). Ce réseau est composé d'une couche de neurones qui reflète passivement les données d'entrée présentées au réseau, et d'une carte topologique. Cette dernière est composée d'un certain nombre de neurones organisés selon une structure bien définie. Cette structuration lie les neurones ensemble en une surface « élastique », et les

contraint à respecter un certain comportement lors de l'apprentissage.
Lorsqu'une entrée est présentée au réseau via la couche d'entrée, les neurones de la carte topologique s'activent différemment et une compétition s'installe entre ces neurones. Le neurone dont le vecteur des poids synaptiques se rapprochant le plus de l'entrée est déclaré vainqueur de la compétition.
Le réseau de Kohonen a été utilisé dans plusieurs applications ; par exemple : la reconnaissance des formes, traitement acoustique du signal vocal, problèmes de classifications par Kohonen (1990) et la compression par Lebail (1989) et Nasrabadi (1988). Pour réaliser la quantification vectorielle, un dictionnaire de données (Codebook) est nécessaire (Figure 1).

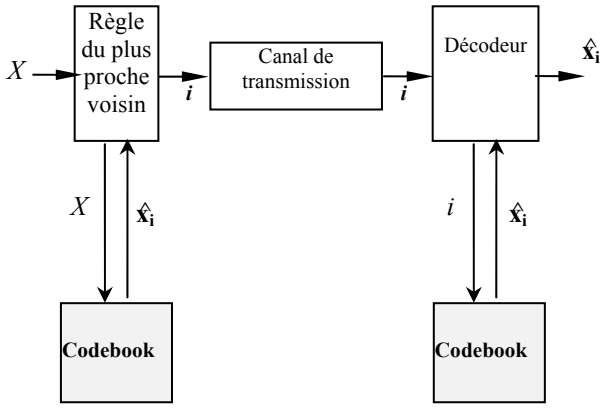


Figure 1. Système de compression Par QV

La quantification vectorielle du vecteur d'entrée X produira en sortie l'indice i de son meilleur représentant $\hat{x}_i$ choisi dans le Codebook suivant la règle du plus proche voisin (Nearest neighbor rule) :

$\hat{x}_i$ est le meilleur représentant de X $\Leftrightarrow$ $d(X, \hat{x}_i) \leq d(X, \hat{x}_j)$; $\forall$ i, j [1]

avec d est une mesure de distance.

Plusieurs recherches ont montré que l'utilisation du réseau de Kohonen permet de générer un Codebook assurant une meilleure représentation des données à traiter, ce qui entraîne une diminution du bruit de quantification.
Le réseau de Kohonen effectue une projection de l'espace des vecteurs d'entrées dans un sous-ensemble fini de vecteurs de représentants (le Codebook), produisant ainsi un Codebook localement optimal. Kohonen (1990) a développé un algorithme dans lequel les poids synaptiques w_i du réseau reflètent, enfin du processus d'apprentissage, le contenu des vecteurs des représentants. L'adaptation des poids du réseau se fait selon les règles suivantes :

1) trouver le neurone c, vainqueur de la compétition tel que :

$$d(X,w_c) \leq d(X,w_i) \; ; \; \forall \; i \; ; \qquad [2]$$

2) mettre à jour les poids w_i du réseau :

$$w_i(t+1)=w_i(t)+h(c,i,t)*[X-w_i(t)] \quad [3]$$

où $w_i(t)$ est le vecteur poids du neurone *i* à l'instant *t*, *h* est une fonction définie par :

$$h(c,i,t) = \begin{cases} \alpha(t) \text{ si } i \in N(c,t) \; ; \\ 0 \quad \text{sinon} \end{cases} \qquad [4]$$

avec $\alpha(t) \in [0,1]$

La fonction *h* définit l'ampleur de la correction apportée au neurone vainqueur *c* ainsi qu'a son voisinage. Le voisinage, à l'instant *t*, du neurone vainqueur *c* est déterminé par la fonction *N* qui est une fonction décroissante par rapport au temps. Le voisinage final d'un neurone se compose du neurone lui-même. La règle [4] permet d'attribuer la même correction $\alpha(t)$ à tous les neurones appartenant au voisinage du neurone vainqueur à un instant *t*. Une autre règle peut être utilisée stipulant que plus un neurone est loin du vainqueur, plus faible sera sa correction. Dans ce cas, le pas d'apprentissage sera en fonction du temps *t* et de la distance entre les neurones de la carte topologique.
Kohonen a démontré l'efficacité de son réseau dans la reconnaissance et la compression du signal vocal. Dans la section qui suit sont exposés les résultats de compression obtenus sur des images médicales.

2 - RESULTATS EXPERIMENTAUX ET DISCUSSIONS

Les images utilisées dans le processus d'apprentissage sont de type médical de format 256 x 256 pixels. Chaque pixel est codé sur 8 bits. La figure 2, montre une des images utilisées pour l'apprentissage.

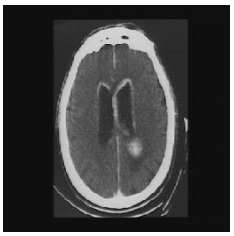


Figure 2. *Une image utilisée dans l'apprentissage*

Lors de l'entraînement, les images sont découpées en blocks carrés qui constituent les vecteurs d'entrées. Après une opération de normalisation, les blocs sont utilisés pour la génération du Codebook suivant les règles [2], [3] et [4].
La qualité des images reconstruite est évaluée par le PSNR (Peak Signal Noise Ratio) et par l'EQM (Erreur Quadratique Moyenne) :

$$EQM=\frac{1}{T}\sum_{i=1}^{T}(\hat{n}_i-n_i)^2 \qquad [5]$$

$$PSNR=10.log_{10}\left(\frac{255^2}{EQM}\right)(\text{en dB}) \qquad [6]$$

avec T : taille de l'image, n_i : i ème pixel de l'image originale, $\hat{n}_i$: i ème pixel de l'image reconstruite.
Les images de tests utilisées sont présentées ci-dessous :

Image 1 Image 2 Image 3

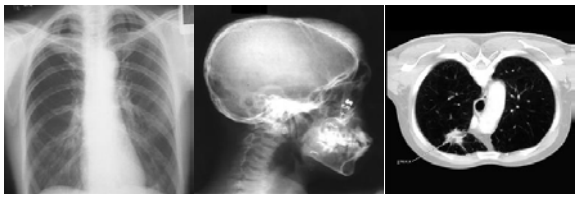


Figure 3. *Les images utilisées dans les tests.*

2.1 - Taille des vecteurs d'entrées égale à 4 pixels avec une carte topologique de 256 neurones

Dans cette expérience, une carte topologique carrée de 16 x 16 neurones a été utilisée. Chaque image à compresser est découpée en blocks non recouvrant de taille 2 x 2 pixels. Dans les figures 4, 5 et 6, les images reconstruites ainsi que les images différences sont présentées.

La qualité des images reconstruites est très bonne, néanmoins, les images différences montrent que les erreurs de codage sont situées principalement aux contours des images (zones en gris clair). Dans le tableau 1 est présenté le résultat de l'évaluation du *PSNR* [6], *EQM* [5], du Taux de Compression (*TC*) et du Nombre de Bits par Pixels (*bit rate*) (*NBP*) :

$$NBP=\frac{log_2(\text{nombre de neurones dans la carte})}{\text{taille des vecteurs d'entrées}}$$

[7]

$$TC=100*(1-(NBP/8)) \ (\text{en \%}) \qquad [8]$$

Pour un nombre de bits par pixel égal à 2 (taux de compression égal à 75 %), le *PSNR* est supérieure à 30 dB pour l'ensemble des images de tests.

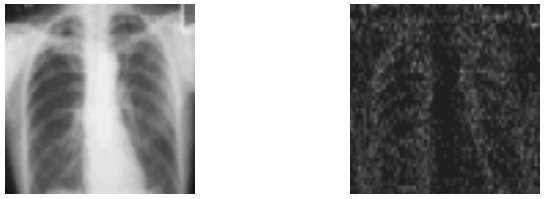


Figure 4. *Image 1 reconstruite et l'image différence.*

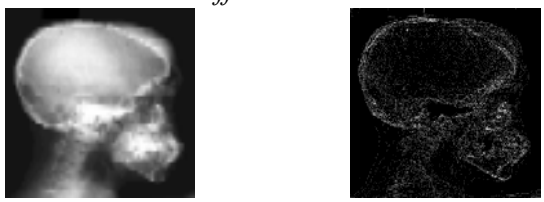


Figure 5. *Image 2 reconstruite et l'image différence*

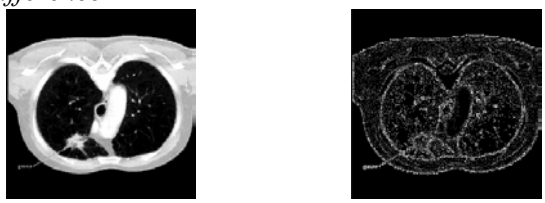


Figure 6. *Image 3 reconstruite et l'image différence*

	PSNR (dB)	***EQM***	***TC %***	***NBP bits/pixel***
Image 1	36.21	15.52	75	2
Image 2	35.36	18.90	75	2
Image 3	31.08	50.71	75	2

Tableau 1. *Résultats de compressions.*

Dans l'expérience qui suit, la taille des vecteurs d'entrées est augmentée, ceci permet d'obtenir des taux de compression plus importants.

2.2 Taille des vecteurs d'entrées égale à 16 pixels

Dans cette expérience, les images à compresser sont découpées en blocks non recouvrant de taille 4 x 4 pixels, ceci permet d'obtenir un taux de compression égale à 93,75 % (0.5 bits / pixel) avec une carte de 256 neurones. Cette augmentation intéressante du taux de compression est suivie par une dégradation assez importante de la qualité des images reconstruites. La Figure 7, montre l'image n°3 reconstruite où l'effet de blocs apparaît clairement. Le *PSNR* obtenu est égal à 24.79 dB ce qui correspond à une perte de 6.29 dB par rapport au premier test.
Les résultats obtenus pour les trois images de tests sont présentés dans la tableau qui suit :

	PSNR (dB)	***EQM***	***TC %***	***NBP bits/pixel***
Image 1	32.03	40.72	93.75	0.5
Image 2	30.03	64.43	93.75	0.5
Image 3	24.79	215.74	93.75	0.5

Tableau 2. *Résultats de compressions. Cas des vecteurs d'entrées de 16 pixels avec une carte de 16 x 16 neurones.*

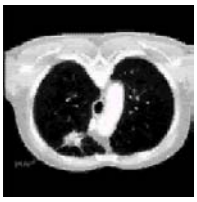


Figure 7. *Image 3 reconstruite.*

Le tableau 2 montre qu'en utilisant des vecteurs de dimensions 16, un gain important de 1.5 bits par pixel peut être atteint. Mais, en contre partie, une baisse moyenne de 5.26 dB est enregistrée dans la qualité des images reconstruites.
Donc, les performances (en terme de qualité des images) du système de compression par le réseau de Kohonen diminuent quand la taille des blocs d'entrées augmente, l'effet de bloc devenant en outre de plus en plus sensible surtout au niveau des contours de l'image. Cette baisse de qualité peut très bien être expliquée ; plus la taille des blocs est importante, plus leurs contenus sont variés et donc les chances de trouver des blocs fortement ressemblants diminuent.
Une première tentative pour résoudre ce problème consiste à augmenter la taille de la carte topologique. Dans le test qui suit, la taille de la carte topologique est doublée à 512 neurones (tableau 3).
Les résultats obtenus montrent que même en augmentant la taille de la carte topologique à 512 neurones l'effet de blocs persiste. Le *PSNR* augmente de 0.52 dB en moyenne.

	PSNR (dB)	***EQM***	***TC (%)***	***NBP bits/pixel***
Image 1	32.85	33.70	92.96	0.5625
Image 2	30.67	55.68	92.96	0.5625
Image 3	24.91	209.87	92.96	0.5625

Tableau 3. *Résultats de compressions. Cas des vecteurs d'entrées de 16 pixels avec une carte de 16 x 32 neurones.*

Pour déterminer des solutions efficaces à ce problème, examinons le *Codebook* obtenu dans le cas des blocs de 16 pixels avec une carte de 256 neurones.

Figure 8. *Codebook obtenu par un réseau de Kohonen de 256 neurones dans la carte avec des blocs de taille 4 x 4.*

Dans la figure 8 est observé que les blocs contours sont faiblement représentés contrairement aux blocks homogènes. Ces derniers occupent une grande partie du *Codebook*. Ainsi, la cause effective de l'effet de blocs peut être expliqué par le fait qu'il y a un nombre insuffisant de vecteurs dans le *Codebook* permettant de représenter les différents blocs contours contenus dans les images aussi, la mesure de distance *d* utilisée, pour déterminer le représentant du vecteur d'entrée, ne peut garantir qu'un bloc contour

aura comme représentant un vecteur contour dans le *Codebook*. Par conséquent, le *Codebook* ne peut coder avec une bonne qualité, les différentes variétés des blocs contours que les images peuvent contenir.
La qualité des contours est très importante pour assurer une bonne qualité visuelle des images reconstruites. En plus, dans le cas de l'image médicale, les images décodées doivent être de très bonnes qualités pour éliminer le risque d'un diagnostic médical erroné dû à une image de mauvaise qualité visuelle. Une solution à ce problème consiste à créer plusieurs *Codebooks*, chacun étant spécialisé dans la représentation d'un certain type de blocs dans l'image.

3 – COMPRESSION DES BLOCS AVEC CLASSIFICATION

3.1- Taille des blocs égale à 16 pixels

Dans cette série d'expériences, lors de l'entraînement, les blocs des images utilisées sont classifiés selon la nature des détails contenus dans chaque bloc. Ainsi, plusieurs Codebooks seront créés par ce procédé. Chaque Codebook sera spécialisé dans le codage d'un type bien défini de blocs. Pour réaliser la classification d'un bloc, plusieurs manières existent ; par exemple : Gersho (1982), Ramamurthi (1986)] et Le bail [1982] utilisèrent une classification basée sur les directions du gradient. Nansrabadi (1988) a utilisé la valeur de la variance pour distinguer entre deux classes de blocs. Dans nos expériences réalisées, les blocs sont classés selon leur degré d'activité. Le degré d'activité d'un bloc B est mesuré grâce à la fonction d'activité A_b définie par :

$$A_b(B)=\sum_{m,n} A_p(x_{m,n});\ x_{m,n}\in B \qquad [9]$$

avec $x_{m,n}$: pixel appartenant au bloc B ;
A_p : fonction d'activité du pixel $x_{m,n}$;

$$A_p(x_{m,n})=\sum_{i=-1}^{+1}\sum_{j=-1}^{+1}(x_{m,n}-x_{m-i,n-j})^2 \qquad [10]$$

La fonction d'activité a pour objet l'identification et la discrimination entre deux classes de blocs ; les blocs de fortes activités et les blocs de faibles activités. Ceux qui appartiennent à la classe de hautes activités sont subdivisés encore en quatre classes selon l'orientation des détails contenus dans ces blocs. Dans un bloc donné, les orientations de la structure des détails sont : horizontales *(h)*, verticales *(v)*, et les deux orientations diagonales *(d* et *e)*. Pour définir ces orientations, des fonctions d'orientations sont utilisées :

$$B_h=\frac{1}{M(M-1)}\cdot\sum_{m=1}^{M}\sum_{n=1}^{M-1}(x_{m,n}-x_{m,n+1})^2 \qquad [11]$$

$$B_v=\frac{1}{M(M-1)}\cdot\sum_{m=1}^{M-1}\sum_{n=1}^{M}(x_{m,n}-x_{m+1,n})^2 \qquad [12]$$

$$B_d=\frac{1}{(M-1)^2}\cdot\sum_{m=1}^{M-1}\sum_{n=1}^{M-1}(x_{m,n}-x_{m+1,n+1})^2 \qquad [13]$$

$$B_e=\frac{1}{(M-1)^2}\cdot\sum_{m=2}^{M}\sum_{n=1}^{M-1}(x_{m,n}-x_{m-1,n+1})^2 \qquad [14]$$

avec : $M = 4$; et $x_{n,m}$: pixel du bloc B;
n et m : la position du pixel dans le bloc.

Chacune des fonctions d'orientations définit une classe de blocs. La classification est réalisée de la manière suivante :

- Pour chaque bloc appartenant à la classe haute activité, calculer les valeurs des quatre fonctions d'orientation B_h, B_v, B_d et B_e.
- La plus petite valeur parmi les quatre fonctions détermine l'orientation du bloc.
  Une fois les cinq *Codebooks* construits, le processus de compression pourra être entamé. Les étapes de la compression d'un bloc sont les suivantes (Figure.9) :
- L'activité du bloc X à compresser est calculée selon [9] ;
- Déterminer si le bloc a une forte ou faible activité (selon un certain seuil à déterminer) ;
- Si le bloc appartient à la classe de haute activité, déterminer l'orientation des contours dans le bloc selon [11], [12], [13] et [14] ;
- Déterminer le *Codebook* à utiliser pour le codage.
- Envoyer au décodeur l'indice du meilleur représentant du bloc X.

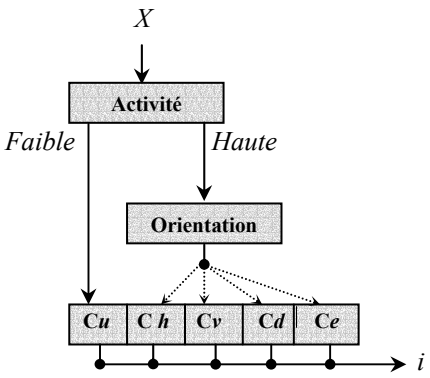


Figure 9. *Système de compression par classification.*

C*h* :*Codebook* des blocs d'orientation horizontale ;
C*v* :*Codebook* des blocs d'orientation verticale ;
C*d* :*Codebook* des blocs d'orientation diagonale selon la première bissectrice;
C*e* :*Codebook* des blocs d'orientation diagonale selon la seconde bissectrice;
C*u* :*Codebook* des blocs de faibles activés.
Les résultats de compression en utilisant la classification sont présentés dans le tableau 4. En comparant les tableaux 4 et 2, on remarque que pour le même taux de compression (93.75 % et 0.5 bits / pixel), la qualité des images compressées par classification est assez meilleur. Un gain moyen de 0.58 dB est obtenu sur les images décomposées.

	PSNR (dB)	***EQM***	***TC (%)***	***NBP bits/pixel***
Image 1	32.41	37.31	93.75	0.5
Image 2	30.68	55.58	93.75	0.5
Image 3	25.51	182.52	93.75	0.5

Tableau 4. *Résultats de compressions par classification. Cas des vecteurs d'entrées de 16 pixels avec cinq Codebooks de taille 256 représentants.*

En utilisant cette technique, les différents tests réalisés sur les images montrent que l'effet de blocs, observé sur les contours des images, est réduit mais il n'est pas tout à fait éliminé. Ceci est expliqué par le fait de répartir les blocs en cinq classes est insuffisant. Les blocs de taille 16 pixels sont très variés et nécessitent alors un nombre élevé de classes.

Pour résoudre ce problème, tout en gardant une complexité raisonnable de la technique de compression, nous proposons d'ajouter une classification sur les blocs contours après les avoir découpés en petits blocs de taille 2 x 2 pixels.

3.2- Classification avec blocs à compresser de taille variable, contours de tailles 4 pixels et blocs homogènes de tailles 16 pixels

La technique que nous proposons maintenant se base sur les observations faites dans plusieurs séries de tests. En effet, nous avons constaté que :

1. La *QV* réalisée en utilisant des blocs de tailles 4 pixels assure une bonne reconstruction des images. Les erreurs de codage se situent généralement sur les contours des images reconstituées ;
2. Le taux de compression assuré par les blocs de 16 pixels est très intéressant néanmoins, l'effet de blocs est très visible sur les contours des images décompressées ;
3. La classification (en cinq classes) réalisée sur les blocs de taille 16 pixels n'augmente pas de manière considérable la qualité des images reconstruites ;
4. Le taux de compression et la qualité des images sont beaucoup plus influencés par la variation de la taille des blocs que par la taille des *Codebooks* utilisés.

Pour assurer un taux de compression assez élevé et une bonne qualité visuelle des images, nous proposons donc la technique suivante (figure 10) :

- Découper l'image en blocs de taille 4 x 4 pixels ;
- Déterminer l'activité du bloc à compresser X ;
- Si le bloc X appartient à la classe de haute activité, le subdiviser en 4 sous blocs (Y_k) de taille 2 x 2 pixels ;
- Déterminer l'activité pour chaque sous bloc Y_k crée ;
- Si le sous bloc appartient à la classe de haute activité, déterminer l'orientation des contours contenant le sous bloc ;
- Déterminer le *Codebook* à utiliser pour le codage ;
- Envoyer au décodeur l'indice du meilleur représentant du bloc.

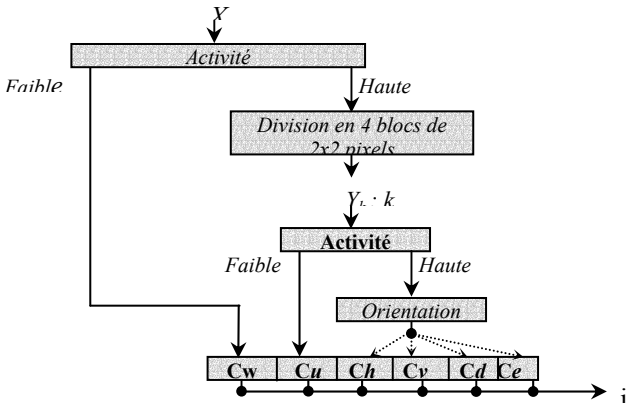


Figure 10. *Le principe de la méthode proposée*

Cw : Codebook pour quantifier les blocs 4 x 4 de faible activité ;
Cu : Codebook pour quantifier les blocs 2 x 2 de faible activité ;
Ch, Cv, Cd, Ce : Codebooks pour quantifier les blocs 2 x 2 de hautes activités (blocs contours).
Les résultats obtenus sur les images 1, 2 et 3 sont :

	PSNR (dB)	***EQM***	***TC (%)***	***NBP bits/pixel***
Image 1	34,43	23.40	90,14	0,78
Image 2	33,61	28,29	91,20	0,70
Image 3	29,64	70,57	91,91	0,64

Tableau 5. *Résultats de compressions par la méthode proposée.*

Ces résultats on été obtenus en utilisant six c*odebooks*. Le *codebook* associé aux blocs de 16 pixels comporte 256 neurones assurant ainsi une bonne fidélité des blocs homogènes. Les blocs de petite taille sont peu variants (par rapport aux blocs plus grands), des petits *codebooks* sont alors suffisants pour les coder, nous avons utilisés des *codebooks* de 64 neurones. L'image 3 reconstruite est présentée en figure 11.

Figure 11. *Image 3 reconstruite.*

La technique proposée permet d'augmenter la qualité des images reconstruites tout en gardant un taux de compression assez élevé. Pour l'image 3, par exemple ; un PSNR égal à 29.64 dB est obtenu avec un codage sur 0.64 bits/pixel (Figure 11). En comparant les résultats obtenus avec la même image dans le tableau 2, un gain dans le PSNR égal à 4.85 dB en plus, l'effet de blocs est éliminé (comparer la figure 11 avec la figure 7).

4 - CONCLUSION

Dans cet article, une technique de compression des images médicales fixes est présentée. L'utilisation des blocs d'images de tailles 16 pixels permet de réaliser un codage à 0.5 bits par pixel. L'augmentation du taux de compression est suivie par une dégradation visuelle de la qualité des images décodées et ceci est essentiellement dû à l'effet de blocs. Pour palier à cet inconvénient, une classification des blocs a été réalisée. En affinant la compression des blocs contours, en créant des sous blocs plus petits qui subissent une pré- classification avant de les quantifier, les contours des images reconstruites sont d'une très bonnes qualités. L'utilisation des blocs homogènes de taille 16 pixels permet d'augmenter le taux de compression.
L'intégration de la quantification vectorielle par le réseau de Kohonen dans un système de compression classique (par Ondelettes par exemple) permettra de tirer avantage de la simplicité et des performances du réseau pour diminuer le bruit de quantification et d'augmenter la qualité des images reconstruites.

Remerciements
Nous témoignons notre reconnaissance aux aides financières dont nous avons bénéficié dans le cadre du projet PNR-ANDRS (agence nationale de recherche en médecine) n°05/04/01/98/071.

BIBLIOGRAPHIE

Le bail, E., Mitche, A. (1989), " Quantification vectorielle d'images par le réseau neuronal de kohonen ", Traitement du signal, Vol. 6, N° 6, 1989.

Le duff, A., " La compression d'images : une nécessité pour l'archivage et la

télémédecine",http://www.eseo.fr/~aleduff /Articles/Pria/article_pria.htm.

Gersho, A., Ramamurthi, B. (1982), *image coding using vector quantization.*

Gray, R. (1984), "Vector quantization ", *IEEE ASSP Magazine*.

Guitter, H. (1995), *La Compression des images numériques* , Edition Hermes, Paris

Jiang, J. (1999), " Image compression with neural networks – A survey", *Signal processing : image communication n°14*, pp 737-760.

Jodouin, J.F. (1994), *Les réseaux neuromimétiques*, Edition Hermes

Kohonen, T. (1990), " The Self-organizing map ", *Proceedings of the IEEE, vol. 78, n° 9.*

Linde, Y., Buzo, A., Gray, R. (1980), ". An algorithm for vector quantizer design ", *IEEE Transactions on Communications* , vol com-28, n° 1.

Mallat, S. G. (1989), "A theory of multiresolution signal decomposition : The Wavelt representation", *in IEEE Transactions Pattern Analysis and Machine Intelligence*, vol 11, n° 7..

Nasrabadi, M.N., Feng, Y. (1988), " Vector Quantisation of Images Based Upon the Kohonen Self-Organizing Feature Map. ", in *IEEE International Conference on Neural Networks, San Diego, California*, pp101-108.

Nasrabasi, M.N ., King, R.A. (1988), " Image Coding Using Vector Quantization *: A* Review", in *IEEE Transactions on Communications*, vol 36, n° 8.

Nezit-Gerek, O., Enis Cetin, A. (2000), " Adaptive polyphase subband decomposition structures for image compression ", in *IEEE Transactions on Image Processing* , vol 9, n°10.

Ordóňez, J.R., Gazuguel, G, Puentes, J., Solaiman, B., Roux, C. (2003), " Indexation d'images médicales basée sur les informations spectrale et spatiale extraites de JPEG-2000 ", *Actes de CORESA'03, Lyon.*

Ramamurthi, B., Gersho, A. (1986), " Classified vector quantization of images ", in *IEEE Transactions on Communications* , vol com-34, n° 11.

Robert, D. D., Haykin, S. (1995), "Neural network approaches to images compression ", in . *Proceedings of the IEEE*, vol 82, n° 2..

Stanley, C. and al. (1990), " Competitive learning algorithms for vector quantization ", in *Neural networks,* vol 3, pp277-290.

Woods, J.W., O'neil, S.D. (1986), " Subband Coding ", in *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol 34, n° 5.

LES CONFLITS ENTRE LES CONTRAINTES DANS LES SCHEMAS CONCEPTUELS DE BASES DE DONNEES : UML – EER

Faouzi BOUFARES
Maître de conférences en informatique
boufares@lipn.univ-paris13.fr +33 1.49.40.40.71

Djamel BERRABAH
Doctorant en informatique
berrabah@math-info.univ-paris5.fr +33 1.42.86.40.42

Charles-François DUCATEAU
Professeur des universités en informatique
ducateau@iut.univ-paris5.fr +33 1.44.14.45.51

Faiez GARGOURI
Maître de conférences en informatique
faiez.gargouri@fsegs.rnu.tn

Résumé : Le travail de modélisation des données est toujours une activité délicate et nécessite une bonne expérience des concepteurs. Plusieurs formalismes permettent de modéliser les données tels que UML et EER. Afin de préserver la sémantique du monde réel, plusieurs catégories de contraintes peuvent être introduites dans ces deux formalismes telles que les contraintes de multiplicité (cardinalité) et les contraintes d'intégrité fonctionnelle. Notre objectif dans ce papier est de traiter la cohérence globale de ces contraintes. Notre méthode consiste à faire correspondre au schéma conceptuel un système linéaire d'inégalités à résoudre en utilisant les techniques de la programmation linéaire. A partir de cette résolution, nous détectons et localisons les conflits engendrés par les contraintes.

Summary: Data modeling is a delicate activity and requires that modelers have a good experience. Several formalisms, such as UML and EER, allow to model data. In order to preserve the semantics of the real world, many categories of constraints have been introduced into these two formalisms. Multiplicity (cardinality) constraints and functional integrity constraints are two examples of such constraints. Our aim in this paper is to treat the total coherence of these constraints. Our method consists in corresponding to the conceptual diagram a linear system of inequalities to solve by using the linear programming techniques. From this resolution, we detect and locate the conflicts generated by the constraints.

Mots-clés : Base de Données, Modélisation UML/ER, diagramme de classes/ER valide, Contraintes de multiplicité/cardinalité, Contraintes d'intégrité fonctionnelle, Outil d'Aide à la Conception, Programmation linéaire, Qualité des données.

Les conflits entre les contraintes dans les schémas conceptuels de Bases de Données : UML – EER

1 - INTRODUCTION

La tendance des recherches en cours est de permettre le partage des connaissances et des données entre les communautés, les utilisateurs et les applications. On peut donc être amené à intégrer des systèmes d'information et des bases de données. Parmi les problèmes qui peuvent surgir, nous pouvons citer par exemple, celui de la **cohérence des contraintes** définies sur les données, la **qualité** de ces dernières en dépend.

Plusieurs formalismes ont été utilisés pour modéliser des données, tels que **UML** (Unified Modeling Language) dans Muller (2001), Soutou (2002) et Rumbaugh (2004), **ER** (Entity-Relationship) et **EER** (Extended Entity-Relationship) dans Chen (1976), Moulin (1976), Tardieu (1979) et Smith (1977). Afin de préserver la sémantique du monde réel, plusieurs *contraintes* sont définies sur les données. Alors que les propriétés formelles de chacune de ces contraintes sont largement maîtrisées, peu de travaux sont faits pour intégrer tous les concepts à la fois et donc étudier les conflits éventuels qui peuvent exister entre les différentes catégories de contraintes. Les outils d'aide à la conception de Bases de Données (BD) qui existent aujourd'hui, tels que Power Designer et Rational Rose, n'offrent pas la possibilité de contrôler la cohérence globale des différents types de contraintes.

Les seules études faites sur la cohérence des contraintes, traitent essentiellement celles de cardinalité dans un schéma conceptuel ER. La découverte des cycles critiques dans un graphe associé au schéma conceptuel permet de détecter que les contraintes de cardinalités ne sont pas satisfiables, Lenzerini (1990). Mais cette étude ne traite pas toutes les associations réflexives. En effet, les différents rôles de ce type d'associations doivent être pris en compte. Une classification des associations réflexives dans le formalisme ER a été donnée dans Dullea (1999). L'interaction de différentes classes de contraintes, en particulier les dépendances fonctionnelles et les contraintes de cardinalités, a été brièvement étudiée par Hartmann (1998, 2000). La satisfaction des contraintes de cardinalité a été abordée par Boufarès (2002, 2004). Elle se vérifie par la résolution du système linéaire associé au schéma ER pour détecter et localiser les conflits.

Certains de ces travaux ont été réalisés dans le cadre du modèle ER. Par contre le formalisme UML, qui a connu un développement très rapide depuis sa création, a fait l'objet de peu de contributions concernant l'étude des différents types de contraintes. Les outils d'aides à la conception dédiés à la modélisation UML n'offrent pas la possibilité de vérifier la cohérence globale de ces différents types de contraintes. Notons, par ailleurs, que le langage OCL (Object Constraint Language) OMG (2003c) n'assure pas la vérification de la cohérence globale des contraintes.

L'étude détaillée des conflits éventuels entre les contraintes définies sur les grosses masses de données nous semble très intéressante à plusieurs titres. Elle pourrait être utilisée dans le processus d'intégration de données et dans l'évaluation de la qualité de celles-ci.

Ce papier est organisé comme suit. La deuxième section présente brièvement les différents formalismes étudiés (UML et EER). Les différentes catégories de contraintes sont exposées dans la section trois qui détaille leur mise en équations. Nous présentons notre algorithme de détection et localisation des conflits dans la quatrième section. Enfin, nos travaux futurs sont donnés en guise de conclusion.

2 - LES FORMALISMES UML ET EER

Dans cette section, nous décrivons très brièvement les caractéristiques des formalismes UML et EER, utiles dans la suite de ce papier dans le but de présenter les différents types de contraintes que nous étudions. Pour plus de détails, les documents qui nous paraissent ceux de base sont ceux de Rumbaugh (2004), Chen (1976), Tardieu (1979) et Rochfeld (1993)

Un schéma, conceptuel S, de BD peut être représenté graphiquement avec deux

formalismes très proches UML et EER. Un schéma de BD peut ainsi être formulé aussi bien par un *diagramme de classes* UML (D_UML) que par un *diagramme entité-association* (D_EER). Les diagrammes de classes UML expriment de manière générale la structure statique d'un système, en termes de *classes* et de *relations* entre elles, agencées selon les règles dictées par les métamodèles donnés par exemple par OMG (2004, 2003a, 2003b). Les associations représentent les relations structurelles entre classes d'objets. La plupart des associations sont binaires. Elles peuvent relier une classe à elle-même dans le cas des structures récursives, ce type d'associations est appelé association réflexive. Il existe des associations n-aires reliant n classes entre elles. Chaque extrémité d'une association peut porter une indication de *multiplicité* qui montre combien d'objets de la classe considérée peuvent être reliés à un objet de l'autre classe. Les valeurs de multiplicité expriment les *contraintes* liées au domaine de l'application. Toutes sortes de contraintes peuvent être définies sur les liens entre objets, nous citons par exemple {ordonné}, {sous-ensemble}, {ou-exclusif}. Une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité s'appelle une agrégation ou une composition. UML emploie le terme de *généralisation* pour désigner la classification entre un élément plus général et un élément plus spécifique, on parle d'héritage.

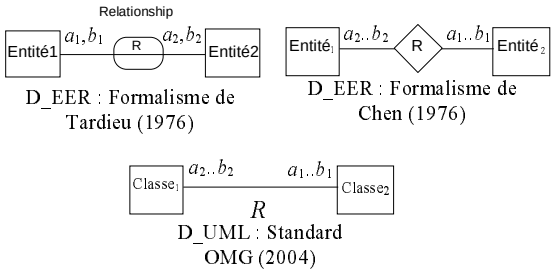


Figure.1 Différents formalismes de modélisation de schémas conceptuels de BD

Le formalisme EER est similaire à celui d'UML. Tous les concepts ci-dessus se retrouvent dans les diagrammes D_EER où on parle de types d'*entités* et d'*associations*, de contraintes de *cardinalité* et de *participation*. Les différents types d'associations cités ci-dessus, sont donnés dans un formalisme très proche, selon le métamodèle approprié, à la seule différence de l'agrégation.

Les diagrammes de la figure-1 résument les caractéristiques essentielles (entité, association, cardinalité, classe, multiplicité) dont on a besoin dans cet article. Notons que la lecture de la contrainte de cardinalité dans le formalisme ER de Tardieu est différente de celle dans les formalismes ER de Chen et celui d'UML.

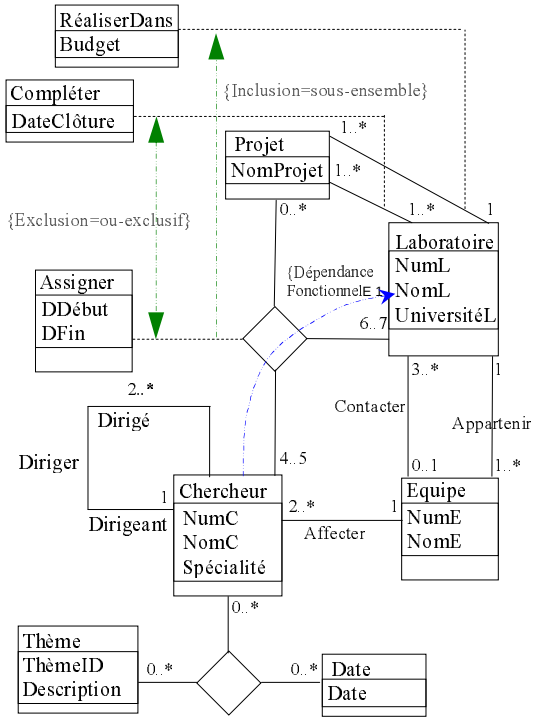


Figure.2 Gestion des laboratoires rattachés au CNRS
Diagramme de classes (D_UML)
Plusieurs catégories de contraintes

Les figures 2 et 3 représentent un exemple de schéma conceptuel de données respectivement UML et EER (les cardinalités sur la figure 3 sont à interpréter selon le modèle de Tardieu). Cet exemple décrit la gestion des laboratoires rattachés au centre national de recherche scientifique (CNRS). Chaque laboratoire doit contenir au moins une équipe. Chaque équipe doit contacter minimum trois laboratoires. Elle doit être composée d'au moins deux chercheurs. Un chercheur a au plus un directeur, ce dernier dirige au minimum deux chercheurs. Un chercheur n'est affecté qu'à une seule équipe. Cette dernière peut effectuer des projets dans des périodes différentes. Un chercheur travaille sur minimum 4 projets et

maximum 5 mais toujours dans un même laboratoire. Dans un laboratoire, il faut assigner minimum 6 couple (projet, chercheur) différents et maximum 7. Les diagrammes de ces deux figures (2 et 3) sont équivalents. Ils permettent de montrer que les mêmes problèmes peuvent être rencontrés dans les deux formalismes. En effet, ces deux diagrammes sont syntaxiquement corrects et donc acceptés par les différents outils d'aide à la conception. Ils sont cependant sémantiquement incorrects car certaines contraintes ne pourront jamais être vérifiées et par conséquent nous ne pouvons pas leur associer une instance de base de données cohérente. Nous prenons une partie de cet exemple et nous appliquons notre algorithme pour détecter les conflits générés par les contraintes.

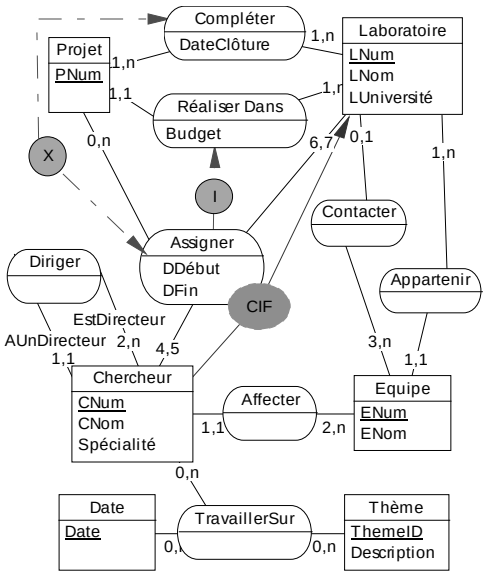


Figure.3 Gestion des laboratoires rattachés au CNRS Diagramme ER (D_EER)

3 - FORMALISATION DES CONTRAINTES

Un schéma conceptuel S, quel que soit le formalisme utilisé, est composé d'un ensemble de structures décrivant les données et d'un ensemble de contraintes sur ces données. Dans cette section nous illustrons notre approche pour formaliser les contraintes de multiplicité / cardinalité et les contraintes d'intégrité fonctionnelle.

Nous associons aux contraintes définies dans le schéma conceptuel S un système d'inéquations Σ_S défini par :

- Un ensemble **X** de variables $\boldsymbol{x_i}$ ($\boldsymbol{x_i}$ entier positif pour tout i) chacune correspond au nombre d'objets d'une classe C_i (type d'entité) ou celui des liens dans une association R_i.
- Un ensemble **A** de constantes $\boldsymbol{a_i}$ ($\boldsymbol{a_i}$ entier positif pour tout i) représentant les multiplicités (cardinalités) minimales.
- Un ensemble **B** de constantes $\boldsymbol{b_i}$ ($\boldsymbol{b_i}$ entier positif pour tout i) représentant les multiplicités (cardinalités) maximales.

La figure 4 représente un exemple d'un diagramme de classes avec une association binaire et comment nous associons des variables aux nombres d'objets de ses classes et au nombre de lien de son association.

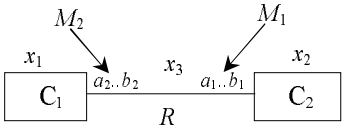


Figure.4 Diagramme de classes avec une association binaire

Nous considérons que x_1 et x_2 sont respectivement les nombres d'objets des classes C_1 et C_2, et x_3 est le nombre de liens de l'association R.

3.1 - Contraintes de multiplicité

La première catégorie de contraintes traitée est celle de multiplicité de type intervalle. Nous allons considérer ci-dessous les associations binaires et ternaires. Nous verrons les cas particuliers de l'agrégation et de la composition dans UML.

Une multiplicité, dans une association binaire, indique le nombre possible d'objets de la classe destination du rôle qui peuvent être reliés à un objet de la classe origine du rôle.

DEFINITION.1 : La multiplicité M_1, notée «$a_1..b_1$», de la classe C_1 appliquée sur l'association R (figure 4) est satisfaite si et seulement si pour tout objet v de la classe C_1, le nombre t de liens de R qui lui sont connectés doit être compris entre la multiplicité minimale et la multiplicité maximale « $a_1 \leq t \leq b_1$ ».

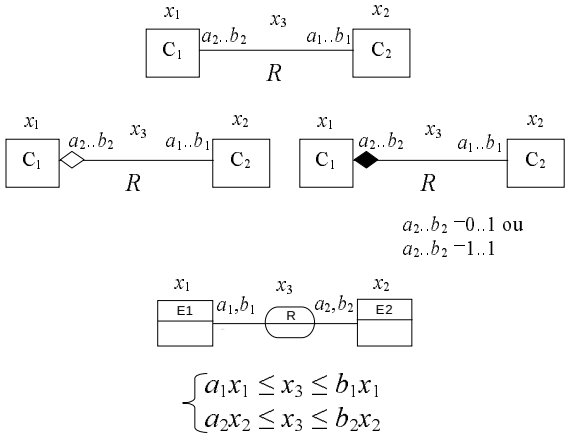
Cette inégalité est démontrée dans Boufarès (2002). D'une manière générale, la multiplicité M_1 est satisfaite si et seulement si x_3 (nombre de liens de l'association R) est compris entre

a_1x_1 (nombre minimum de participations de tous les objets de la classe C_1 dans l'association R) et b_1x_1 (nombre maximum de participations de tous les objets de la classe C_1 dans R).

DEFINITION.2 : Une multiplicité M_1 est satisfaite, si et seulement si, l'inéquation : $a_1x_1 \leq x_3 \leq b_1x_1$ est satisfaite.

Remarque : si la multiplicité minimum (resp. maximum) est égale à "0" (resp. est égale à "*") alors sa vérification est évidente, elle ne serra pas formalisée.

Les figures 5, 6 et 7 ci-dessous résument les inéquations associées aux contraintes de multiplicité définies sur les associations binaires non réflexives et réflexives ainsi que les associations ternaires dans les deux formalismes UML et EER. Les cas particuliers de l'agrégation et de la composition sont présentés.



$$\begin{cases} a_1x_1 \leq x_3 \leq b_1x_1 \\ a_2x_2 \leq x_3 \leq b_2x_2 \end{cases}$$

Figure.5 Multiplicité/Cardinalité dans une association binaire et les inéquations associées

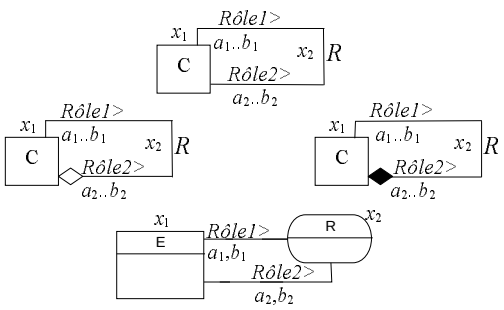
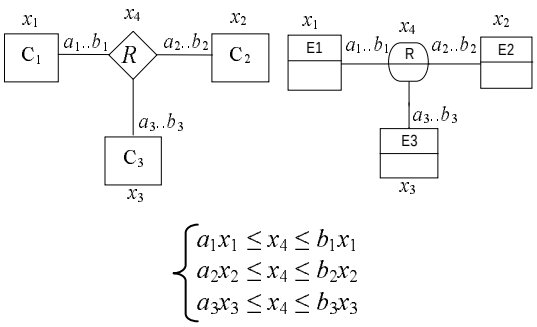


Figure.6 Multiplicité/Cardinalité dans une association récursive et les inéquations associées



$$\begin{cases} a_1x_1 \leq x_4 \leq b_1x_1 \\ a_2x_2 \leq x_4 \leq b_2x_2 \\ a_3x_3 \leq x_4 \leq b_3x_3 \end{cases}$$

Figure.7 Multiplicité/Cardinalité dans une association ternaire et les inéquations associées

3.2- Contraintes d'intégrité fonctionnelle

3.2.1 - Dépendance fonctionnelle

Une dépendance fonctionnelle (DF) entre deux classes C_1 et C_2 exprime qu'à un objet c_1 de C_1 correspond au plus un objet c_2 de C_2.

Pour satisfaire cette condition, il faut que le nombre d'objets de la classe C_2 soit inférieur ou égal à celui de la classe C_1. Cela se traduit par : $\boldsymbol{x_2 \leq x_1}$.

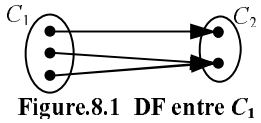


Figure.8.1 DF entre C_1 et C_2

Une DF, définie dans une association ternaire, entre deux classes C_1 et C_2 doit satisfaire les règles de correspondances entre les objets selon la figure-8.2.

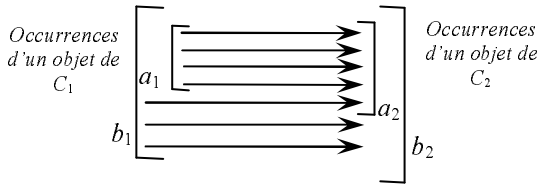


Figure.8.2 Correspondances entre les occurrences de C_1 et C_2

Il faut qu'il y ait, dans R, suffisamment d'occurrences des objets de C_2 pour satisfaire les occurrences des objets de C_1 :

- La multiplicité $a_1..b_1$, définie sur l'association R, exige la présence d'au minimum a_1 liens pour un même objet de la classe C_1, et d'au maximum b_1 liens pour ce même objet ;
- La multiplicité $a_2..b_2$, définie sur l'association R, exige la présence d'au minimum a_2 liens pour un même objet de la

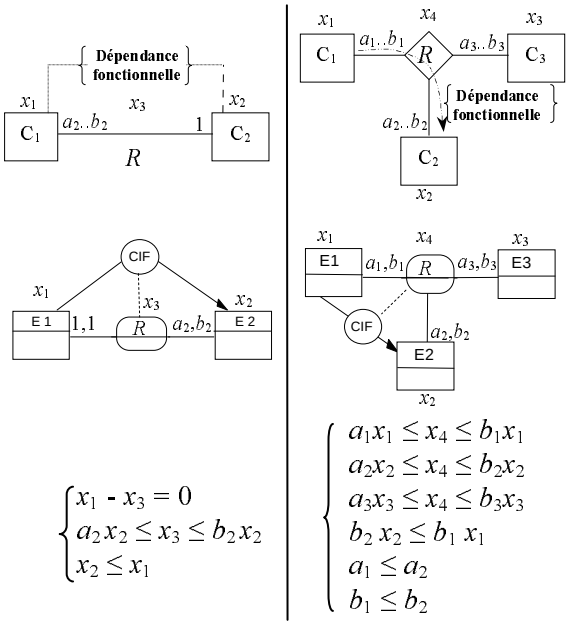
classe C_2, et d'au maximum b_2 liens pour ce même objet ;

- La dépendance existant entre C_1 et C_2 exige qu'un objet de C_1 ne puisse être relié qu'à un seul objet de C_2.

L'interaction de ces deux types de contraintes (multiplicité et dépendance fonctionnelle) génère un ensemble de critères. Ces derniers doivent être respectés afin de préserver la sémantique du monde réel.

- Le fait d'avoir la possibilité de prendre minimum a_1 occurrences d'un objet de C_2 exige que la multiplicité minimum a_2 soit supérieure ou égale à a_1. ($\boldsymbol{a_1 \leq a_2}$ d'où $\boldsymbol{a_1 \leq b_2}$)
- Le fait d'avoir la possibilité de prendre un maximum b_2 d'occurrences d'un objet de C_2 exige que la multiplicité maximum b_2 soit supérieure ou égale à b_1. ($\boldsymbol{b_1 \leq b_2}$)
- Le fait d'avoir une correspondance entre les objets de C_1 et C_2 exige avoir suffisamment d'occurrences des objets de la classe C_2 pour satisfaire les occurrences des objets de la classe C_1 ce qui donne : $b_2 x_2 \leq b_1 x_1$.

La figure.9 résume les différents cas possibles.



$$\begin{cases} x_1 - x_3 = 0 \\ a_2 x_2 \leq x_3 \leq b_2 x_2 \\ x_2 \leq x_1 \end{cases}$$

$$\begin{cases} a_1 x_1 \leq x_4 \leq b_1 x_1 \\ a_2 x_2 \leq x_4 \leq b_2 x_2 \\ a_3 x_3 \leq x_4 \leq b_3 x_3 \\ b_2 x_2 \leq b_1 x_1 \\ a_1 \leq a_2 \\ b_1 \leq b_2 \end{cases}$$

Figure.9 Interaction entre Multiplicité/Cardinalité et DF ainsi que les inéquations associées

3.2.2 - Contraintes de participation

Ces conditions concernent fréquemment la coexistence d'objets de plusieurs associations au départ d'une classe commune. La cohérence globale de ces contraintes devra être contrôlée afin d'empêcher la production des conflits. Dans ce paragraphe, nous présentons les contraintes d'exclusion et de simultanéité dans le formalisme UML. Il en est de même pour le formalisme EER (Figure.10).

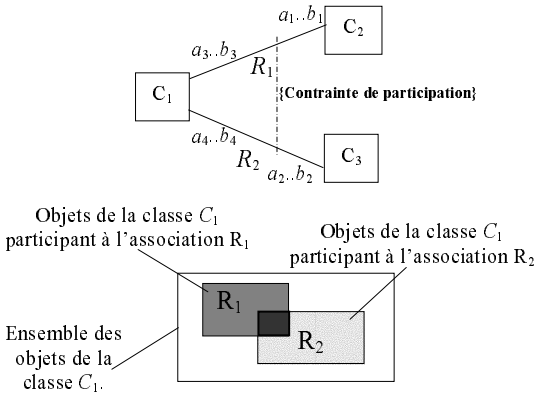


Figure.10 Contrainte de participation

3.2.2.1- Contrainte d'exclusion

Les objets d'une classe peuvent participer à deux ou plusieurs associations différentes. Cette participation peut être mutuellement exclusives. Cette contrainte est notée {**Exclusion**} (**X** en EER). Des définitions plus détaillées sont données par Matheron (1991), Rochfeld (1993) et Nanci (2001). Une contrainte d'exclusion d'une association R_1 vis-à-vis d'une association R_2 exprime le fait que les objets de la classe C_1 participant à l'association R_1 ne peuvent pas participer à l'association R_2 (figure.11).

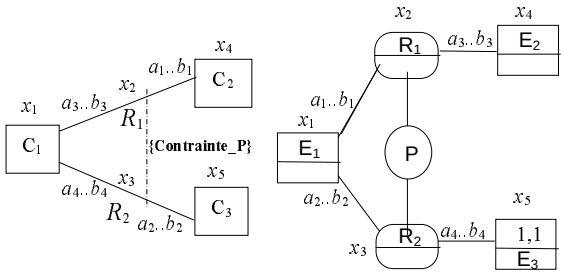


Figure.11 Interaction entre Multiplicité / Cardinalité et exclusion/simultanéité.

Dans le cas d'une exclusivité de participation d'une classe à deux associations, il faut que les multiplicités minimales de cette classe sur ces associations soient nulles. Dans le cas contraire, multiplicités minimales non nulles, on constate deux cas possibles. Le premier est celui où l'une des multiplicités n'est pas nulle ce qui oblige la participation totale de tous les objets de la classe C_1 dans cette association et non plus dans l'autre. On déduit alors que la

présence de cette dernière dans le schéma conceptuel est inutile. Le deuxième cas est celui où les deux multiplicités minimales ne sont pas nulles. Par conséquent, la participation de tout objet de C_1 est obligatoire dans R_1 et R_2 en même temps. La contrainte d'exclusion devient inutile dans ce cas.

Pour que la contrainte d'exclusion soit respectée, Il faut vérifier le nombre total des participations des objets de C_1 dans R_1 et R_2 par rapport au nombre minimum et le nombre maximum possibles de participations des objets de C_1, d'où :

$$\boldsymbol{x_2 + x_3 \leq \max(b_1,b_2)\, x_1}$$

Dans le cas d'interaction des contraintes de multiplicité avec la contrainte d'exclusion (P = X), le système d'inéquations est le suivant :

$$\begin{cases} x_2 \leq b_1 x_1 \\ x_3 \leq b_2 x_2 \\ a_3 x_4 \leq x_2 \leq b_3 x_4 \\ a_4 x_5 \leq x_3 \leq b_4 x_5 \\ x_2 + x_3 \leq \max(b_1,b_2)\, x_1 \end{cases}$$

3.2.2.2- Contrainte de simultanéité

Les définitions données à la contrainte de simultanéité dans la littérature, sont ambiguës. Cette dernière est notée {**Simultanéité**} (**S** en EER). Nous présentons dans ce papier deux cas différents de cette contrainte :

- Simultanéité de participation des objets d'une classe dans deux associations en terme d'existence.
- Simultanéité de participation des objets d'une classe dans deux associations en terme du nombre de participations de ces objets dans ces associations.

Le premier cas ne pose pas de problème puisqu'il n'a pas d'influence sur le nombre des occurrences des objets de la classe participant aux associations. Le deuxième cas pose problème d'une part dans la définition des contraintes de multiplicité, et d'autre part dans l'interaction des cette contrainte avec les contraintes de multiplicité. D'après la deuxième définition de la contrainte de simultanéité, les objets d'une classe doivent participer dans une association autant de fois que dans l'autre. Donc les multiplicités, minimales et maximales, de cette classe dans ces associations doivent être égales. Un autre critère est que le nombre d'objets des associations sur lesquelles est définie cette contrainte doit être compris entre $\boldsymbol{2\ a_1\ x_1}$ (tel que $a_1=a_2$) et $\boldsymbol{2\ b_1\ x_1}$ (tel que $b_1=b_2$) d'où l'inégalité :

$$\boldsymbol{2\ a_1\ x_1 \leq x_2 + x_3 \leq 2\ b_1\ x_1}$$

Le fait que la simultanéité soit définie sur le nombre de participations, on déduit que le nombre de liens de R_1 est égal à celui de R_2 ($\boldsymbol{x_2 = x_3}$). Donc, dans le cas d'interaction des contraintes de multiplicité avec la contrainte de simultanéité (P = S), le système d'inéquations est le suivant :

$$\begin{cases} a_1 x_1 \leq x_2 \leq b_1 x_1 \\ a_3 x_4 \leq x_2 \leq b_3 x_4 \\ a_4 x_5 \leq x_3 \leq b_4 x_5 \\ x_2 = x_3 \\ a_1 = a_2 \text{ et } b_1 = b_2 \end{cases}$$

Exemple 1 : Formalisation des contraintes d'une partie nommée S du schéma conceptuel de données de la figure 2.

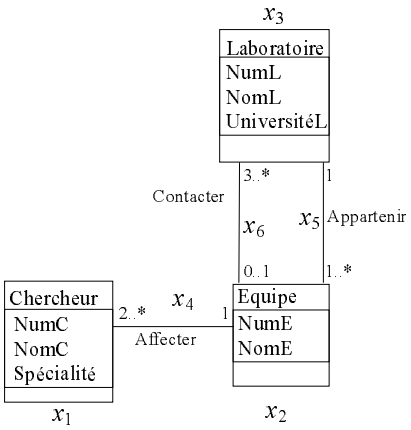


Figure.12 une partie du Schéma conceptuel de la figure.2 nommée S

L'ensemble des inéquations correspondant aux contraintes définies dans le schéma conceptuel S

$$\begin{cases} x_1 \leq x_4 \leq x_1 \\ 2.\ x_2 \leq x_4 \\ x_2 \leq x_5 \leq x_2 \\ x_3 \leq x_5 \\ 3.\ x_2 \leq x_6 \\ x_6 \leq x_3 \end{cases}$$

4 - DETECTION ET LOCALISATION DES CONFLITS

Nous associons à tout schéma conceptuel S, un système linéaire d'inéquations Σ_S. Nous étudions dans cette section la manière de définir ce système à partir des inégalités

associées aux différentes contraintes introduites dans le schéma conceptuel (section.3). Ce système sera ensuite résolu en faisant appel aux techniques de la programmation linéaire. Pour plus de détail, se reporter à Schrijver (1986) et Williams (1993). Dans notre approche, nous avons adapté l'algorithme Fourier-Motzkin (figure 13) adapté par Boufarès (2002). La solution fournie par cet algorithme permet de décider de la validité du schéma conceptuel. Les conflits sont engendrés soit par des contraintes de multiplicité (cardinalité) ou par l'interaction de ces dernières avec d'autres contraintes (contraintes d'intégrité fonctionnelle). Notre but est de détecter et localiser les conflits.

DEFINITION.3 : Un schéma conceptuel de données S est valide, si et seulement s'il existe au moins une instance (un diagramme d'objet, une base de données) qui vérifie toutes les contraintes définies sur S.

DEFINITION.4 : Un schéma conceptuel de données S est valide, si et seulement si le système linéaire correspondant Σ_S a une solution, Boufarès (2002, 2004).

Nous avons montré dans la section précédente que toute contrainte peut être traduite sous la forme $a\,x \le y \le b\,x$ (exemple 2), où a, b, x et y sont des entiers positifs. Cette inégalité sera transformée comme suit :

- Si $a = b$ alors $ax - y = 0$.
- Si $a \neq b$ alors $\begin{cases} ax - y \le 0 \\ y - bx \le 0 \end{cases}$

Exemple 2 : La transformation des inégalité de l'exemple 1 sera comme suit. Cet exemple sera traité par l'algorithme de la figure 13.

Σ_S

Le système d'inéquations linéaires correspondant au schéma conceptuel S deviendra comme ci-contre

$$\begin{cases} x_1 - x_4 = 0 \\ 2.\ x_2 - x_4 \le 0 \\ x_2 - x_5 = 0 \\ x_3 - x_5 \le 0 \\ 3.\ x_2 - x_6 \le 0 \\ x_6 - x_3 \le 0 \\ x_i \text{ sont des entiers positifs } \forall\ i=1,6 \end{cases}$$

Le principe de l'algorithme est de générer de nouvelles inégalités. Si l'une parmi celles-ci a tous ses coefficients positifs, il s'agit alors d'une condition d'arrêt qui traduit l'incohérence des contraintes introduites dans le schéma conceptuel.

```
Etant donné ΣS associé au diagramme de classe S ;
Si (association récursive) et (multiplicité dans {(1-1..*) ;
    (1-0..*) ; (1-1)}) Alors S n'est pas valide
Sinon
    Pour chaque contrainte de la forme y – a x = 0
        Substituer y par a x et éliminer la contrainte ;
    V ← { liste des variables de ΣS } ;
    Sol ← vrai ;
    Tant que V non vide et Sol faire
        Choisir une variable x de V ;
        V ← V - { x } ;
        CP ← {liste des contraintes de ΣS dont le
        coefficient de x est positif} ;
        CN ← {liste des contraintes de ΣS dont le
        coefficient de x est négatif} ;
        Si CP et CN ne sont pas vide
        Alors
            Diviser chaque contrainte de CP et CN par la
            valeur absolue du coefficient de x ;
            Additionner les contraintes de CP à celles de
            CN deux à deux
                    et ajouter les inégalités générées au
            système ΣS;
            Eliminer du système toutes les contraintes
            nouvellement générées n'ayant que des
            coefficients négatifs ;
            Si parmi les nouvelles contraintes générées il
            existe une dont les coefficients sont positifs
            Alors Sol ← Faux ;
            Fin Si
        Fin Si
        Eliminer de ΣS toutes les contraintes qui ont été
        sélectionnées dans CP et CN ;
    Fin tant que
    Si Sol = Faux
    Alors ΣS n'a pas de solution; S n'est pas valide ;
    Sinon ΣS a une solution entière; S est  Valide ;
    Fin Si ;
Fin Si.
```

Figure.13 Algorithme de détection

Ainsi notre algorithme nous permet de détecter différents types de conflits sur les schémas conceptuels des figures 1 et 2. Sur la figure 2, par exemple, les conflits détectés sont :

- Ceux engendrés par les contraintes de multiplicité des associations *Contacter* et *Appartenir* reliant les classes *Laboratoire* et *Equipe* ;
- Ceux générés par la sémantique de l'association réflexive *Diriger* ;
- Ceux engendrés par l'interaction entre la dépendance fonctionnelle définie sur l'association *Assigner* entre les classes *Chercheur* et *Laboratoire* et les contraintes de multiplicité de l'association *Assigner* appliquées sur les classes *Chercheur*, *Laboratoire* et *Projet*.

5 - CONCLUSION

Dans ce papier, nous avons présenté une étude sur le traitement de la cohérence globale des contraintes pouvant être définies dans les formalismes UML ou EER. Il s'agit de déterminer où et par quelles contraintes sont engendrés les conflits. Nous avons explicité d'une part, les conflits au sein d'une même catégorie de contraintes (les contraintes de multiplicité Berrabah (2003) et Boufarès (2004)), et d'autres part, nous avons montré certains conflits qui peuvent exister entre les deux catégories de contraintes étudiées : les contraintes de multiplicité (cardinalité) et celles d'intégrité fonctionnelle.

Notre approche se résume dans le fait de traduire les contraintes du schéma conceptuel sous forme de système d'inéquations linéaires. Ce dernier sera résolu en faisant appel aux techniques de la programmation linéaire. La solution obtenue permet de valider le schéma conceptuel, le cas échéant de déterminer et localiser les conflits.

L'objectif de cette recherche est d'intégrer ces techniques de vérification dans les outils d'aide à la conception. On pourra ainsi « *compiler* » un schéma conceptuel de bases de données avant de générer les schémas logique et physique correspondants. Notre étude devra être généralisée pour étudier les différents types de contraintes en incluant tous les concepts des formalismes UML et EER (contraintes de multiplicité de type énuméré, héritage, contraintes de participation…).

BIBLIOGRAPHIE

Berrabah, D., Boufares, F.,Ducateau, C. F., Heiwy, V.(2003), "Etude de la cohérence des contraintes de multiplicité dans un schéma conceptuel de BD avec UML", *National Conference of Research in IUT* (CNRIUT 2003), Tarbes, France, May 2003, Pages 375-382.

Boufarès, F., Bennaceur, H. (2002), "Consistency Problems in ER-Schemas for Database systems", *Proceedings of the 2002 International Arab Conference on Information Technology* (ACIT'2002), Doha, Qatar, December 16-19, 2002 ; pages 699-706; Vol.2.

Boufarès, F., Bennaceur, H. (2004), "Consistency Problems in ER-Schemas for Database systems", *Information Sciences journal*, Volume 163, 18 June 2004, Pages 263-274.

Chen, P.P. (1976), "The Entity-Relationship Model -Towards a Unified View of Data", volume 1, *ACM Transactions On Database Systems*, March 1976, pages 9-36.

Dullea, J., Il-Yeol Song (1999), "A Taxonomy of Recursive Relationships and Their Structural Validity in ER Modelling", ER'99, Conceptual Modelling, *18th International Conference on Conceptual Modelling*, LNCS 1728, Paris, France, November 1999 , pages 384-398.

Hartmann, S. (1998), "On the Consistency of Int-cardinality Constraints", ER'98 : Conceptual Modelling, *17th International Conference on Conceptual Modelling*, LNCS 1507, Singapore, November 1998, pages 150-163.

Hartmann, S. (2000), "On Interactions of Cardinality Constraints, Key, and Functional Dependencies", FolKS'2000 : *Foundations of Information and Knowledge Systems, First International Symposium*, LNCS 1762, Burg, Germany, February 2000, pages 136-155.

Lenzerini, M., Nobili, P. (1990), "On the satisfiability of dependency constraints in Entity-Relationship schemata", Info. Syst. 90, *Information Systems*, volume 15, N° 4, 1990, pages 453-461.

Matheron, J.P. (1991), "Approfondir Merise", Tome1, Edition Eyrolles 1991.

Moulin, P., Randon, J., Spaccapietra, S., Tardieu, H., Teboul, M. (1976), "Conceptual model as database design tool", *Proceedings of the IFIP Working conference on modelling in database Management Systems*, G.M. Nijssen Ed., North-Holland, 1976.

Muller, P.A. (2001), "*Modélisation objet avec UML*", 2ème édition Eyrolles, 2001.

Nanci, D., Espinasse, B., Cohen, B., Asselborn, J.C., Heckenroth, H. (2001), "*Ingénierie des systèmes d'information : Merise deuxième génération*, 4e édition". Edition Vuibert, 2001.

OMG (2004), editor: "*UML 2.0 Superstructure Specification*", OMG, 2004. http://omg.org.

OMG (2003a), editor: "*UML 2.0 Diagram Interchange Specification*", OMG, 2003. http://omg.org.

OMG (2003b), editor: "*UML 2.0 Infrastructure Specification*", OMG, 2003. http://omg.org.

OMG (2003c), editor: "*UML 2.0 Object Constraint Language Specification*", OMG, 2003. http://omg.org.

Rochfeld, A., Negros, P.(1993), "Relationship of relationships and other inter-relationship links in ER model", *Data and Knowledge Engineering*, volume 9, 1993, pages 205-221.

Rumbaugh, J., Jacobson, I., Booch, G. (2004) *UML 2.0 Guide de Référence*, Edition CampusPress, 2004.

Schrijver, A. (1986), "*Theory of linear and integer programming*", Edition Wiley, 1986.

Smith, J.M., Smith, D.C.P. (1977), "Database abstractions: Aggregation and Generalization", *ACM Transactions on Database Systems*, volume 2, N°2, June1977, pages 105-133.

Soutou, Ch. (2002), "*De UML à SQL, Conception de bases de données*", Edition Eyrolles, 2002.

Tardieu, H., Nanci, A., Pascot, D. (1979), "A method, A Formalism and Tools for Database Design (three years of Experimental practice)". *Proceedings of the international Conference on Entity-Relationship Approach to Systems Analysis and Design*, 1979.

Williams, H.P. (1993), "*Model Solving in Mathematical Programming*" John Wiley and Sons Ltd, England, 1993.

APPLICABILITE DU CRITERE D'EPSILON-SERIALISABILITE DANS LES SGBD TEMPS REEL

Emna Bouazizi,
Doctorante en Informatique
Emna.Bouazizi@univ-lehavre.fr, +33 (0) 232 744 384

Bruno Sadeg, Claude Duvallet,
Maîtres de conférences en Informatique
Bruno.Sadeg@univ-lehavre.fr, +33 (0) 232 744 405
Claude.Duvallet@univ-lehavre.fr +33 (0) 232 744 405

Adresse professionnelle
LIH, UFR des Sciences et Techniques ★ Université du Havre
25 rue Philippe Lebon ★ BP 540 ★ F-76058 Le Havre Cedex

Résumé : Le critère de sérialisabilité, reconnu comme le critère de correction des transactions dans les *SGBD (Systèmes de Gestion de Bases de Données)* traditionnels, est difficilement applicable dans un contexte temps réel. Ce critère s'avère trop strict pour l'exécution des transactions et pour l'accès aux données temps réel. L'epsilon-sérialisabilité est un critère de correction qui semble davantage adapté aux *SGBDTR (SGBD Temps Réel)* car il permet de relaxer la sérialisabilité lors du traitement des transactions temps réel au détriment de la cohérence stricte des résultats et des données. Dans beaucoup d'applications actuelles, ceci n'est pas très pénalisant à condition que cette incohérence soit bornée et contrôlée. On utilise pour cela des algorithmes de contrôle de divergence qui s'appuient sur le critère d'epsilon-sérialisabilité. L'objectif de cet article est d'étudier une adaptation du concept de l'epsilon sérialisabilité pour les *SGBDTR*.

Abstract: The serialisability, defined as the standard criterion of the transaction correctness in traditional Database Management Systems (DBMS), is poorly supported in a real-time context. This criterion is proved to be too strict for transaction execution and real-time data management. Epsilon-serialisability is a correctness criterion that seems to be more suited to Real-Time DBMS. It allows to relax the serialisability severity for processing of real time transactions to the detriment of the strict consistency of the results and data. Indeed, the Epsilon-serialisability criterion allows to control the inconsistencies by bounding the imported and exported database inconsistencies. For this purpose, we use divergence control algorithms which ensure epsilon-serialisability. This paper aims to study the adaptation of the epsilon-serialisability concept in the real-time context.

Mots clés : *SGBD* temps réel, sérialisabilité, epsilon-sérialisabilité, contrôle de concurrence, contrôle de divergence.

Keywords: real-time databases, serialisability, epsilon-serialisability, concurrency control, divergency control.

Applicabilité du critère d'epsilon-sérialisabilité dans les SGBD temps réel

1 – INTRODUCTION

Dans les *SGBD*, le concept de transaction est fondamental pour maintenir la cohérence des données. Les transactions effectuant des accès concurrents aux données, doivent être contrôlées afin d'éviter les conflits d'accès. Les protocoles de contrôle de concurrence permettent le respect du critère de correction, appelé *sérialisabilité* (*SR*). Ce critère conduit à n'accepter que les exécutions simultanées de transactions produisant les mêmes résultats qu'une exécution séquentielle de ces mêmes transactions. Vu l'importance de ce concept dans la gestion des transactions, la sérialisabilité a fait l'objet d'un grand nombre de travaux (Bernstein, 1987 ; Ullman, 1980 ; Date, 1985). Le concept de sérialisabilité s'est alors enrichi d'un développement récent pour faire apparaître d'autres types de critères de correction, car il s'avérait trop restrictif pour certaines applications (Ramamritham, 1993 ; Duvallet, 1999).

Avec l'avènement des nouvelles applications, une nouvelle notion est apparue dans les *SGBD*. Il s'agit du temps réel ainsi que la notion de contraintes temporelles qui lui est sous-jacente. Avec l'apparition des contraintes temporelles, les *SGBDTR* doivent non seulement gérer une masse importante de données, mais doivent également offrir des mécanismes pour respecter les contraintes temporelles des transactions et des données. Ces contraintes sont souvent explicitées sous forme d'échéances. Dans un contexte temps réel, la sérialisabilité est généralement trop restrictive (Ramamritham, 1993). D'autres critères de correction ont alors été proposés qui semblent davantage adaptés aux *SGBDTR* (Ramamritham 1993 ; Ramamritham, 1992). Dans cet article, nous présentons le critère d'epsilon-sérialisabilité (*ESR*) qui était proposé pour les *SGBD* classiques et nous étudions son applicabilité dans les *SGBDTR*. Nous prouvons qu'il adoucit la sévérité de la *SR* dans le traitement des transactions temps réel en permettant aux transactions, appelées alors Epsilon-Transactions (*ETs*), d'être entachées d'une incohérence contrôlée et limitée.

Par analogie avec les algorithmes de contrôle de concurrence (*CC*) qui supportent efficacement la *SR*, des méthodes de contrôle de divergence (*CD*) sont proposées pour garantir l'*ESR* des *ETs*. Les méthodes de *CD* sont une extension des méthodes de *CC*. Les algorithmes de *CD* sont similaires aux algorithmes de *CC* correspondants sauf que les méthodes de *CD* autorisent des conflits *non-sérialisables*, sous certaines conditions. Dans cet article, nous appliquons un algorithme de contrôle de divergence qui contrôle les incohérences importées et exportées par les *ETs* sujettes à des contraintes temporelles sous forme d'échéances. Des résultats obtenus lors de simulations ont montré que davantage de transactions temps réel respectent leurs échéances par rapport à l'application du critère de sérialisabilité classique.

La suite de cet article est organisée de la façon suivante. La section 2 définit le concept de *SR* et celui du contrôle de concurrence dans les *SGBD* classiques. La section 3 est consacrée à un rappel sur l'*ESR*, et la section 4 aux *ETs*. La section 5 présente les méthodes de contrôle de divergence. L'applicabilité de l'*ESR* est discutée dans la section 6. Dans la section 7, nous présentons les résultats de simulation de nos travaux et discutons des résultats obtenus. Nous concluons cet article en soulignant l'apport de nos travaux et présentons quelques perspectives de recherche.

2 - SERIALISABILITE ET CONTROLE DE CONCURRENCE

L'application des protocoles de *CC* permet de garantir la *SR* et de résoudre les conflits d'accès aux données. Il est alors nécessaire de commencer par définir le concept du *CC* avant de présenter la *SR*.

2.1 - Contrôle de concurrence

Le contrôle de concurrence a pour fonction d'assurer, à chaque transaction, la propriété d'Isolation, une des propriétés *ACID*[1] des transactions. Cette propriété vise à garantir à chacune des transactions un accès aux objets d'une base de données comme si elle était seule à s'exécuter dans le système. Elle est obtenue par différentes méthodes qui ont pour but de synchroniser les accès aux objets, effectués par des transactions concurrentes. Toutes ces méthodes reposent sur le principe de la sérialisabilité des transactions (Bernstein, 1987).

2.2 - La sérialisabilité

La sérialisabilité consiste à n'accepter que les exécutions simultanées de transactions produisant les mêmes résultats qu'une exécution séquentielle de ces transactions. La sérialisabilité représente aussi le critère de correction généralement accepté pour le contrôle de concurrence des transactions (Kamath, 1993 ; Ramamritham, 1992 ; Ramamritham, 1993).

Une exécution sérialisable est une exécution entrelacée des actions d'un ensemble de transactions {T1, T2,...,Tn}, qui donne globalement

[1]Atomicité, Cohérence, Isolation, Durabilité.

et pour chaque transaction participante le même résultat qu'une exécution en série de T1, T2,...,Tn. La sérialisabilité maintient la cohérence de la base de données. En effet, si la base de données exécute seulement des transactions sérialisables et si l'état initial de cette base de données est cohérent, alors la sérialisabilité garantit un passage vers un autre état cohérent de la base.

La sérialisabilité s'appuie sur la relation de précédence entre transactions qui peut être représentée par un graphe de précédence dont les nœuds représentent les transactions et un arc de T_i vers T_j modélise la relation : « T_i précède T_j » dans l'exécution analysée. Une condition suffisante de sérialisabilité est que le graphe de précédence soit sans circuit (acyclique).

2.3 - Limitation de la sérialisabilité classique

Malgré l'importance des services offerts, la *SR* possède aussi ses limitations. Elle exige que les transactions concurrentes, y compris celles effectuant seulement des opérations de lecture, soient ordonnancées dans un ordre sérialisable. Lorsque le nombre de transactions concurrentes augmente, la quantité de conflits augmente, ce qui provoque l'annulation de certaines transactions et la diminution du niveau d'efficacité. Pour les transactions en *lecture seule* qui peuvent tolérer une certaine quantité d'incohérence, la sévérité de la *SR* peut imposer des limitations sur le débit du système et la qualité de service (*QdS*).

3 - EPSILON-SERIALISABILITE

3.1 - Motivations

Pour atténuer les effets négatifs de la *SR*, la notion d'epsilon-sérialisabilité, notée par la suite *ESR*, a été proposée (Pu, 1991). En effet, l'*ESR* permet d'étendre la notion de cohérence et les utilisateurs peuvent alors bénéficier d'un haut niveau de tolérance aux incohérences. Certaines applications peuvent avoir recours à l'*ESR*. De telles applications doivent pouvoir tolérer une certaine quantité d'incohérence, due à l'imprécision naturelle des données ou à des considérations de performances du système et de disponibilité des données. Elle autorise les traitements asynchrones et par conséquent une grande disponibilité et une meilleure autonomie.

3.2 - Définition

La notion d'*ESR* traite le problème du côté des transactions. À chaque requête, est associée une incohérence exportée vers la base de données et une incohérence importée. Plus précisément les écritures exportent une incohérence vers la base et les lectures en importent. Ces incohérences sont rassemblées dans deux accumulateurs et la transaction peut poursuivre son exécution tant que ces accumulateurs ne dépassent pas une certaine valeur (ε). L'idée générale est que l'*ESR* maintient un nombre d'incohérences à l'intérieur de limites spécifiées par ε. ε est alors la quantité d'incohérence autorisée par l'*ESR*. Lorsque ε tend vers 0, l'*ESR* tend vers la *SR* classique (Ramamritham, 1995 ; Sadeg, 2003).

Soit T un ensemble de transactions qui accèdent à la base de données. Nous désignons dans la suite par $\sum_{t\in T} Inc^{import}(t)$ l'incohérence importée par T, et par $\sum_{t\in T} Inc^{export}(t)$ l'incohérence exportée par T. ε^{import} et ε^{export} sont les limites d'incohérence autorisées par l'*ESR*.

La définition de l'*ESR* est établie en supposant l'existence d'une condition sûre pour l'ensemble de transactions T, notée par :

- $\sum_{t\in T} Inc^{import}(t) \le \varepsilon^{import}$
- $\sum_{t\in T} Inc^{export}(t) \le \varepsilon^{export}$

Cette condition exige que l'incohérence importée (resp. exportée) à la base de données ne dépasse pas la limite d'incohérence autorisée.

L'objectif de l'*ESR* est de contrôler la quantité d'incohérence dans les applications qui manipulent de grandes quantités de données. Cette quantité ε dépend de l'application et elle est spécifiée par l'administrateur de la base de données. Elle est mesurée en termes de fonction '*distance*' définie sur les états de la base de données. Désignons par ***P(conflit(L,E))*** le poids d'un conflit entre la lecture (L) et l'écriture (E), qui est représenté par la fonction '*distance*' tel que :

$$\boldsymbol{P(conflit(L,E)) = distance\ (E_{avant}, E_{après})}$$

C'est-à-dire que le poids du *conflit(L,E)* est la distance entre l'état avant l'écriture (E_{avant}) et l'état après l'écriture ($E_{après}$). ***P(conflit(L,E))*** est alors la quantité d'incohérence potentielle introduite par les transactions à cause des conflits.

Supposons que *x1* et *x2* soient deux états de la donnée *x*, la distance entre ces deux états est la différence entre ces deux valeurs :

$$\boldsymbol{Distance((x1),(x2)) = |x1-x2|}$$

Supposons qu'une opération de lecture sur *x* renvoie la valeur 5, et plus tard, qu'une opération d'écriture modifie *x* par 13. Le poids du conflit de lecture/écriture est 8.

$$\boldsymbol{P(conflit(L,E)) = Distance((5),(13)) = |5-13| = 8}$$

Plus généralement, on peut aussi appliquer la fonction de distance sur l'ensemble des états des données de la base. Supposons que la base de données contienne trois données de type numériques *x, y* et *z*. Un état est représenté par le triplet (*x*, *y*, *z*). On peut alors définir la distance comme suit :

$$\boldsymbol{Distance\ ((x1,y1,z1),(x2,y2,z2)) = |x1-x2| + |y1-y2| + |z1-z2|}$$

Par exemple, la distance entre l'état (100,200,400) et l'état (300,500,400) est 500 (i.e, 200+300+0) qui représente la quantité d'incohérence ε introduite dans la base de données. Deux valeurs doivent être modifiées lors de la transition entre les deux états donc le passage de l'état (100,200,400) à l'état (300,500,400) exige au moins deux opérations d'écriture qui peuvent causer des conflits de type lecture/écriture.

4 - EPSILON-TRANSACTION ET INCOHERENCE DES DONNEES

L'Epsilon-Transaction (*ET*) est une extension des transactions classiques (pour la *SR*) par l'ajout d'une spécification de la quantité d'incohérence. Cette quantité est donnée par quelques mesures sur les opérations de la base de données ou de la fonction distance dans l'espace d'états de cette base de données. Pour chaque Epsilon-Transaction, cette spécification est divisée en deux parties (Pu, 1991 ; Pu, 1992) :

1. une incohérence importée, notée « $\boldsymbol{\varepsilon}_{ET}^{import}$ »
2. une incohérence exportée, notée « $\boldsymbol{\varepsilon}_{ET}^{export}$ »

	$\boldsymbol{\varepsilon}_{ET}^{import}$	$\boldsymbol{\varepsilon}_{ET}^{export}$
T	**= 0**	**= 0**
$\boldsymbol{ET_L}$	**> 0**	**= 0**
$\boldsymbol{ET_E}$	**= 0**	**> 0**

Tableau 1: les différents types d'ETs

Le tableau 1 illustre les différents types d' *ETs* :

- des *ETs* qui sont sérialisables (au sens classique), notées "Transaction" (et par la suite T),
- des *ETs* de lecture, notées $\boldsymbol{ET_L}$,
- des *ETs* de mises à jour, notées $\boldsymbol{ET_E}$.

Si $\boldsymbol{\varepsilon}_{ET}^{import}$ = 0 et $\boldsymbol{\varepsilon}_{ET}^{export}$ = 0, alors l'*ET* est sérialisable. Par conséquent les *ETs* considèrent les transactions classiques comme un cas limite. Lorsque $\boldsymbol{\varepsilon}_{ET}^{import}$ > 0 et $\boldsymbol{\varepsilon}_{ET}^{export}$ = 0, les $\boldsymbol{ET_L}$ peuvent importer une incohérence limitée par $\boldsymbol{\varepsilon}_{ET}^{import}$. De même lorsque $\boldsymbol{\varepsilon}_{ET}^{import}$ = 0 et $\boldsymbol{\varepsilon}_{ET}^{export}$ > 0, les $\boldsymbol{ET_E}$ peuvent exporter une incohérence limitée par $\boldsymbol{\varepsilon}_{ET}^{export}$. Cependant, si les *ETs* importent et exportent des incohérences, elles peuvent alors introduire une nouvelle incohérence illimitée dans la base de données. Ces cas sont résumés dans le tableau 1.

À part l'incohérence des transactions, il faut aussi penser à limiter l'incohérence des données. En effet, chaque donnée a une quantité d'incohérence autorisée qu'elle peut importer de (ou exporter dans) la base de données. L'incohérence importée par une epsilon transaction sera la somme des incohérences des données lues. Et l'incohérence exportée sera la somme des incohérences exportées par les données écrites. Il existe alors une forte dépendance entre l'incohérence des transactions et celle des données, représentée par les deux inéquations suivantes :

- Incohérence des données lues ≤ $\boldsymbol{\varepsilon}_{ET}^{import}$
- Incohérence des données modifiées ≤ $\boldsymbol{\varepsilon}_{ET}^{export}$.

On peut illustrer notre définition par l'exemple suivant qui met en évidence la dépendance des deux incohérences.

Soient deux transactions T1 et T2, et trois données *a*, *b* et *c* qui ont initialement pour valeurs : 3, 4 et 2 respectivement (cf. figure 1).

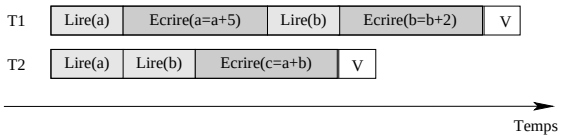


Figure 1 : Importation/Exportation des incohérences

Leur exécution est alors la suivante :

- T1 commence par lire la valeur de *a* (*a=3*) puis la modifier (*a=a+5*), ensuite elle lit la valeur de *b* (*b=4*) puis la modifie (*b=b+2*).
- T2 commence par la lecture de *a* et de b. Ces valeurs n'ont pas encore été modifiées en base car T1 n'est pas encore validée. Une incohérence importée a été alors associée à *a* et à *b*. L'incohérence importée par T2 est la somme des incohérences associées à *a* et à *b*. De même, pour écrire *c*, T2 doit toujours se baser sur les anciennes valeurs de *a* et de *b*, et l'incohérence exportée par T2 est l'incohérence associée à *c*.

5. - CONTROLE DE DIVERGENCE

5.1 - Définition

Comme nous l'avons déjà cité, l'*ESR* adoucit la sévérité de la *SR* dans le traitement des transactions en autorisant une incohérence limitée de la base de données. La limite d'incohérence est automatiquement maintenue par les méthodes de contrôle de divergence (*CD*), de la même façon que la *SR* est maintenue par le mécanisme de contrôle de concurrence (*CC*). Les méthodes de *CD* sont une extension des méthodes de *CC*. Cependant, le contrôle de divergence pour l'*ESR* améliore la concurrence puisque des transactions qui seraient bloquées par le critère de sérialisabilité peuvent ne plus l'être.

Par analogie avec les algorithmes de *CC* qui supportent efficacement la *SR*, les méthodes de *CD*

présentent une conception efficace pour la garantie de l'*ESR*.
Les méthodes de contrôle de divergence (*CD*) effectuent une analyse des méthodes de contrôle de concurrence (*CC*) pour identifier les endroits où les algorithmes de *CC* détectent les conflits des opérations sur les bases de données, i.e. qui provoquent la *non-sérialisabilité* (*non-SR*). Ensuite, elles modifient les algorithmes, qui tiennent compte de la *non-SR* et des opérations conflictuelles (Lecture/Ecriture), de telle façon que la tolérance à l'incohérence soit limitée. Les méthodes de *CD* sont similaires aux algorithmes de *CC* correspondants à la différence près que les méthodes de *CD* autorisent des conflits , sous certaines conditions (Pu, 1997 ; Son, 1993 ; Ulusoy, 1995).

5.2 - Mise en œuvre

Malgré les conceptions différentes pour les méthodes de *CD*, l'idée commune de base consiste à détecter chaque conflit *non-SR* vu par $\boldsymbol{ET_L}$ et à limiter la quantité d'incohérence importée. Par exemple, si les conflits *non-SR* sont comptés uniquement une seule fois pour chaque donnée, alors le nombre de ces conflits est équivalent au nombre total de données lues de façon *non-SR*. Si une $\boldsymbol{ET_L}$ lit des données et une $\boldsymbol{ET_E}$ concurrente est en train de modifier la donnée, alors les transactions ne sont pas sérialisables. Ces données sont donc lues de façon "non-sérialisable".
Les données lues de façon *non-SR* peuvent être utilisées pour modéliser diverses applications pratiques. Si le nombre total des données, lues par une ET_L *non-SR*, est limité par ces spécifications, alors les ET_L peuvent être exécutées simultanément et non-sérialisablement avec d'autres ET_E.
Deux accumulateurs d'incohérence sont associés à chaque *ET*, Accumu-Import et Accumu-Export.
Accumu-Import enregistre la quantité totale d'incohérence que l'*ET* a importé et Accumu-Export maintient et préserve la quantité totale d'incohérence que l'*ET* a exporté. Pour limiter l'incohérence, les méthodes de *CD* s'assurent que pour chaque *ET*, Accumu-Import $\leq \varepsilon_{ET}^{import}$ et Accumu-Export $\leq \varepsilon_{ET}^{export}$.
La conception des méthodes de *CD* repose sur deux phases :

- une « phase d'extension » : les méthodes de *CC* sont étendues par l'identification des endroits où la *non-SR* et les conflits sont détectés.
- une « phase de relaxation » : la *SR* est détendue et relâchée par l'utilisation de l'Accumu-Import et de l'Accumu-Export pour tenir compte de la concurrence des *ETs*.

Dans le *CD,* il s'agit juste d'identifier les conflits *non-SR* et d'empêcher la formation de cycles dans les graphes de sérialisation.
L'étape d'extension isole la partie d'identification des conflits, et l'étape de relaxation modifie la partie de prévention du cycle afin de permettre des incohérences spécifiques. Les modifications varient d'une méthode à l'autre. Il existe plusieurs modifications possibles et plusieurs façons de limiter l'incohérence. Des algorithmes ont été proposés pour garantir l'*ESR* dans les traitements des *ETs* (Wu, 1992 ; Tsang, 1997). Nous étudions le verrouillage à deux phases du *CD* (noté par la suite 2*PLDC*) car le protocole 2*PL*[2] est de loin la technique la plus utilisée (Bernstein, 1987).

5.3 - 2PLDC pour le temps réel

Comme tout protocole de contrôle de divergence (CD), l'application du 2*PLDC* comporte deux phases.

Phase d'extension
C'est la phase d'identification et de détection des conflits en se basant sur une matrice de compatibilité des verrous.
La matrice de compatibilité des verrous pour le 2*PL* standard (*2PLCC*) est présentée par le tableau 2, et celle pour le 2*PLDC* par le tableau 3.

	VL^T	VE^T
VL^T	**ACC**	**X**
VE^T	**X**	**X**

Tableau 2 : ***Compatibilité pour 2PLCC***

Dans le tableau 2 :
- VL^T désigne le verrou de lecture détenu par une transaction,
- VE^T désigne le verrou d'écriture détenu par une transaction.

	VL^{ET_L}	VL^{ET_E}	VE^{ET_E}
VL^{ET_L}	**ACC**	**ACC**	**ACCL-1**
VL^{ET_E}	**ACC**	**ACC**	**X**
VE^{ET_E}	**ACCL-2**	**X**	**X**

Tableau 3 : ***Compatibilité pour 2PLDC***

Dans le tableau 3 :
VL^{ET_L} désigne le verrou de lecture détenu par une ET_L, VL^{ET_E} désigne le verrou de lecture détenu par une ET_E, VE^{ET_E} désigne le verrou d'écriture détenu par une ET_E.
Dans les deux tableaux, les colonnes représentent les verrous détenus et les lignes représentent les verrous demandés. Les cases marquées par ACC

[2]Verrouillage à 2 phases

correspondent aux verrous compatibles (Accord) et les cases marquées d'une croix aux verrous incompatibles.
Le tableau 3 diffère du tableau 2 par l'ajout des deux cases : ACCL-1 et ACCL-2 (Accord Limité). ACCL-1 permet aux ET_L de lire les données non validées (non committées) alors que ACCL-2 permet aux ET_E d'écraser la donnée qui a déjà été lue par une ET_L. On suppose qu'une fois le verrou de VE^{ET_E} acquis, l'ET_E écrit sa nouvelle donnée sur place dans le tampon (buffer) pour que toute autre ET_L puisse immédiatement lire la nouvelle donnée mise à jour mais non encore validée.
L'histoire (1) ci-dessous montre un exemple de verrou ACCL-1sur la donnée 'b' :

$$L_1^{ET_E}(a)\ L_1^{ET_E}(b)\ L_2^{ET_L}(a)\ \underline{E_1^{ET_E}(b)\ L_2^{ET_L}(b)}\ E_1^{ET_E}(a) \qquad (1)$$

Dans cette histoire, la partie soulignée met en évidence un conflit de type E/L sur la donnée 'b'. Le verrou de lecture VL^{ET_L}, demandé par ET_L sur la donnée 'b', peut être acquis si la transaction est proche de son échéance, même si le verrou d'écriture VE^{ET_E} est encore détenu par ET_E.

L'histoire (2) est un exemple de concurrence permise par ACCL-2 sur la donnée '*a'* :

$$L_1^{ET_E}(a)\ L_1^{ET_E}(b)\ E_1^{ET_E}(b)\ \underline{L_2^{ET_L}(a)\ E_1^{ET_E}(a)}\ E_2^{ET_L}(b) \qquad (2)$$

Dans cette histoire, la partie soulignée met en évidence un conflit de type L/E sur la donnée 'a'. Le verrou d'écriture VL^{ET_E}, demandé par ET_E sur la donnée 'a', peut être acquis si la transaction est proche de son échéance, même si le verrou de lecture VL^{ET_L} est encore détenu par ET_L.

Phase de relaxation
Pour contrôler la quantité d'incohérence permise dans les ET_L, nous raffinons la gestion des verrous sur **ACCL-1** et **ACCL-2**. A chaque fois que ET_L demande un verrou VL^{ET_L} dans le cadre de **ACCL-1**, on examine d'abord les valeurs des variables Accumu-Import et Accumu-Export, respectivement de ET_L et ET_E, pour voir si, en les incrémentant d'une unité, elles dépasseront leurs limites (Son, 1993). De la même façon, à chaque fois qu'une ET_E demande un verrou VE^{ET_E} dans le cadre de **ACCL-2**, toutes les ET_L détentrices du verrou VL^{ET_L} en conflit ont leurs Accumu-Import examinés pour voir s'ils ont dépassé leurs $\varepsilon_{ET}^{import}$ après incrémentation d'une unité. Cependant, l'ET_E doit vérifier ses propres Accumu-Export contre (face à) l'incrémentation par le nombre total des ET_L détentrices des verrous VL^{ET_L}. Si l'Accumu-Export de ET_E dépasse son seuil $\varepsilon_{ET}^{export}$ ou si n'importe quel Accumu-Import d'une ET_L dépasse son $\varepsilon_{ET}^{import}$, alors nous devons arrêter le flux d'informations et interdire cet accès particulier. Dans ce cas, on peut choisir de faire attendre le demandeur ou d'annuler la transaction, puis recommencer en retournant les verrous échoués aux *ETs* demandeuses ou en brisant quelques verrous déjà détenus par d'autres *ETs* (en fonction des priorités).

6 - APPLICATION DE L'ESR

6.1 - Applicabilité de l'ESR aux transactions temps réel

L'*ESR* possède un grand nombre d'algorithmes exécutables et efficaces. Sa compatibilité est ascendante puisqu'elle atteint la *SR* classique lorsque ε tend vers 0.
Une classe d'applications où l'*ESR* semble particulièrement souhaitable concerne celles qui manipulent des données dont les valeurs sont mises à jour périodiquement. Cette classe d'applications est souvent caractérisée par :
- une grande quantité de données,
- un rafraîchissement fréquent de la base de données,
- des contraintes temps réel strictes non critiques où les mises à jour doivent être acceptées à l'intérieur d'un intervalle de temps, sinon elles sont rejetées,
- une conception des applications qui accepte que les décisions puissent être basées sur des résultats incomplets/imprécis.

Des algorithmes de *CD* ont été proposés pour limiter l'incohérence vue par les ET_L à un degré qui peut être toléré selon l'application.

6.2 - L'apport de l'application de l'ESR

Dans cet article, nous avons proposé d'utiliser le critère d'epsilon sérialisabilité pour les transactions temps réel. En effet, l'*ESR* est à la fois faisable et largement applicable pour le traitement des transactions temps réel. Des simulations que nous avons effectuées montrent que ce critère est tout à fait adéquat dans les SGBDTR. Par rapport à la sérialisabilité classique, l'*ESR* :
- autorise des traitements asynchrones,
- offre une plus grande disponibilité des données puisqu'elle autorise l'acquisition des verrous sur une donnée qui est déjà verrouillée par une autre transaction,
- permet d'augmenter le nombre des transactions qui réussissent à s'exécuter avant leur échéance car elles ne sont pas obligées d'attendre la libération d'un verrou pour accéder à une donnée verrouillée.

7 - SIMULATION

Dans cette simulation, nous souhaitons mettre en évidence le gain de performance engendré par l'utilisation du critère d'*ESR* dans les *SGBDTR* par rapport au critère de *SR*. On applique pour cela une technique de contrôle de divergence, 2*PLDC*, qui permet de garantir l'*ESR* dans le cadre d'un *SGBD* classique.

Le modèle simplifié que nous avons utilisé consiste en une base de données centralisée dans laquelle s'effectuent des transactions (qui utilisent comme critère de correction la *SR*), puis des Epsilon-Transactions (qui utilisent comme critère de correction l'*ESR*). Pour le *CC*, le protocole *2PLCC* est utilisé alors que pour le *CD*, le *2PLDC* est utilisé. *2PLCC* est mis en œuvre en deux phases : (1) une phase d'acquisition de verrous et (2) une phase de libération de verrous.

Les verrous sont demandés au moyen de l'opération `LOCK(G,M)` de l'objet `LOCKController` et libérés par l'opération `UNLOCK(G)` du même objet. *G* désigne le granule à verrouiller ou à déverrouiller et *M* le mode de verrouillage.

Pour introduire la notion de temps réel, les transactions (et Epsilon-Transactions) possèdent des échéances. Chaque transaction possède son propre contrôleur de temps qui compare l'échéance avec le temps courant. Tant que la transaction n'est pas encore validée, on vérifie si l'instant courant est égal ou non à l'échéance. Si le temps est égal à l'échéance alors que la transaction n'est pas terminée alors la transaction est abandonnée.

Un gestionnaire de file d'attente permet d'insérer, dans une file, les transactions qui sont en attente de libération de verrous. Elles sont ordonnées en fonction de leurs priorités. La priorité la plus élevée correspond à l'échéance la plus proche suivant le principe du protocole Earliest Deadline First (*EDF*) (Liu, 1973). Un gestionnaire de transactions permet de consulter régulièrement la file, de la mettre à jour, de vérifier la validité des transactions (c'est-à-dire si elles n'ont pas encore raté leur échéance) et il envoie celles qui sont encore valides[3] à la base pour qu'elles effectuent une nouvelle tentative de verrouillage.

Une Epsilon-Transaction (*ET*) est caractérisée par sa propre échéance, sa quantité d'incohérence importée ($\varepsilon_{ET}^{import}$), sa quantité d'incohérence exportée ($\varepsilon_{ET}^{export}$) et une liste des opérations de lectures et d'écritures.

Deux accumulateurs (Accumu-Import et Accumu-Export) permettent de contrôler la quantité d'incohérence autorisée par l'*ESR*. À chaque importation d'incohérence (resp. exportation) l'Accumu-Import (resp. l'Accumu-Export) sera incrémenté de 1 jusqu'à la limite autorisée $\varepsilon_{ET}^{import}$ (resp. $\varepsilon_{ET}^{export}$).

Une exécution de notre programme a permis de confirmer les apports de l'*ESR* par rapport à la *SR*. Les résultats ainsi obtenus sont présentés dans le tableau 4.

Le tableau 4 montre que le nombre de transactions qui réussissent par l'*ESR* est beaucoup plus important que celui offert par la *SR* classique. Les transactions qui réussissent sont celles qui respectent leurs échéances, c'est à dire qui valident avant échéance. Le fait d'adoucir les limitations de la *SR* améliore donc le résultat final.

On constate également que le taux de réussite des transactions augmente avec l'augmentation du nombre des transactions à exécuter. Ceci s'explique car : lorsque le nombre de transactions augmente dans le système, le nombre de conflits augmente également (en probabilité). En appliquant le critère de *SR* classique, il y a de fortes chances que beaucoup de verrouillages surviennent. Ces verrouillages conduiront fatalement à davantage de transactions qui manquent leurs échéances. Avec l'*ESR*, beaucoup de conflits sont évités puisqu'on permet que des incohérences contrôlées surviennent. Les transactions, appelées Epsilon-Transactions, ont donc plus de chances de se terminer avant leurs échéances.

Les résultats donnés dans le tableau 4 sont représentés graphiquement sur la figure 2. On constate que l'*ESR* permet à plus de transactions de terminer avant leur échéance. L'amélioration est de l'ordre de 35% par rapport à la *SR* classique.

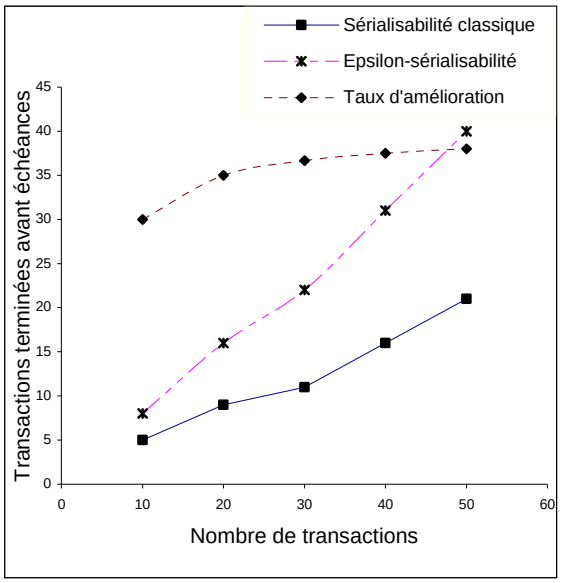


Figure 2: Nombre de transactions qui réussissent par la SR et par l'ESR et taux d'amélioration

[3] qui n'ont pas encore manqué leur échéance

Nombre initial de transactions	Nombre de transactions qui réussissent par la *SR*	Nombre de transactions qui réussissent par l'*ESR*	Taux d'amélioration
10	5	8	0.3
20	9	16	0.35
30	11	22	0.366667
40	16	31	0.375
50	21	40	0.38

Tableau 4: ***Comparaison des résultats entre la SR et l'ESR***

8. - CONCLUSIONS ET PERSPECTIVES

L'objectif de ce travail était d'étudier l'*ESR* et la gestion des conflits entre *ETs* afin de démontrer l'apport de l'*ESR* par rapport à la sérialisabilité classique dans le cadre des *SGBDTR*. Pour cela, un ensemble de concepts ont été définis afin de pouvoir mettre en œuvre l'*ESR*. Les applications pouvant bénéficier de ce type de *SGBDTR* (ayant comme critère l'*ESR*) doivent évidemment tolérer des imprécisions sur les données et/ou les résultats. Il s'agit notamment des applications où un résultat obtenu dans le temps (même partiel ou imprécis), peut être préférable à un résultat complet et précis obtenu en retard. Par exemple, dans les systèmes de prise de décision (marchés boursiers, décision de suivre telle ou telle trajectoire pour un robot, ...).
Afin d'être complète, notre étude a été validée par une implémentation d'un algorithme de contrôle de divergence. Au moyen de cet algorithme, nous avons pu mettre en évidence les améliorations qu'apporte l'*ESR* par rapport à la *SR*. En effet, l'incohérence autorisée par l'*ESR* augmente le nombre d'*ETs* qui se terminent avant échéance et améliore les critères de performance des applications reposant sur un *SGBDTR*.
L'*ESR* traite le problème du coté des transactions. Notre objectif est d'étendre nos travaux pour prendre en compte la spécification de la qualité de service au travers de la qualité des transactions et de la qualité des données. Dans nos futurs travaux, nous pensons utiliser le paramètre ε de l'*ESR* pour faire varier les critères de qualité de service en fonction des besoins des applications.

REMERCIEMENTS

Ces travaux ont été effectués dans le cadre de l'ACI – Jeune Chercheur 1055.

BIBLIOGRAPHIE

Bernstein, P., Hadzilacos, V., Goodman, N. (1987). *Concurrency control and recovery in database systems.* Addison-Wesley.

Date, C.S. (1985). *An Introduction to Database Systems*. Addison-Wesley.

Duvallet, C., Mammeri, Z., Sadeg, B. (1999). *Les SGBD Temps Réel.* Technique et Science Informatiques, 18(5): 479–517.

Kamath, M., Ramamritham, K. (1993). *Performance Characteristics of Epsilon Serialisability with Hierarchical Inconsistency Bounds*. In Proceedings of the 9th International Conference on Data Engineering. pages 587–594. IEEE Computer Society Press.

Liu, C., Leyland, J. (1973). *Scheduling algorithms for multiprogramming in hard real-time environment.* Journal of the ACM, 20(1): 46–61.

Lam, K., Yau, W. (1998). *On Using Similarity for Concurrency Control in Real-Time Database Systems*. Journal of Systems and Software, 43(3): 223–232,

Pu, C. (1991). *Generalized Transactions Processing with Epsilon Serializability*. In Proceedings of 4th International Workshop on High Performance Transactions Systems, Asilomar, California.

Pu, C., Leff, A. (1992). *Autonomous Transaction Execution with Epsilon Serializability*. In Proceeding of 1992 RIDE Workshop on Transaction and Query Processing. IEEE Computer Society.

Ramamritham, K., Chrysanthis, P.K. (1992). *A Taxonomy of Correctness Criteria in Database Applications*. Journal of Very Large Databases, 4(1): 85–97.

Ramamritham, K. (1993). *Real-Time Databases*. Journal of Distributed and Parallel Databases, 1(2): 199–226.

Ramamritham, K., Chrysanthis, P. K. (1993). *In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties,* Chapter in "Distributed Object Management", pages 211–229. Morgan Kaufmann Publisher.

Ramamritham, K., Pu, C.A. (1995). *Formal Characterization of Epsilon Serialisability*. IEEE Transaction Knowledge and Data Engineering, 7(6): 997–1007.

Sadeg, B., Amanton, L., Haubert, J. (2003). *Tradding Precision for Timeliness in Distributed Real-Time Databases*. In Proceedings of 5th International Conference on Enterprise Information System (ICEIS'2003), Anger, France.

Son, S.H., Kouloumbis, S. (1993). *A Token-Based Synchronization Scheme Using Epsilon-Serialisability and its Perfo-rmance for Real-Time Distributed Database Systems.* In Proceedings of 3th International Symposium on Database Systems for Advanced Applications, Korea.

Tsang, M.K., Wu, K.L., Yu, P.S. (1997). *Multiversion Divergence Control Time Fuzziness*. Technical Report No. OR 97291-1000, Department of Computer Science and Engineering, Yorktown Heights, NY 10598.

Ullman J.D. (1980). *Principles of Database Systems*. Computer Science Press, Inc., Maryland.

Ulusoy, Ö. (1995). *Research Issues in Real-Time Database Systems*. Information Sciences, 87: 123–151.

Wu, K.L., Yu, P.S., Pu. C. (1992). *Divergence Control Algorithms for Epsilon Serialisability*. In Proceedings of 8th International Conference on Data Engineering, pages 506–515. IEEE Computer Society.

ACCO_CF/RTT : UN ALGORITHME DE CONTROLE DE CONCURRENCE OPTIMISTE POUR LES RELATIONS TEMPORELLES DE TRANSACTION

Rafik BOUAZIZ,
Maître assistant en informatique
Raf.Bouaziz@fsegs.rnu.tn

Achraf MAKNI,
Doctorant en informatique
Achraf.Makni@fsegs.rnu.tn

Adresse professionnelle
Faculté des Sciences Economiques et de Gestion de Sfax ★B.P. 1088★ 3018 Sfax, Tunisie

Résumé : ACCO_CF/RTT est un nouvel algorithme optimiste de contrôle de concurrence pour les relations temporelles de transaction. Cet algorithme procède à l'estampillage des transactions par leur instant d'arrivée et à la notification de toute opération réalisée sur les données. Il valide toute transaction terminée et devenant prioritaire, en utilisant la technique de photos de transactions et celle de marquage de fin de transaction. Les risques d'avortement non justifiés sont ainsi évités.

Summary : ACCO_CF/RTT is a new optimistic concurrency control algorithm for transaction-time relations. This algorithm proceeds to timestamping transactions by their arrival moments and to notify any operation realized on data. It validates any terminated transaction that has the most priority, by using the techniques of transaction snapshots and end of transaction marker. The probabilities of no proof abortion are then eliminated.

Mots clés : Contrôle de concurrence, Méthode optimiste de contrôle de concurrence, Cohérence forte, Bases de données temporelles, Relation temporelle de transaction.

Key words : Concurrency control, optimistic concurrency control method, strong consistency, temporal databases, transaction-time relation.

ACCO_CF/RTT : Un algorithme de contrôle de concurrence optimiste pour les relations temporelles de transaction

1 - INTRODUCTION

La cohérence d'une base de données (BD) exige que le contrôle de concurrence (CC) garantisse une exécution simultanée des transactions qui produit les mêmes résultats que leur exécution séquentielle [Gardarin (1988)]. La cohérence forte d'une BD exige que le CC garantisse une exécution simultanée des transactions qui produit les mêmes résultats que l'exécution séquentielle des transactions dans l'ordre strict de leur arrivée [Rahgozar (1987)].

Le CC prend de nouvelles dimensions lorsqu'on veut l'appliquer aux BD temporelles (BDT), ayant pour objectif de gérer l'historique des données. Peu de travaux ont abordé les problèmes liés à cet aspect [Finger et McBrien (1997)] [Elloumi, Bouaziz et Moalla (1998)] [Castro (1998)]. Nous proposons dans cet article un nouvel algorithme optimiste permettant d'assurer la cohérence forte pour les relations temporelles de transaction (RTT).

La section 2 présente l'état de l'art du CC et de son application aux BDT. La section 3 décrit l'environnement et les objectifs de l'algorithme proposé, ACCO_CF/RTT, dont la structuration est discutée au niveau de la section 4 : définition et ordonnancement des tâches du contrôleur de concurrence pour une RTT, procédure de certification et procédure de validation. La section 5 est consacrée à l'analyse d'un cas d'application de cet algorithme. Finalement, un prototype est présenté dans la section 6.

2 - ETAT DE L'ART DU CC ET DE SON APPLICATION AUX BDT

A notre connaissance, les travaux récents concernant le CC sont peu nombreux. Ils se sont intéressés :

- aux BDT [Finger et McBrien (1997)] [Elloumi, Bouaziz et Moalla (1998)] [Castro (1998)] [Makni et Bouaziz (2003)] qui constituent le domaine de notre contribution ;
- aux BD réparties et aux applications temps réel, avec l'introduction du concept de "epsilon-sérialisation". Ce nouveau critère de correction autorise l'introduction de certaines incohérences, mais bornées et contrôlées. Une transaction poursuit son exécution malgré l'apparition d'incohérences tant que la quantité d'incohérence permise n'est pas dépassée [Lee et Lam (2000)] [Amanton, Sadeg et Haubert (2003)].

Les méthodes de contrôle des accès concurrents sont classifiées dans la littérature en deux grandes catégories :

- Les méthodes pessimistes où le contrôle de cohérence est effectué lors de chaque opération d'une transaction [BERNSTEIN et GOODMAN (1980)] [Miranda et Busta (1986)] [Kumar (1989)] [Yu, Dias et Lavenberg (1990)].
- Les méthodes optimistes où le contrôle de cohérence n'est effectué qu'à la fin de la transaction. Ceci veut dire que ces méthodes autorisent l'exécution simultanée des transactions sans aucun contrôle préalable. C'est au moment de la validation que la cohérence de la base est testée et, le cas échéant, corrigée [Menasce et Nakanishi (1982)] [Chan (1986)] [Yu, Dias et Lavenberg (1990)].

Avec un mécanisme de contrôle de concurrence optimiste, le traitement d'une transaction comporte trois phases : phase de lecture, phase de validation et phase d'écriture.

La validation peut avoir lieu soit après la phase de lecture, soit au cours de cette phase. Selon le moment de la validation, deux types de méthodes optimistes peuvent être envisagés [Yu, Dias et Lavenberg (1990)] : pure optimistic method (POM) et broadcast optimistic method (BOM).

POM, la méthode originale [Kung et Robinson (1979)], adopte une stratégie de validation

basée sur l'histoire des transactions validées. POM présente l'inconvénient d'une définition sévère du conflit, d'où le risque d'avortement non nécessaire de transactions. Cette approche est améliorée par l'introduction de la méthode de marquage de fin de transaction "EOT Marker" qui a permis de surmonter le problème de la définition sévère du conflit.

Quant à la méthode BOM, elle effectue la validation par les photos de toutes les transactions concurrentes qui sont dans leur phase de lecture. Elle permet de détecter les conflits plus tôt que dans les autres méthodes ; une transaction ne doit pas terminer son exécution si on détecte un conflit. Deux variantes de cette méthode ont été proposées :

- BOM avec section critique (SC) : les phases de validation et d'écriture forment ensembles une SC, durant laquelle la BD est verrouillée ; aucune transaction n'a le droit d'y accéder.
- BOM sans SC.

BOM avec SC permet de choisir la transaction à avorter en cas de conflit, ce qui permet d'éviter le problème de famine (starvation problem) qui peut se poser dans les autres méthodes.

Pour accroître le degré de parallélisme, un algorithme de CC peut procéder à la tenue de plusieurs versions pour chaque granule. C'est le cas de l'algorithme que nous proposons ici. Cet algorithme concerne les relations temporelles de transaction (RTT) et exploite les versions tenues par ces relations elles-mêmes.

L'objectif de ces relations consiste à fournir aux applications, devenues nombreuses, non seulement les données courantes, mais aussi tous les états qui se succèdent dans le temps. Pour pouvoir maintenir ces états, il faut que les opérations de mise à jour soient non destructives, tel que c'est proposé, entre autres, par les RTT d'une BDT. Les RTT procèdent au stockage des versions dépassées en les estampillant par les deux temps physiques suivants :

- "temps de début de transaction (TDT)" : temps d'exécution de la transaction qui insère le tuple correspondant. Ce temps est connu a priori.
- "temps de fin de transaction (TFT)" : temps d'exécution de la transaction qui met à jour ou supprime le tuple considéré. Il n'est pas donc connu a priori.

L'intervalle formé par ces deux attributs représente l'intervalle de temps pendant lequel le tuple était le tuple courant [Bouaziz (1991)].

Du fait que ces attributs ne sont jamais modifiés par l'utilisateur, il serait alors possible de rendre compte de l'état de la base à n'importe quel instant du passé, et en particulier pour les états incohérents qui, eux non plus, ne sont pas détruits. Mais, sachant que le temps de début de transaction d'un tuple correspond à l'instant où il est inséré dans la base, il n'est pas possible d'enregistrer des mises à jour postactives, pour insérer des tuples qui se rapportent au futur, ou rétroactives, pour corriger le passé.

Dans le cadre d'une RTT, l'application d'un algorithme de CC qui exploite les anciennes versions des granules maintenues dans la BD serait l'un des buts à atteindre.

Elloumi, Bouaziz et Moalla (1998) ont étudié la possibilité d'appliquer les algorithmes pessimistes dans le cadre des RTT et ont proposé, par la suite, un nouvel algorithme de CC pessimiste qui permet d'assurer la cohérence forte de la BD. Cet algorithme exige une connaissance a priori des granules à manipuler par toute transaction de mise à jour, ce qui constitue une difficulté de mise en œuvre.

Un autre algorithme pessimiste a été proposé dans [Finger et McBrien (1997)]. Il assure une sérialisation temporelle qui permet de garantir une exécution sérialisable des transactions dans l'ordre strict de leurs estampilles (qui correspondent à leurs instants d'arrivée et qui marquent la maturité de ces transactions). Ainsi, 2PL est étendu avec un mécanisme d'ordonnancement de maturité. Dans cet algorithme :

- pour deux transactions Ti et Tj avec $i < j$, si Tj a obtenu un verrou sur un granule x non encore libéré et si Ti demande un verrou non compatible sur x, alors Tj est avortée ;
- une transaction ne peut être validée qu'après validation de toute transaction plus vieille (plus prioritaire) qu'elle.

Cet algorithme a été amélioré par l'utilisation de connaissances a priori, ce qui a permis à une transaction T de valider ses traitements même

si d'autres plus vieilles sont encore en exécution, à condition que ces dernières n'ont pas déclaré a priori des verrous en conflit avec T. Cependant, l'algorithme ne garantit plus la cohérence forte.

Par ailleurs, Castro (1998) a étudié l'amélioration des algorithmes de CC classiques dans les BDT, en essayant de cerner les conflits effectifs sur les données impliquées en fonction des portions temporelles, afin d'optimiser le temps d'exécution global des opérations. Chaque transaction est découpée en deux sous-transactions pour isoler les traitements qui s'adressent à la portion temporelle des données, objet de conflit. La sous-transaction conflictuelle doit être sérialisée avec les sous-transactions concurrentes. Cependant, le concept de l'atomicité d'une transaction est révisé.

Par contre, les algorithmes optimistes n'ont pas été, à notre connaissance, étudiés pour un environnement d'une RTT. Notre travail se propose de le faire afin de construire un algorithme optimiste permettant d'assurer la cohérence forte et d'accroître le parallélisme.

3 - ENVIRONNEMENT ET OBJECTIFS DE L'ALGORITHME PROPOSE

Pour étudier le contrôle de concurrence, nous admettons l'hypothèse adoptée dans [Bernstein, Hadzilacos et Goodman (1987)] consistant à considérer tout système de bases de données (SBD) comme étant formé, tel que le montre la figure 1, des trois modules suivants :

- **Un gestionnaire des transactions** (ou contrôleur d'exécution) qui traite les opérations des transactions reçues ; c'est à travers lui que s'effectue l'interaction entre les transactions et le SBD. Le gestionnaire des transactions lance l'exécution en mode multitâche (que l'on appelle par abus de langage "parallèle") de plusieurs transactions dans différents processus. Lorsqu'une transaction arrive au système, elle est d'abord estampillée par l'instant de son arrivée et stockée dans une file d'attente, puis affectée à un processus dès que le gestionnaire des transactions en trouve un de libre.

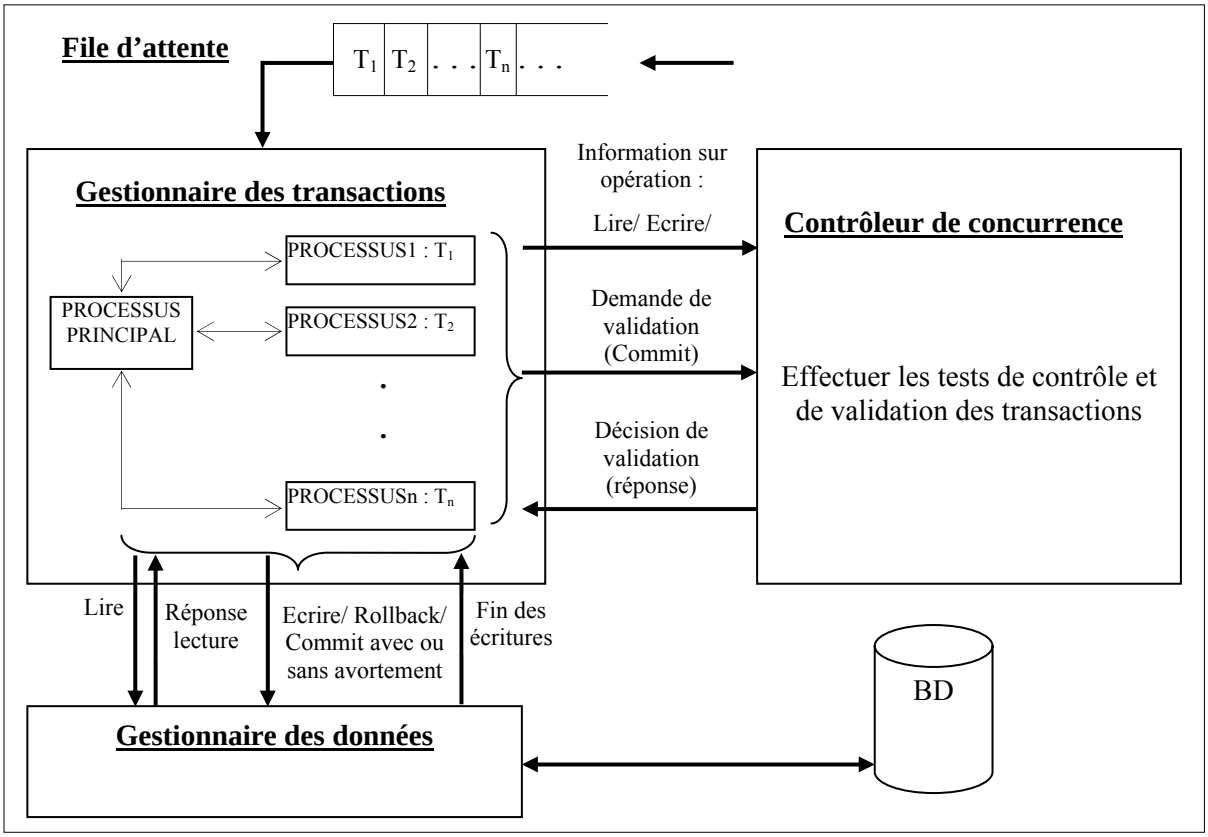


Figure 1. Architecture d'un système de bases de données

- **Un contrôleur de concurrence** qui surveille l'exécution concurrente des transactions afin que cette exécution soit sérialisable. Il a pour rôle d'effectuer les

tests de contrôle et de validation des transactions lancées par le gestionnaire des transactions. Le contrôleur de concurrence communique avec les processus en exécution. En effet, à chaque opération Lire, Ecrire ou Rollback et à chaque demande de validation (Commit), ce contrôleur effectue un traitement approprié.

- **Un gestionnaire des données** qui assure la répercussion sur la BD de tous les effets des transactions validées. C'est lui qui réalise les opérations Lire, Ecrire, Rollback et Commit.

Pour exécuter une opération de lecture, d'écriture ou d'annulation (Rollback), le gestionnaire des transactions fait exécuter cette opération par le gestionnaire des données et en informe le contrôleur de concurrence.

Pour exécuter une opération de validation sur la BD, le gestionnaire des transactions soumet cette opération au contrôleur de concurrence qui décide de l'une des actions suivantes :

- Faire exécuter l'opération par le gestionnaire des données.
- Retarder l'opération ; la transaction est alors mise dans une file d'attente interne pour être validée ultérieurement.
- Annuler la transaction concernée ou les transactions impliquées pour éviter des incohérences.

L'algorithme optimiste que nous nous proposons de concevoir doit garantir les objectifs suivants :

- Maintenir la cohérence forte du système d'information dans un environnement de RTT.
- Exploiter les versions dépassées d'une RTT au lieu de gérer des versions propres au contrôle de la concurrence.
- Minimiser le degré d'avortement des transactions.
- Eviter le problème de famine.
- Détecter les conflits le plus tôt possible.

4 - STRUCTURATION DE L'ALGORITHME ACCO_CF/RTT

4.1 - Définition et ordonnancement des tâches du contrôleur de concurrence pour une RTT

Pour chaque transaction T_i, le contrôleur de concurrence maintient deux ensembles : l'ensemble RS_i (read set), relatif aux objets lus par T_i, et l'ensemble WS_i (write set), relatif aux objets écrits par T_i.

Durant l'exécution d'une transaction, lorsque le contrôleur de concurrence reçoit :

- une opération Lire (T_i, g), il ajoute le granule g à l'ensemble des objets lus ;
- une opération Lire (T_i, g, pt), il n'ajoute le granule g à l'ensemble des objets lus que si pt indique la version courante de g. Si pt désigne une version dépassée, cette opération de lecture ne doit pas être prise en considération dans le test de validation, car elle ne peut en aucun cas produire des conflits ;
- une opération Ecrire (T_i, g), il ajoute le granule g à l'ensemble des objets écrits ;
- une opération Rollback, il élimine les objets lus et écrits de RS_i et WS_i ;
- une opération Commit, il vérifie s'il existe ou non un conflit entre la transaction à valider et les transactions qui ne sont pas encore validées.

Nous proposons de partir de la stratégie de validation des algorithmes BOM. Cette stratégie stipule qu'à chaque exécution, à un instant t, de la commande COMMIT d'une transaction T_i, le contrôleur de concurrence doit effectuer un test de validation de T_i vis à vis des transactions concurrentes qui sont encore dans leur phase de lecture. Pour chacune de ces dernières, le contrôleur de concurrence doit donc examiner l'intersection de l'ensemble des objets qu'elle a lus, jusqu'à l'instant t, avec celui des objets écrits par T_i. Si l'intersection n'est pas vide, c'est qu'il existe un conflit entre les deux transactions. La transaction à avorter est la moins prioritaire.

Afin de pouvoir assurer la cohérence forte, nous proposons de procéder à **l'estampillage des transactions par les instants de leur arrivée**, c'est la transaction la plus jeune qui doit être considérée comme la moins prioritaire. Cependant, d'après les études que nous avons effectuées, estampiller les transactions par les instants de leur arrivée ne suffit pas pour garantir la cohérence de la BD dans le cadre d'une RTT. De ce fait, cette stratégie ne suffit pas pour garantir la cohérence du système d'information dans le

cadre d'une RTT. Ceci est dû à la non synchronisation des transactions. Nous proposons donc de définir un **ordre de validation** entre les transactions lors de l'exécution de la commande COMMIT de toute transaction T_i. Par conséquent, le contrôleur de concurrence doit vérifier, avant de commencer le test de validation, que T_i est la transaction la plus prioritaire de l'ensemble des transactions qui ne sont pas encore validées. Nous proposons alors d'ajouter une **phase de certification** qui précède la phase de validation de chaque transaction.

4.2 - Structuration de la procédure de certification de ACCO_CF/RTT

A l'exécution de la commande COMMIT, T_i passe d'abord par un test de certification. Ce test vérifie que T_i est la plus prioritaire :

- Si ce n'est pas le cas, T_i est mise en attente pour être certifiée ultérieurement.
- Si T_i est la plus prioritaire, le contrôleur de concurrence effectue le test de validation de T_i avec toute transaction T_j en phase de lecture ou en attente de certification. Ces dernières sont forcément plus jeunes que T_i et donc moins prioritaires. Par conséquent :
  - S'il existe un conflit, c'est T_j qui doit être avortée pour être reprise avec la même estampille ;
  - Dans le cas contraire :
    - Si T_j était en attente de certification, le contrôleur de concurrence réexécute de nouveau la procédure de certification pour T_j ;
    - Sinon, le contrôleur d'exécution poursuit l'exécution des instructions de T_j.

Une fois arrivée à sa phase de validation, T_i est validée d'office. Les nouvelles versions des granules manipulés par cette transaction seront enregistrées dans la BD et prendront comme estampille le temps de transaction de T_i, égal à t_i, instant d'arrivée de T_i.

Nous proposons que le test de certification, qui accompagne l'exécution de l'opération COMMIT de toute transaction T_i, soit structuré comme suit :

```
CERTIFICATION (Ti) : entier
   Début
      /* Certification de Ti avec les transactions en attente de certification */
      v := CERTIF_TA (Ti)
      Si v = 1 Alors
      /* Certification de Ti avec les transactions qui sont encore en phase de lecture */
         v := CERTIF_TL (Ti)
      Fin Si
      Retourner (v)
   Fin.
```

Les fonctions "CERTIF_TA" et "CERTIF_TL" permettent de certifier une transaction respectivement par rapport aux transactions en attente de certification et par rapport aux transactions en phase de lecture. Sachant que ces transactions sont ordonnées dans leur liste selon l'ordre croissant de leurs estampilles, ces fonctions sont structurées comme suit :

```
CERTIF_TA (Ti) : entier
   Début
      v := 1
      Si il existe des transactions en attente de certification Alors
         "j prend l'estampille de la première transaction en attente de certification : c'est la transaction la plus prioritaire parmi celles qui sont en attente"
         Si i > j Alors
            Ti est mise en attente de certification
            v := 0
         Fin Si
      Fin Si
      Retourner (v)
   Fin.

CERTIF_TL (Ti) : entier
   Début
      v := 1
      Si il existe des transactions en phase de lecture Alors
         "j prend l'estampille de la première transaction en phase de lecture : c'est la transaction la plus prioritaire parmi celles qui sont en exécution "
         Si i > j Alors
            Ti est mise en attente de certification
            v := 0
         Fin Si
      Fin Si
      Retourner (v)
   Fin.
```

4.3 - Structuration de la procédure de validation de ACCO_CF/RTT

4.3.1 - Validation sous BOM sans SC pour une RTT

Le test de validation de BOM sans SC commence par attribuer un numéro de transaction à T_i, à travers un compteur global. Ce test, effectué avec chaque transaction concurrente T_j, dépend du fait que T_j est en phase de validation ou en phase de lecture.

Mais dans un environnement de RTT, il est nécessaire que l'on procède à l'estampillage des transactions par les instants de leur arrivée, afin de pouvoir assurer la cohérence forte. La procédure de validation doit utiliser ces estampilles et n'a pas donc à attribuer des numéros de transactions. Par ailleurs, le test de validation ne peut concerner que les transactions concurrentes qui n'ont pas encore commencé leur phase de validation. En effet, on ne peut jamais avoir, à un instant donné, deux transactions concurrentes en phase de validation, puisque les transactions passent d'abord par un test de certification. Par conséquent, la procédure de validation vérifie l'absence de conflit uniquement avec les transactions en phase de lecture, y compris celles en attente de certification.

Le test de validation devient alors comme suit :

VALIDATION_SSC (T_i)
Début
"Donner l'accord pour répercuter les écritures sur la BD"
/* Sachant que toutes les transactions qui précèdent Ti devaient être déjà validées */
"j prend l'estampille de la première transaction qui suit T_i"
Tantque j <> "!" **Faire**
Si (CONFLIT (WS_i, RS_j) = 0) **Alors**
"Donner un ordre d'avortement de T_j : ROOLBACK T_j"
Fin Si
"j prend l'estampille de la transaction suivante"
Fin Tantque
Fin.

La fonction "CONFLIT", appelée par la procédure de validation, teste l'absence de conflit entre deux transactions. Elle retourne la valeur 1 si le résultat du test est positif, 0 sinon. Elle est définie comme suit :

CONFLIT (WS_i, RS_j) : entier
Début
r := 1
Si $WS_i \cap RS_j \neq \varnothing$ **Alors**
r := 0
Fin Si
Retourner (r)
Fin.

4.3.2 - Validation sous BOM avec SC pour une RTT

Le test de validation de BOM avec SC comporte une SC durant laquelle il fait appel à la fonction "CONFLIT" et donne l'accord pour répercuter les écritures sur la BD. Toutes les transactions sont donc bloquées durant cette SC. Ce test est alors défini comme suit :

VALIDATION_ASC (T_i)
Début
/* Début de la section critique, annoncée par le symbole < */
< "j prend l'estampille de la première transaction qui suit T_i"
Tantque j <> "!" **Faire**
Si (CONFLIT (WS_i, RS_j) = 0) **Alors**
"Donner un ordre d'avortement de T_j : ROOLBACK T_j"
Fin Si
"j prend l'estampille de la transaction suivante"
Fin Tantque
"Donner l'accord pour répercuter les écritures sur la BD" >
/* Fin de la section critique, annoncée par le symbole > */
Fin.

4.3.3 - Comparaison des validations sous BOM avec et sans SC pour une RTT

Nous constatons que les deux tests de ces deux approches sont composés de deux lots de codes identiques : la vérification de l'absence de conflit et l'accord pour répercuter les écritures sur la BD. Cependant, ils diffèrent sur les deux points suivants :

- L'ordre d'exécution des deux lots : la première approche donne l'accord pour répercuter les écritures sur la BD, puis vérifie l'absence de conflit. La deuxième approche inverse ces lots. Mais cette différence n'a pas d'importance puisque les deux lots se sont avérés indépendants dans le contexte du nouvel algorithme,

contrairement aux contextes des deux variantes originales de BOM. En effet, puisque la transaction à valider ne peut être que la plus prioritaire, alors la répercussion des écritures sur la BD est à exécuter dans tous les cas et indépendamment du résultat de la vérification de l'absence de conflit. Par conséquent, l'ordre d'exécution de ces deux lots n'a pas ici d'influence sur le test de validation.

- La présence de la SC dans la deuxième approche. Vérifions si la présence de la SC est obligatoire ou non. Considérons deux transactions T_i et T_j où T_i est plus prioritaire que T_j. T_i arrive à sa phase de validation alors que T_j est encore dans sa phase de lecture et ces deux transactions ne sont pas, jusque là, en conflit. Supposons qu'après la validation de T_i, T_j effectue une opération de lecture d'un granule manipulé en écriture par T_i.

Si nous appliquons le test de validation avec BOM avec SC, T_i et T_j ne seront pas en conflit. L'opération de lecture de cette dernière sera effectuée sur la nouvelle version créée par T_i.

Si nous appliquons le test de validation avec BOM sans SC, T_i et T_j peuvent être signalées, à tort ou à raison, en conflit. En effet, en l'absence d'une SC, pendant que T_i n'est pas encore arrivée à vérifier l'absence de conflit avec T_j, cette dernière continue son exécution. Elle peut donc atteindre la dite opération de lecture. Cette lecture peut concerner d'une manière aléatoire :

- Soit la nouvelle version du granule créée par T_i ; dans ce cas, il n'existe pas d'incohérence.
- Soit la version précédente ; dans ce cas, il existe une incohérence.

Mais dans les deux cas, le test de validation signale l'existence d'un conflit car il utilise l'ensemble des objets lus de T_j (RS_j) après cette lecture. D'où une définition sévère du conflit.

Par conséquent, ceci implique que l'adaptation de l'approche de BOM sans SC peut conduire à des avortements non justifiés de transactions.

4.3.4 - Procédure de validation

Compte tenu de l'étude du paragraphe précédent, nous proposons de partir de l'approche BOM avec SC et d'imposer un ordre pour la validation des transactions afin de pouvoir garantir la cohérence forte dans le cadre de RTT. Aucune transaction ne peut être validée s'il existe des transactions qui la précèdent et qui n'ont pas été encore validées.

Après avoir validé une transaction T_i, on doit toujours vérifier s'il existe une transaction T_j en attente de certification qui est devenue la plus prioritaire. En effet, la mise en attente de certification de T_j est causée par l'existence d'autres plus prioritaires non encore validées. Deux possibilités peuvent avoir lieu :

- Si T_i était la seule transaction plus prioritaire que T_j, cette dernière, en passant de nouveau le test de certification, s'avère la plus prioritaire ; elle passe donc à la phase de validation.
- Autrement, le test de certification de T_j échoue et elle reste en attente de certification jusqu'à validation de toutes les transactions prioritaires.

Nous proposons alors d'ajouter une fonction de réveil qui sélectionne, le cas échéant, la transaction la plus prioritaire de l'ensemble des transactions mises en attente de certification pour repasser de nouveau le test de certification. Cette fonction doit être exécutée après chaque validation de toute transaction. Nous proposons donc de l'appeler juste après la procédure de validation. Elle est donc exécutée en dehors de la SC du test de validation, ce qui permet d'améliorer le parallélisme.

Il nous reste maintenant de traiter le risque de prolongement de la SC. Cette dernière s'étend durant les deux phases de validation et d'écriture de la transaction T_i. Durant cette période, tous les granules manipulés en écriture par T_i sont verrouillés. Comment alors peut-on réduire cette période ?

La réduction de cette période par l'exclusion de l'une des deux phases de la SC pose des problèmes. En effet :

- Durant la phase d'écriture, tout granule g manipulé en écriture par T_i (transaction à valider) doit être inaccessible par toute transaction concurrente T_j ; cette dernière n'a le droit ni de lire ni d'écrire g pour éviter toute incohérence. Par conséquent, la SC doit couvrir toute la durée de la phase d'écriture.

- Durant la phase de validation de T_i, les ensembles des objets lus des transactions concurrentes doivent être figés. Dans le cas contraire, une définition sévère du conflit peut apparaître.

Cependant, nous pouvons réduire la période de la phase de validation en faisant appel à la technique utilisée par les EOT marker pour une définition correcte des conflits. EOT marker se base sur le principe qu'une transaction concurrente n'a pas besoin d'être bloquée sur l'utilisation d'un granule pour pouvoir définir correctement un conflit. Nous pouvons alors tirer profit de cette technique afin d'assurer une définition correcte de conflit tout en garantissant la cohérence de la BD. Si nous utilisons EOT marker, nous n'aurons pas besoin de verrouiller la BD durant la phase de validation de T_i. Ceci est dû au fait que chaque transaction T_j en phase de lecture prend note de la fin de la transaction T_i à valider. Ainsi, quand cette dernière passe le test de certification avec succès, le contrôleur marque la fin de T_i dans l'ensemble des objets lus par T_j qui continue son exécution. Au moment de la vérification de l'absence de conflit T_i/T_j, l'ensemble des objets lus de T_j concerné par le test de validation se limite aux objets lus du début jusqu'à la marque de fin de T_i (EOT_i).

Appliquons cette technique sur l'exemple du paragraphe précédent, où nous avons constaté une définition sévère du conflit dans le cas de BOM sans SC. Il s'agit de deux transactions T_i et T_j où T_i est plus prioritaire que T_j. T_i arrive à sa phase de validation alors que T_j est encore dans sa phase de lecture et ces deux transactions ne sont pas, jusque là, en conflit. Supposons qu'après la validation de T_i, T_j effectue une opération de lecture d'un granule g manipulé en écriture par T_i.

Si nous intégrons EOT marker, le scénario d'exécution sera comme suit : au moment où T_i arrive à sa phase de validation, T_j en prend note dans son ensemble RS_j. Supposons que RS_j, avant cet instant est égal à {x, y, …, z}. Lorsque T_j prend note de la fin de T_i, RS_j devient {x, y, …, z, EOTi}. Une fois que T_j effectue l'opération Lire (T_j, g), le granule g s'ajoute à RS_j, devenant comme suit : {x, y, …, z, EOT_i, g}. Au moment où le contrôleur de concurrence effectue le test d'absence de conflit T_i/T_j, l'ensemble des objets lus de T_j, concerné par ce test, commence du début de RS_j jusqu'à EOT_i : c'est {x, y, …, z}. Ceci évite l'avortement, à tort, de T_j.

Puisque l'intégration de EOT marker permet de définir correctement les conflits sans avoir besoin de verrouiller la BD dans la phase du contrôle, nous proposons alors que la SC se restreint à la phase d'écriture et à la période de marquage de fin de transaction, sachant que cette dernière est beaucoup plus courte qu'une phase de validation. La procédure de validation que nous proposons est définie alors comme suit :

VALIDATION (T_i)
Début
/* Début de la SC */
< "Donner l'accord pour répercuter les écritures sur la BD"
"j prend l'estampille de la première transaction qui suit T_i"
Tantque j <> "!" **Faire**
"Ajouter l'élément EOT_i à RS_j"
Fin Tantque
> /* Fin de la SC */
"j prend l'estampille de la première transaction qui suit T_i"
Tantque j <> "!" **Faire**
/* $RS_j(T_i)$ est l'ensemble des objets lus de T_j jusqu'à EOT_i */
Si (CONFLIT (WS_i, $RS_j(T_i)$) = 0)
Alors
"Donner un ordre d'avortement de T_j : ROOLBACK T_j"
Fin Si
"j prend l'estampille de la transaction suivante"
Fin Tantque
Fin.

5 - ANALYSE D'UN CAS D'APPLICATION DE L'ALGORITHME PROPOSE

Considérons les granules x, y et z d'une RTT et les transactions TA, TB et TC arrivant au système respectivement aux instants t_1, t_2 et t_3. Ces transactions sont ainsi définies :

TA	TB	TC
Lire (x)	Lire (x)	Lire (x, pt1)
.	Lire (y)	Lire (y, pt2)
Ecrire (x)	.	.
Lire (z)	Ecrire (y)	Ecrire (y)
Ecrire (z)	.	Commit
Commit	Commit	

Supposons qu'il existe 5 versions du granule x, 7 versions du granule y et 3 versions du granule z. Supposons aussi que pt1 appartient à l'intervalle de transaction de la première version de x et pt2 appartient à celui de la troisième version de y.

Notons par ti le temps d'exécution d'une instruction et par g_m^n la version N° n du granule g qui a comme estampille m (le TT). Une exécution sérialisable de ces transactions, basée sur l'application du nouvel algorithme de contrôle de concurrence, peut être comme celle présentée au tableau 1.

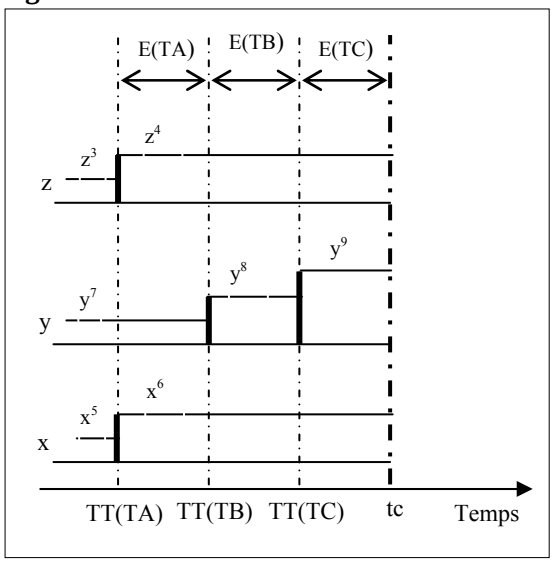
t	TA	TB	TC	Commentaires et suivi des RS et des WS
t_1	Arrivée de TA			
t_2		Arrivée de TB		
t_3			Arrivée de TC	
t_4	Lire (x)			$RS_A = \{x^5\}$
t_5		Lire (x)		$RS_B = \{x^5\}$
t_6			Lire (x, pt1)	pt1 désigne une ancienne version de x et non pas sa version courante : Exécuter la lecture sans prendre note dans RS_C
t_7	Ecrire (x)			$WS_A = \{x_{t1}\}$
t_8		Lire (y)		$RS_B = \{x^5, y^7\}$
t_9			Lire (y, pt2)	pt2 désigne une ancienne version de y et non pas sa version courante : Exécuter la lecture sans prendre note dans RS_C
t_{10}	Lire (z)			$RS_A = \{x^5, z^3\}$
t_{11}		Ecrire (y)		$WS_B = \{y_{t2}\}$
t_{12}			Ecrire (y)	$WS_C = \{y_{t3}\}$
t_{13}	Ecrire (z)			$WS_A = \{x_{t1}, z_{t1}\}$
t_{14}		Commit		Le test de certification de TB échoue car TA est plus prioritaire : mise en attente de TB.
t_{15}			Commit	Le test de certification de TC échoue car TA et TB sont plus prioritaires : mise en attente de TC.
t_{16}	Commit Validation de TA et création des nouvelles versions x_{t1}^6 et z_{t1}^4			- TA est la plus prioritaire, elle passe le test de certification avec succès. Elle est validée. $RS_B = \{x^5, y^7, EOT_A\}$ $RS_C = \{ EOT_A\}$ - $WS_A \cap RS_B \neq \varnothing$ - $WS_A \cap RS_C = \varnothing$ - Avortement de TB qui doit être reprise. - Exécution de la procédure Réveil : TC non certifiée car TB est plus prioritaire.
t_{17}		Lire (x)		$RS_B = \{x^6\}$
t_{18}		Lire (y)		$RS_B = \{x^6, y^7\}$
t_{19}		Ecrire (y)		$WS_B = \{y_{t2}\}$
t_{20}		Commit Validation de TB et création de la nouvelle version y_{t2}^8		- TB est la plus prioritaire, elle passe le test de certification avec succès. Elle est validée. $RS_C = \{EOT_A, EOT_B\}$ - $WS_B \cap RS_C = \varnothing$ - Exécution de la procédure Réveil : réveil de TC.
t_{21}			Commit Validation de TC et création de la nouvelle version y_{t3}^9	- TC est la seule transaction non encore terminée, elle passe le test de certification avec succès. Elle est validée. - Aucune transaction concurrente pour le test de validation. - Exécution de la procédure Réveil : aucune transaction en attente.

Tableau 1 : Exemple d'exécution de transactions concurrentes

La figure 2 donne l'ensemble des états pris par la base suite à cette exécution sérialisable. Sur cette figure :

- L'état E(TA) est formé par les nouvelles versions de x et de z (x^6et z^4) et l'ancienne version de y (y^7) qui continue à être courante. Cet état s'étend de TT(TA) jusqu'à TT(TB). Les nouvelles versions x^6et z^4 des granules x et z ont pris effet à partir de l'instant TT(TA).
- L'état E(TB) est formé par la nouvelle version de y (y^8) et des anciennes versions de x et de z (x^6 et z^4) qui continuent à être courantes. Cet état s'étend de TT(TB) jusqu'à TT(TC). La nouvelle version y^8 du granule y a pris effet à partir de l'instant TT(TB).
- L'état E(TC) est formé par la nouvelle version de y (y^9) et des anciennes versions de x et de z (x^6 et z^4) qui continuent à être courantes. Cet état s'étend de TT(TC) jusqu'au temps courant (tc). La nouvelle version y^9 du granule y a pris effet à partir de l'instant TT(TC).

Figure 2. Suivi des états cohérents de la RTT suite à l'exécution du cas d'application



6 - PROTOTYPE REALISE

Le prototype, réalisé sous VC++6.0, utilise un ensemble de structures de données pouvant être appelées par les différents modules du SBD. Ces structures sont des listes chaînées permettant de mémoriser toutes les informations relatives aux transactions arrivées au système. Le contrôleur de concurrence ACCO_CF/RTT se charge de la gestion des listes des objets lus et écrits par les transactions pour la vérification de la cohérence.

L'interface montre en détail l'avancement de l'exécution des opérations des différentes transactions. L'instruction sélectionnée est celle en cours d'exécution. Les états que peut avoir une transaction durant son exécution sont également affichées ("en exécution", "non certifiée", "Mise en attente", …). L'affichage des ensembles des objets lus et écrits des transactions montre leur évolution suite à l'application des règles de gestion et les éventuels conflits afin de pouvoir suivre la vérification de l'absence de conflits. Enfin, pour mieux suivre le fonctionnement attendu du prototype, la succession de l'exécution des différents modules du contrôleur de concurrence est affichée. Nous nous limitons ici à la présentation de l'image-écran qui correspond à l'instant d'exécution t_{21} où on arrive à Commit (TC).

Puisque TB est la plus prioritaire des transactions en phase de lecture, elle passe le test de certification avec succès. Puis, la marque de la fin de TB est notée dans l'ensemble des objets lus de TC (unique transaction encore en attente de certification). La vérification de l'absence de conflit TB/TC montre qu'il n'existe pas de conflit. Lorsque la fonction REVEIL ( ) est exécutée, TC est la seule transaction dans le SBD et donc la plus prioritaire. Elle est alors réveillée et elle passe le test CERTIFICATION (T_j) vis-à-vis des transactions non encore validées. Le résultat du test est positif, TC est donc certifiée et validée (figure 3).

7 - CONCLUSION

L'algorithme optimiste ACCO_CF/RTT traite du contrôle de concurrence (CC) dans le cadre des relations temporelles de transaction (RTT) et assure leur cohérence forte. La phase de certification n'autorise une transaction T à passer à la phase de validation que si elle est la plus prioritaire parmi toutes les transactions qui sont encore en phase de lecture ou en attente de certification. Cet ordre de priorité correspond à l'ordre défini par l'estampillage des transactions par les instants de leur arrivée. Une fois arrivée à la phase de validation, la transaction T est validée d'office. En cas de conflit avec une autre transaction T', forcément plus jeune que T, c'est T' qui est à

avorter. La répercussion des écritures sur la BD se déroule durant une section critique (SC) pour pouvoir maintenir la cohérence des RTT. Durant cette SC, l'accès à l'ensemble des granules manipulés par la transaction à valider est interdit pour toutes les transactions concurrentes. Cependant, la SC est réduite au maximum, elle ne comprend pas la période de recensement de conflits.

L'algorithme est caractérisé par un degré de parallélisme élevé dû à sa nature optimiste, à l'exploitation de versions multiples maintenues dans les RTT sans se charger de leur gestion et à l'intégration de la technique EOT Marker qui a permis de diminuer le temps de blocage des transactions par la réduction de la SC. Par ailleurs, la stratégie de validation basée sur les photos des transactions concurrentes, améliorée par la technique EOT Marker, permet, d'une part, d'accélérer la détection des conflits et, d'autre part, d'éviter le risque d'avortements non justifiés. Ceci améliore nettement les temps d'exécution des transactions.

Cet algorithme se limite aux RTT et ne peut pas être appliqué en tant que tel aux relations temporelles de validité et aux relations bitemporelles. Cependant, il nous ouvre la voie à des études plus approfondies du CC pour les BD temporelles. Par ailleurs, nous prévoyons d'étudier la robustesse de cet algorithme, à travers la vérification formelle sous l'environnement SPIN et le langage PROMERA.

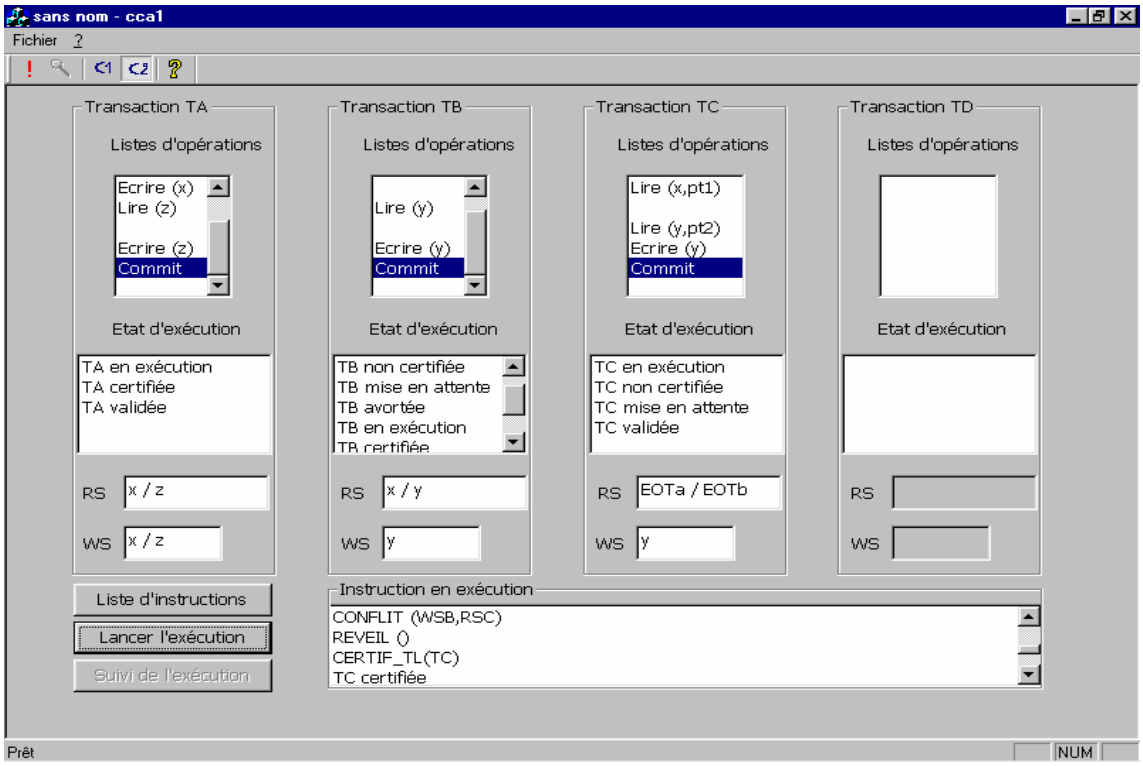


Figure 3. Réveil et validation de TC

Bibliographie

Amanton, L., Sadeg, B., Haubert, J. (2003), "Tradding Precision for Timeliness in Distributed Real-Time Databases", *International Conference on Enterprise Information Systems*, Vol 1, pp558-561.

Bernstein, P. A., Goodman, N. (1980), "Timestamp-based algorithms for concurrency control in distributed database systems", *Very Large DataBases* VLDB'80.

Bernstein, P. A., Hadzilacos, V., Goodman, N. (1987), "*Concurrency control and recovery in DBS* ", Edition ADDISON-WESLEY.

Bouaziz, R. (1991), "*Gestion temporelle et historisation des données dans les systèmes d'information*" , Thèse de doctorat de spécialité en informatique présentée à la Faculté des Sciences de Tunis, Université

des Sciences, des Techniques et de Médecine de Tunis.

Castro, C. (1998), "On concurrency management in temporal relational databases", *SEBD*, pp189-202.

Chan, M. Y. (1986), "Database concurrency control using READ/WRITE set information", in *Information Systems*, Vol 11, n° 4.

Elloumi, Dhouib, S., Bouaziz, R., Moalla, M. (1998), "Contrôle de concurrence multiversion dans les bases de données temporelles", *Bases de Données Avancées BDA'98*, pp135-155.

Finger, M., McBrien, P. (1997), "Concurrency Control for Perceivedly Instantaneous Transactions in Valid-Time Databases", in *TIME*, pp 112-118.

Gardarin, G. (1988), "*Bases de données : Les systèmes et leurs langages*", Edition EYROLLES.

Kumar, V. (1989), "A study of the behaviour of the read/write ratio under two-phase locking schemes", in *Information Systems*, Vol 14, n° 1.

Kung, H. T., Robinson, J. T. (1979) "On optimistic methods for concurrency control", *Very Large DataBases* VLDB'79.

Lee, V. C., Lam, K. W. (2000), "Conflict free transaction scheduling using serialisation graph for real-time databases", in *journal of Systems and Software*, Vol 55, n° 1, pp57-65.

Makni, A., Bouaziz, R. (2003), "Principe de base d'un algorithme optimiste de contrôle de concurrence d'accès", *Génie Electrique et Informatique*, Vol 1, pp207-211.

Menasce, D., Nakanishi, T. (1982), "Optimistic versus pessimistic concurrency control mechanisms in database management systems", in *Information Systems*, Vol 7, n° 1.

Miranda, S., Busta, J. M. (1986), "*L'art des bases de données : Les base de données relationnelles*", Edition EYROLLES.

Rahgozar, M. (1987), "*Contrôle de concurrence par gestion des événements*", Thèse de doctorat de l'Université de Paris 6.

Yu, S. P., Dias, D. M., Lavenberg, S. S. (1990), "*On modeling database concurrency control*", IBM Research Center, 195 RC 15386 (#68455).

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.

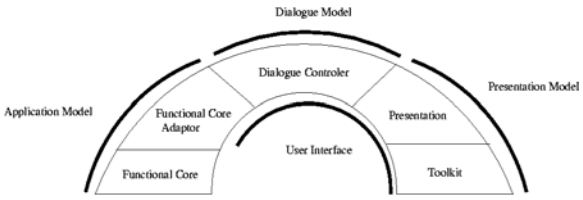


Figure 1. *ARCH et modèles d'interface*

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.

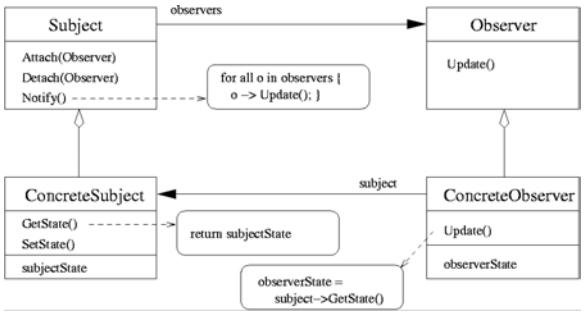


Figure 2. *Le design pattern Observer*.

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.

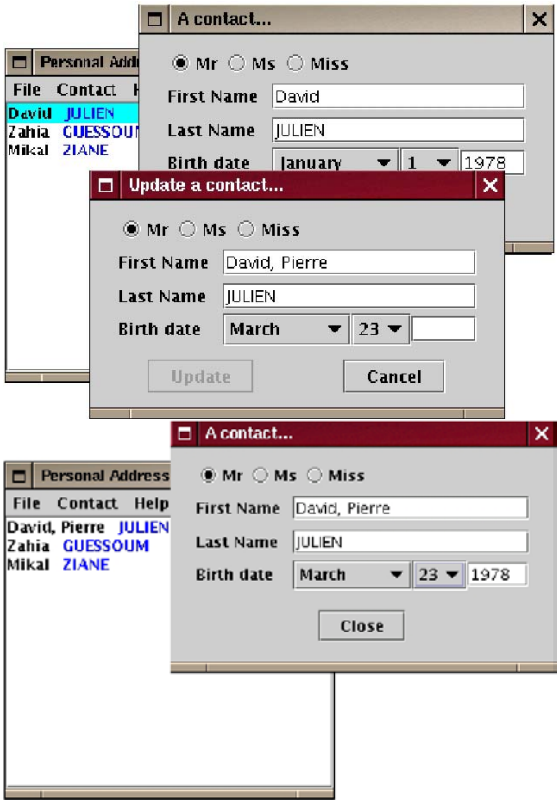


Figure 5. *En haut, l'interface avant modification. En bas, la même interface après mise à jour du contact. Les données ont été mises à jour automatiquement.*

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.

UNE APPROCHE ORIENTEE MODELE POUR LA GESTION DES ARCHITECTURES LOGICIELLES DISTRIBUEES DYNAMIQUES

Karim Guennoun,
Doctorant en Systèmes informatiques
guennoun@laas.fr , + 33 5 61 33 78 15

Khalil Drira,
Chargé de recherche CNRS
khalil@laas.fr + 33 5 61 33 78 86

Michel Diaz,
Directeur de recherche CNRS
diaz@laas.fr + 33 5 61 33 62 56

Adresse professionnelle
LAAS-CNRS, 7 Avenue du colonel Roche
31077 Toulouse Cedex 4, France

Résumé : Cet article présente les graphes abstraits de composants (Abstract Component Graph **ACG**), une approche visant à coordonner l'évolution dans les architectures logicielles dynamiques distribuées. Elle permet, en plus de spécifier l'évolution dynamique à l'échelle de la configuration de l'architecture, la spécification de l'évolution dynamique à l'échelle du composant. Elle peut être employée pour simuler les différentes étapes d'instanciation du composant, le changement de comportement pendant l'exécution, la migration, et d'autres caractéristiques spécifiques aux architectures logicielles des systèmes distribués. Cette approche permet également la vérification de propriétés à ces deux niveaux des spécifications (i.e. configuration de l'architecture et structure du composant).

Summary : This article presents the Abstract Component Graph (**ACG**), an approach which addresses the coordination issue for distributed dynamic software architectures. It allows, in addition of the specification of architecture dynamic changes on the configuration level, the specification of dynamic changes at the level of the component. The ACG approach can also be used to simulate the various steps of component instantiation, behaviour changes on the run-time, migration, and other characteristics specific to distributed software architectures. This approach allows checking properties at these two levels of specification (i.e. architecture configuration and component structure).

Mots clés : Architectures logicielles, Dyanmicité, Coordination, Graphes, Edition partagée.

Une approche orientée modèle pour la gestion des architectures logicielles distribuées dynamiques

1 - INTRODUCTION

La distribution et l'évolution dynamique des composants et des schémas d'interaction sont deux exigences importantes qui rendent les modèles traditionnels de l'architecture logicielle inadéquats pour la conception des nouveaux systèmes logiciels tels que ceux dédiés à la collaboration distribuée d'activités de groupe (Adler (1995)). De telles applications font de la spécification de l'évolution dynamique un domaine de recherche important traité par différents travaux tels que ceux réalisés par Allen, Douence, et Garlan (1998), et par Riveil et Senart (2002).

La programmation orientée composant, d'après Wegner (1993), est susceptible d'être le modèle le plus prometteur pour les systèmes logiciels de prochaine génération, où les techniques doivent traiter la dynamicité pour décrire l'activation et la désactivation de composants. Cette technologie constitue une base appropriée pour développer des solutions qui traitent la dynamicité dans la conception des applications coopératives (Guerrero et Fuller (2001)). Pour de telles applications, l'adaptabilité aux changements spécifiques à la coopération peut exiger l'activation et l'intercommunication dynamiques de différents composants logiciels, à différents moments d'exécution. Ce problème est traité dans cet article comme une fonction de coordination se concentrant sur la gestion de l'évolution dynamique des architectures logicielles. Cette fonction de coordination est basée sur une description utilisant un graphe et un ensemble de règles de transformation qui seront appliquées selon un protocole bien défini.

Dans les divers travaux passés utilisant les graphes pour décrire les architectures dynamiques tel que ceux de Dorfel et Hofmann (1998), les graphes sont employés pour décrire des modèles d'architectures matérielles utilisés pour générer des implémentations VHDL/C. La dynamicité est considérée seulement au niveau des connexions définissant les flux de communication entre les composants.

Contrairement à notre approche, il n'est pas possible de créer des composants dynamiquement. La technologie orientée composant des architectures logicielles a l'avantage de permettre ce niveau de dynamicité. Notre approche s'appuie sur cette technologie et permet d'augmenter l'adaptabilité des architectures des applications distribuées.

Des travaux antérieurs tels que ceux de Wermelinger et Fiadeiro (2002), et ceux de Ermel, Bardhol, et Padberg (2001) proposent également d'utiliser les graphes et les règles de transformation pour modéliser les problèmes de coordination. Cette approche est mise en application par Holzbacher, Perin, et Sudholt (1997) pour les systèmes de commande de chemins de fer, et par Le Metayer (1998) pour la spécification de styles d'architectures. Fahmy et Holt (2000) proposent d'employer la transformation de graphe pour traiter l'architecture logicielle décrite par un graphe où les nœuds représentent des composants et où les arcs représentent les relations d'interdépendance entre ces composants. Contrairement à notre approche, seulement deux transformations prédéfinies sont considérées. Elles laissent représenter différents niveaux de détails (ou d'abstractions) dans le graphe d'architecture en remplaçant un noeud par ses composants de constitution et l'opération inverse. Le travail de Li et Horgan (2000) traite également la transformation d'architecture en utilisant des méthodes formelles basées sur le formalisme des machines à états finies étendues et le langage SDL. Il est consacré à la simulation de modèles dans des environnements *workflow*.

Jager (2000) adopte la réécriture de graphes pour produire des outils de gestion pour l'exécution des modèles de processus en utilisant un formalisme spécifique appelé les réseaux dynamiques de tâches (*dynamic task nets*). Alancar et Lucena (1995) proposent une approche formelle pour gérer l'évolution des architectures logicielles. Leur travail s'applique à la gestion de configuration impliquant le choix et l'intégration de différentes versions de modules logiciels.

Dans la section 2, nous présentons l'approche **ACG** du point de vue description. Dans la section 3, nous présentons la technique fondamentale de transformation du point de vue du fonctionnement du système. Nous illustrons cette approche par l'exemple du modèle d'architecture pour l'édition partagée. Des perspectives de notre travail sont données en conclusion.

2 - LES GRAPHES D'ARCHITECTURE ET LES REGLES DE TRANSFORMATION

2.1 - La structure ACG

Les graphes abstraits de composants (ACG) est une structure marquée et générique que nous employons pour définir les graphes d'architecture (ACG

totalement instantié), et les graphes de règles (ACG partiellement instantié). La première structure décrit une architecture orientée composant comme un ensemble de composants associés aux nœuds du graphe et un ensemble d'arcs dénotant les relations d'interdépendance entre ces composants.

ACG: *P*(Nodes) * *P*(Edges)

Dans une structure ACG, les nœuds décrivent des composants logiciels et sont ainsi marqués par les champs suivants : la classe, le comportement, l'état du composant (actif ou inactif), les facettes de son comportement, sa localisation (la machine sur laquelle ils sont exécutés), et une liste supplémentaire de paramètres concernant le niveau applicatif. Nous distinguons deux catégories de nœuds : les nœuds de règles et les nœuds de graphes. La différence entre ces deux types de nœuds est que le premier est une abstraction (au niveau structurel ou au niveau fonctionnel) d'un type de composants et peut avoir ainsi des champs variables[1], alors que le second correspond à un composant instantié de l'architecture et ne peut donc avoir que des champs totalement instantiés.

Node: Class * State * Facet * Location * Parameters

Les arcs sont orientés et sont définis par les deux nœuds qu'ils relient. Ils constituent le deuxième dispositif élémentaire de la description d'architecture et peuvent modéliser un large panel de relations (e.g. relations de contrôle, de composition, de niveau applicatif).

Edge : Node * Node

2.2 - Structure des règles de transformation

Nous définissons une *règle de transformation* comme une partition d'un *graphe de règle* permettant de décrire les contraintes qui conditionnent l'évolution de l'architecture et les changements qui se produisent quand une règle est applicable. Au niveau supérieur de l'abstraction, une règle de transformation peut être vue comme un triplet $RT \equiv <Partition, constraints, Substitutions>$, où :

- ***Partition*** : est une décomposition du graphe de la règle de transformation en quatre zones :
  - La zone *Inv* : Un fragment du graphe de la règle qui devrait être identifié (par homomorphisme) dans le graphe d'architecture. Ce fragment du graphe restera inchangé après l'application de la règle.
  - La zone *Del* : Un fragment du graphe de la règle qui doit être identifié (par homomorphisme) dans le graphe de l'architecture. Le fragment du graphe qui lui a été associé par l'homomorphisme est supprimé après l'application de la règle.
  - La zone *Abs* : Un fragment du graphe de la règle qui ne doit pas être identifiable (par homomorphisme) dans le graphe de l'architecture pour que la règle de transformation soit applicable.
  - La zone *Add* : Le fragment du graphe de la règle qui sera ajouté après l'application de la règle.
- ***Constraints*** : Décrit les contraintes sur les champs des noeuds. La règle n'est applicable que si toutes ces contraintes sont satisfaites par les nœuds du graphe de l'architecture qui sont unifiés avec les nœuds de la règle. Une contrainte est un couple dont le premier champ est la fonction d'évaluation de la contrainte δ (une fonction prenant en paramètre un ensemble de nœuds et renvoyant un booléen), et le deuxième est l'ensemble des nœuds qui sera évalué par **δ**.

Const : *P*(δ (: *P*(Nodes) → boolean) * *P*(Nodes)))

- ***Substitutions***: Modélise les différentes substitutions que devrait subir les champs de certains nœuds du graphe après l'application de la règle. Les substitutions sont modélisées par la procédure de substitution σ qui prend en paramètres un ensemble de noeuds et permet de substituer à certains de leurs champs des nouvelles valeurs. Ceci permet de spécifier, outre des architectures évoluant au niveau de la configuration et impliquant les quatre opérations fondamentales (i.e. l'addition et la destruction de composants ou de connections), des évolutions dynamiques à l'échelle d'un composant (e.g. migration, changement de comportement) via des changements au niveau des attributs. Ainsi, par exemple, pour un composant d'édition partagée représentant une fenêtre d'édition, en liant son comportement à la valeur d'un attribut, on peut décrire l'évolution dynamique de ce comportement et ainsi pouvoir, par exemple, spécifier son passage du mode lecture vers un mode lecture/écriture et vice-versa. Cette spécification sera réalisée en décrivant quand est-ce que l'évolution dynamique est consistante avec, par exemple, la vérification de l'exclusion mutuelle sur la fenêtre d'édition, et comment cette évolution serait traduite au niveau implémentation avec, par exemple, le positionnement correcte du droit d'écriture et l'activation des boutons d'édition sur l'interface graphique.

Sub : *P*(σ (:*P*(Nodes) → void) * *P*(Nodes))

Ainsi avec les définitions précédentes, une règle possède la structure suivante :

Rule: $\underbrace{Inv * Del * Add * Abs}$ * Const * Sub

2.3 - Le protocole de coordination

Le protocole de coordination est en charge de gérer l'évolution dynamique de l'architecture du système. Ce protocole manipule, le graphe courant, les règles de coordination, et les événements à traiter, et

[1] Dans notre notation, les variables seront préfixées par le symbole "_". Par exemple, la notation <*E, _x,F,Ad1*> dénote un noeud de la classe *E* avec une facette appartenant à *F*, situé dans le site *Ad1*. La variable *_x* indique que le noeud est peut-être dans un état actif ou inactif.

associe à chaque type d'événements les règles de transformation correspondantes. Il associe, aussi, pour chaque type d'événements et pour chacune des règles lui correspondant, la procédure de transformation qui doit être appliquée à chaque règle (au niveau de son graphe, son champ *constraints*, et son champ *substitutions*) avant l'unification avec le graphe. Le protocole de coordination peut introduire aussi des événements spéciaux traduisant, par exemple, des vérifications de propriétés telles que des propriétés de *sûreté* ou de *complétude*. Ces propriétés sont décrites sous la forme d'une ou de plusieurs règles de coordination[2].

Protocol :

Graph * *P*(Rule) * *P*(EventType * Trans * *P*(Rule))

L'événement décrit l'action de déclenchement menant à l'application d'un ensemble de règles de transformation sur le graphe courant de l'architecture. Il peut être produit par le système lui-même ou par son environnement, et est décrit comme un couple contenant le type de l'événement, et les paramètres qu'il transporte.

Event: EventType * EventParameters

Le champ *Trans* permet de répercuter les paramètres des événements sur les règles de transformation. Les règles sont ainsi partiellement instantiées en affectant des valeurs à certaines de leurs variables.

Trans: P(α (: *P*(Nodes) * Event → void) * *P*(Nodes))

3 - APPLICATION DES REGLES DE TRANSFORMATION

Nous définissons la fonction d'unification comme une fonction récursive qui établit un homomorphisme entre la partition du graphe de la règle de transformation et le graphe de l'architecture (en tenant compte du champ *constraints* de la règle). Elle est basée sur les quatre définitions suivantes qui décrivent l'unification d'un ensemble de nœuds du graphe de règle avec un ensemble de nœuds du graphe de l'architecture.

Définition 1 [*Unification de champs de nœuds*]

Unifiable(champ$_i$, champ$_j$) ≡

$$\begin{cases} \exists \_X \in Variables / champ_i = \_X, OU \\ champ_i = champ_j \end{cases}$$

Définition 2 [*unification de deux nœuds*]

Soient N_1=(N_1.champ$_1$,…,N_1.champ$_n$) un nœud d'un graphe de règle, et N_2=(N_2.champ$_1$,…,N_2.champ$_m$) un nœud d'un graphe d'architecture. Alors,

Uni*fiable(N_1,N_2)* ≡

$$\begin{cases} (n = m), Et \\ \forall i \in [1,..,n], Unifiable(N_1.champ_i, N_2.champ_i), Et \\ Unifiable(N_1.successeurs, N_2.successeurs), Et \\ Const(r), Et \\ \text{Pas d'inconsistance dans les unifications} \end{cases}$$

Définition 3 [*Unification de deux ensembles ordonnés de nœuds*]

Soit EO_1 un ensemble ordonné de nœuds d'un graphe d'une règle tel que $EO_1=[N_{1,1},...,N_{1,n}]$. Soit EO_2 un ensemble ordonné de nœuds d'un graphe d'architecture tel que $EO_2=[N_{2,1},...,N_{2,m}]$. Alors,

S_Unifiable(EO_1,EO_2) ≡

$$\begin{cases} (n = m), Et \\ \forall i \in [1,..,n], Unifiable(N_{1,i}, N_{2,i}), Et \\ Const(r), Et \\ \text{Pas d'inconsistance dans les unifications} \end{cases}$$

Définition 4 [*Unification de deux ensembles de nœuds*]

Soient *NR* un ensemble de nœuds d'un graphe de règle tel que $NR=\{N_{1,1},...,N_{1,n}\}$ et *NG* un ensemble de nœuds de graphe d'architecture tel que $NG=\{N_{2,1},\ldots,N_{2,m}\}$. Alors,

Unifiable(NR,NG) ≡

$$\begin{cases} (n \le m), Et \\ \exists \{N_{2,i1},..., N_{2,in}\} \subset \{N_{2,1},...N_{2,n}\}, \\ S\_Unifiable\ ([N_{1,1},..., N_{1,n}], [N_{2,i1},..., N_{2,in}]) \end{cases}$$

Exemple

Soit le graphe de règle *G(r)* et le graphe **G** décrits dans Fig1, avec :

Fig. 1 – *G(r) le graphe de la règle r, et G le graphe de l'architecture courante.*

On a alors,

1- **S_Unifiable**([n1,n2,n3],[n4,n5,n6])?

⇔

Unifiable(n1,n4) ***Et*** **Unifiable**(n1.Succ,n4.Succ) ***Et*** **Unifiable**(n2,n5) ***Et*** **Unifiable**(n2.Succ,n5.Succ) ***Et*** **Unifiable**(n3,n6) ***Et*** **Unifiable**(n3.Succ,n6.Succ)

⇒

(_d=nom$_2$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** …

⇒

Inconsistance car *_d* est unifié avec deux valeurs différentes.

⇒ **Faux**.

[2] Un exemple est donné dans la section 4.2.2

2- **S_Unifiable**([n1,n2,n3],[n7,n6,n8])?
⇔
(_d=nom$_1$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** (14=14) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** **Unifiable**({n3},{n8}) ***Et*** **Unifiable**({n3},{n8}).
⇔
(_d=nom$_1$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** (14=14) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** **Unifiable**({ },{ }) ***Et*** (_d=nom$_1$)
⇔ **Vrai** avec comme résultat les unifications: *_d=nom$_1$ et _f=17.*

Définition 5 [*Unification d'une règle avec un graphe*]
Soient *r* une règle et *g* un graphe, soit *précondition(r)* l'ensemble des nœuds et des arcs appartenant à *(Inv(r)∪Del(r))* et soit *restriction(r)* l'ensemble des nœuds et des arcs appartenant à *(Inv(r)∪Del(r)∪Abs(r))*, alors,
Unifiable*(r,g)* ≡

$$\begin{cases} \exists s_g = \left(\{n_0,...,n_i\},\{e_0,...,e_j\}\right) \in g, Telque \\ S\_Unifiable\left(precondition(r), s_g\right), Et \\ Const(r), Et \\ \left(\forall s'_g = \left(\{n_0,...,n_i,...,n_{i+k}\},\{e_0,...,e_j,...,e_{j+l}\}\right) \subset g\right) \\ \Rightarrow \neg\left(S\_Unifiable(restriction(r), s'_g)\right) \end{cases}$$

L'unification d'une règle *r* avec un graphe *g* suivra la démarche suivante:
1. Si *r* n'est pas unifiable avec *g*, alors *g* reste inchangé.
2. Sinon :
   2.1. On rajoute dans le graphe *g* des copies des nœuds et des arcs contenus dans le champ *Add(r)*.
   2.2. On détruit les nœuds et les arcs du graphe qui ont été unifiés avec des nœuds et des arcs du champ *Del(r)*.
   2.3. On modifie les champs des nœuds de *g* qui ont été unifiés avec les nœuds figurant dans le champs *Sub(r)* en appliquant les procédures de substitution de la règle.

4 - UN CAS D'ETUDE, L'EDITEUR PARTAGE

Nous considérons ici, la gestion de l'architecture d'une application d'édition partagée. Il s'agit d'une application coopérative et distribuée permettant à plusieurs utilisateurs de travailler sur un même document. Ce document est organisé en composantes (pouvant représenter des caractères, des lignes, des paragraphes, des pages etc.) numérotées de *0* à *n*. Un utilisateur se doit, avant d'écrire dans une zone (une zone est un ensemble de composantes consécutives), de la réserver. Une fois qu'un utilisateur a fini sa rédaction, il peut libérer la zone qu'il a auparavant réservée. Chaque composante du document ne peut se trouver que dans l'un des deux cas de figures suivants : ou bien elle est libre, ou bien elle est réservée par un utilisateur unique. Pour éviter une surcharge inutile dans la description du cas d'étude, nous allons nous limiter à décrire les attributs des noeuds qui sont vraiment significatifs pour la coordination de l'application d'édition partagée.

4.1- Modélisation de l'architecture du cas d'étude

L'architecture de l'application d'édition partagée est modélisée par un graphe linéaire représentant les zones consécutives du document (e.g. un nœud *n1* est fils d'un nœud *n2 si et seulement si* la zone représentée par *n1* est consécutive à celle représentée par *n2*). Les noeuds du graphe modélisant ces zones devront donc décrire les paramètres de début et de fin de zone, l'état de la zone (*F* pour l'état libre et *B* pour l'état réservé), le propriétaire de la zone[3], et la localisation de la zone[4] (e.g. l'adresse IP de la machine où se trouve le propriétaire). Les nœuds du graphe auront donc pour champs cinq paramètres (un exemple est donné dans Fig. 2).

document où les composantes, de 0 à 10 sont réservées par Jacques, les composantes de 11 à 20 sont libres, de 21 à 35 sont réservées par Bernadette, et de 26 à 50 sont réservées par Claude.

4.2- Les règles de coordination

4.2.1- Les règles de réservation et de libération des composantes
On a classé les règles de transformation de l'application d'édition partagée en trois catégories selon les événements qu'elles traitent. La première catégorie concerne la réservation, la deuxième concerne la libération, et la troisième la vérification des deux propriétés de sûreté et de complétude. Par souci de simplification, et pour éviter le cas particulier d'un graphe avec un noeud unique[5], nous rajoutons deux noeuds (auxquels nous affectons des champs garantissant qu'ils ne seront jamais détruits, et qu'ils ne lèveront jamais une violation des règles de sûreté ou de complétude) l'un en tête et l'autre en queue du graphe à structure linéaire.

[3] Dans le cas d'une zone libre ce paramètre sera positionné à *nobody*

[4] Ce paramètre sera positionné à *nowhere* dans le cas d'une zone libre

[5] Le traitement de ce cas particulier doublerait inutilement le nombre de règles de réservation et de libération

Considérons un utilisateur "*owner*" sur la machine "*host*" et voulant réserver la zone [*begin,end*]. Pour que cela puisse être possible, il doit exister dans le document une zone libre [$begin_f$,end_f] telle que [*begin,end*] $\subseteq$ [$begin_f$,end_f]. Quatre cas de figure sont possibles:

1- Cas1 : ($begin_f$ = *begin*) ***Et*** (end_f = *end*),

2- Cas 2 : ($begin_f$ = *begin*) ***Et*** (end_f > *end*),

3- Cas 3 : ($begin_f$ < *begin*) ***Et*** (end_f = *end*),

4- Cas 4 : ($begin_f$ < *begin*) ***Et*** (end_f > *end*).

Considérons maintenant un utilisateur "*owner*" sur la machine "*host*" et possédant la zone *z*, lors de la libération de cette zone, quatre cas de figures sont possibles:

1- Cas 5 : La zone précédant *z* et celle qui la suit sont libres,

2- Cas 6 : La zone précédant *z* et celle qui la suit sont réservées,

3- Cas 7 : La zone précédant *z* est libre, et celle qui la suit est réservée,

4- Cas 8 : La zone précédant *z* est réservée et celle qui la suit est libre.

Fig. 3- *Les graphes des règles de réservation et de libération*

On obtient donc les quatre règles de réservation (*r1,r2,r3,r4*) et les quatre règles de libération (*r5,r6,r7,r8*) présentées dans Fig. 3. La réservation (respectivement la libération) d'une zone pour un utilisateur est donc possible *si et seulement si* l'une des quatre règles *r1, r2, r3,* ou *r4* (respectivement *r5, r6, r7,* ou *r8*) est applicable au graphe courant. Il est, aussi, à noter que la fonction d'unification telle qu'elle a été définie nous assure qu'un utilisateur ne peut libérer une zone que s'il en est le propriétaire.

Boolean **supAtt1**(*P(Nodes)* {N_1;N_2})
{ *return* (N_1.champ(1) > N_2.champ(1)); }

Boolean **supAtt0** (*P(Nodes)* {N_1;N_2})
{ *return*(N_1.champ(0) > N_2.champ(0)); }

Node : n1 = {_x; _y;B;_owner1;_host1},
n2 = {_begin;_end;F;nobody;nowhere},
n3 = {_begin;_end';F;nobody;nowhere},
n4 = {_begin';_end;F;nobody;nowhere},
n5 = {_begin';_end';F;nobody;nowhere},
n6 = {_a;_b;B;_owner2;_host2},
n7 = {_begin';_begin-1;F;nobody;nowhere},
n8 = {_begin;_end;B;_owner;_host},
n9 = {_end+1;end';F;nobody;nowhere};

Graph : Inv = ({n1;n6},{ }),
Del(r1) = ({n2},{(n1,n2);(n2,n6)}),
Del(r2) = ({n3},{(n1,n3);(n3,n6)}),
Del(r3) = ({n4},{(n1,n4);(n4,n6)}),
Del(r4) = ({n5},{(n1,n5);(n5,n6)}),
Add(r1) = ({n8},{(n1,n8);(n8,n6)}),
Add(r2)=({n8;n9},{(n1,n8);(n8,n9);(n9,n6)},
Add(r3)=({n7,n8},{(n1,n7);(n7,n8);
(n8,n6)}),
Add(r4)=({n7;n8,n9},{(n1,n7);(n7,n8);
(n8,n9);(n9,n6)});
'onst : C(r2)={(supAtt1,{n9;n8})},
C(r3)={(supAtt0,{n8;n7})},
C(r4)={(supAtt1,{n9;n8});
(supAtt0,{n8;n7})};
'ule : r1=(Inv,Del(r1),Add(r1),{ },C(r1)),
r2 = (Inv,Del(r2),Add(r2),{ },C(r2)),
r3=(Inv,Del(r3),Add(r3),{ },C(r3)),
r4 = (Inv,Del(r4),Add(r4),{ },C(r4)),
r5 = (Inv,Add(r1),Del(r1),{ }, { }),
r6 = (Inv ,Add(r2),Del(r2),{ }, { }),
r7 = (Inv ,Add(r3),Del(r3),{ }, { }),
r8 = (Inv ,Add(r4),Del(r4),{ }, { }),

.2.2 - Vérification des propriétés de sûreté et de omplétude.

es deux propriétés de sûreté[6] et de complétude[7] eront vérifiées par l'unification des deux règles les représentant (Fig. 4) déclarées comme suite :

Boolean **Intersect** (*P(Nodes)* {n1;n2})
{*return* ((n1.champ(0) ≥ n2.champ(0)) ***AND***
(n1.champ(0) ≤ n2.champ(1)))};}

Node: n10 = {_x; _y;F;nobody;nowhere},
n11={_a;_b;F;nobody;nowhere},
n12={_x;_y;_state1;_owner1;_host1},
n13={_a;_b;_state2;_owner2;_host2};
Graph: Inv(r9)=({n10;n11},{(n10,n11)}),
Inv(r10)=({n12;n13},{ }),

[6] La propriété de sûreté est violée ssi il existe deux zones consécutives libres

[7] La propriété de complétude est préservée ssi pour chaque paire de zones, leur intersection est vide.

```
        Empty=({ },{ });
Constr: C(r10)={(Intersect,{n12;n13})};
Rule:   r9=(Inv(r9),{ },{ },{ },{ }),
        r10=(Inv(r10),{ },{ },{ },C(r10));
```

```
        node1 = (0, N - 1, F, nobody, nowhere);
Graph : G = ({InitialNode; node1; FinalNode},
        {(InitialNode,node1);(node1,FinalNode)});
Protocol: EPCoo = (G,
                ("completness", {r10}, { })),
```

Fig.
et d

Une unification possible de l'une de ces deux règles indiquerait que la propriété correspondante est violée dans l'architecture courante.

3.3- Les événements traités par l'application

L'application d'édition Partagée doit traiter quatre types d'événements. Le premier concerne une demande d'un utilisateur de réserver une zone du document. Un événement de ce type doit donc transporter le début et la fin de la zone à réserver, le nom ou l'alias du demandeur, et la machine sur laquelle il se trouve. Le deuxième concerne une demande de libération. Un événement de ce type transportera exactement le même nombre et types de paramètres qu'un événement de réservation. Le troisième concerne une demande de vérification de la propriété de sûreté sur l'architecture courante, ce type d'événement ne transportera aucun paramètre et le quatrième concernera la propriété de complétude et ne transportera également aucun paramètre. On obtient donc la description suivante :

```
Event :
Book = ("book",{begin(:Int);end(:Int);
                owner(:String); host(:Int)}),
Free = ("free",{begin(:Int);end(:Int);
                owner(:String);host(:Int)}),
VerifySafety = ("safety",{ }),
VerifyCompletness = ("completness",{ });
```

3.4- Protocole de coordination

L'événement *Book* doit répercuter ses paramètres sur les nœuds des règles *r1,…,r4* et donc les nœuds *n1,…,n9*, l'événement *Free* sur les règles *r5,…,r8* et donc les nœuds *n1,…,n9*, l'événement *VerifySafety* sur la règle *r9* et donc les nœuds *n10* et *n11*, et l'événement *VerifyCompletness* sur la règle *r10* et donc les nœuds *n12* et *n13*. Au vu de ce qui a été développé précédemment, le protocole de coordination pour l'architecture de l'application d'édition partagée peut être décrit sous la forme suivante :

```
Node :  InitialNode = (T, T, B, nobody, nowhere),
        FinalNode = (⊥, ⊥ , B, nobody, nowhere),
```

Où,

```
Void  T ( Event e , P(Nodes) nodes)
    {for ( n in nodes)}
        {if (n.champ(0) ==_begin)
            {n.champ(0)=e.parameters(0);}
         if (n.champ(0) ==_end+1)
            {n.champ(0)=e.parameters(1)+1;}
         if (n.champ(1) ==_end)
            {n.champ(1)=e.parameters(1);}
         if (n.champ(1) ==_begin-1)
            {n.champ(1)=e.parameters(0)-1;}
         if (n.champ(3) ==_owner)
            {n.champ(3)=e.parameters(2);}
         if (n.champ(4) ==_host)
            {n.champ(4)=e.parameters(3);}}}
```

CONCLUSION

Nous avons présenté une approche orientée modèle pour la synthèse des protocoles de coordination pour les applications logicielles distribuées orientées composant. En nous basant sur des descriptions sous formes de graphes et de règles de transformation, nous avons pu introduire une approche pour décrire et gérer l'évolution dynamique d'une architecture logicielle de façon consistante. En comparaison avec celle présentée par Le Metayer (1998), notre approche permet une description plus fine et plus large puisqu'elle nous permet de modéliser des contraintes plus complexes pour l'évolution dynamique de l'architecture. Elle introduit des contraintes structurelles additionnelles modélisées par les deux champs *Inv* et *Abs*, et des contraintes fonctionnelles sur les champs de nœuds, permettant le changement de comportements et de propriétés du composant par l'intermédiaire du champ *Sub*stitutions. La vérification des propriétés via une modélisation par des règles de transformation est également possible.
Dans le cas particulier des logiciels de support des activités coopératives distribuées, cette coordination permet d'éviter les conflits entre les participants lors des accès simultanés aux mêmes objets de l'espace de travail partagé. Nous avons illustré et éprouvé notre approche pour les activités

d'édition en groupe avec des documents partagés à structure linéaire dynamique. La démarche est généralisable pour les structures arborescentes statiques et dynamiques. L'approche est en cours d'expérimentation avec les technologies EJB et CCM.
Nous travaillons actuellement sur la description des architectures dynamiques sous forme de styles d'architectures modélisés par les grammaires de graphes. Cette approche nous permettra de vérifier la conformité du protocole de coordination avec le style d'architecture spécifié.

BIBLIOGRAPHIE

Allen, R., Douence, R., Garlan, D. (1998), "Specifying and analyzing dynamic software architectures", in 1998 conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal.

Adler, R.M. (1995), "Distributed coordination models for client/server computing", in IEEE Computer, vol. 28, n° 4, pp14-22.

Alencar, P.S.C., Lucena, C.J.P. (1995), "A formal description of evolving software systems architectures", in Science of Computer Programming, vol. 24, n° 1, pp 41-61.

Dorfel, M., Hofmann R. (1998), "A prototyping system for high performance communication systems", in RSP'98, Leuven, Belgium.

Ermel, C., Bardhol, R., Padberg, J. (2001), "Visual design of software architecture and evolution based on graph transformation", in Uniform Approaches to graphical process specification techniques, Genove, Italy.

Fahmy, H., Holt, R.C. (2000), "Software architecture transformations", in ICSM, pp 88-96.

Guerrero, L.A., Fuller, D.A. (2001), "A pattern system for the development of collaborative applications", in Information and Software technology, vol. 43, n° 7, pp 457-467.

Holzbacher, A., Perin, M., Suldholt, M. (1997), "Modelling railway control systems using graph grammars: a case study", in the second international conference on Coordination Languages and Models, Berlin, Germany, pp 172-186.

Jager, D. (2000), "Generating tools from graph-based specifications", in Information and Software Technology, vol. 42, n° 2, pp 129-139.

Li, J.J., Horgan, J.R. (2000), "Applying formal description techniques to software architectural design", in Computer Communications, vol. 23, n° 12, pp 1169-1178.

Le Metayer, D. (1998), " Describing software architecture styles using graph grammars", in IEEE Transactions On Software Engineering, vol. 24, n° 7, pp 521-533.

Riveil, M., Senart, A. (2002), "Aspects dynamiques des langages de description d'architecture logicielle", in L'objet: Coopération dans les systèmes a objets", vol. 8, n° 3, pp 109-129.

Wegner, P. (1993), "Tradeoffs between reasoning and modelling", MIT Press.

Wermelinger, M., Fiadeiro, J.L. (2002), "A graph transformation approach to software architecture reconfiguration", in Science Of Computer Programming, vol. 44, pp 133-155.

QUALITE DE SEVICE WI-FI : ALGORITHME POUR LE SUPPORT DE DIFFSERV

Ilyes Gouta,
Ingénieur - Chercheur, ENSI
ilyes.gouta@ensi.rnu.tn

Pr. Abdelfettah Belghith,
Professeur, ENSI
abdelfattah.belghith@ensi.rnu.tn

Dr. Jean-Marie Bonnin,
Maître de conférences, ENST - Bretagne
jm.bonnin@enst-bretagne.fr

Adresse professionnelle,
Ecole Nationale des Sciences de l'Informatique, campus de Manouba,
Tunis, Tunisie

Résumé - Ce papier présente un algorithme destiné à offrir le support des classes de services DiffServ au-dessus de 802.11. L'objectif est de différentier les classes de service dans leur capacité à obtenir le support et ainsi d'offrir une certaine gestion de la QoS sur ce genre de médium. Ce papier présente le fonctionnement de l'architecture à différentiation de services (DiffServ) ainsi que la norme 802.11 de l'IEEE. Une étude de performances à l'aide d'une simulation sous NS met ensuite en relief les différences, en termes de débit et de gigue, entre le 802.11b original et le 802.11b modifié. Quelques perspectives et applications sont présentées en guise de conclusion.

Mots-clés: QoS, IEEE 802.11, DiffServ, CSMA/CA

QUALITE DE SEVICE WI-FI : ALGORITHME POUR LE SUPPORT DE DIFFSERV

1. INTRODUCTION

Le besoin en “informatique mobile” s’est fait sentir depuis le milieu des années 90. En effet, il est devenu indispensable de pouvoir accéder depuis n’importe quel emplacement aux données de son établissement/entreprise afin de répondre efficacement et rapidement aux exigences de l’entreprise mais aussi du particulier. Avant l’avènement des réseaux sans-fil, il fallait câbler d’une façon permanente l’endroit en question ce qui risquait de limiter/conditionner l’accès à ces données. Avec les technologies sans-fil, il est maintenant possible d’avoir un lien plus ou moins permanent avec son environnement qu’il soit de travail ou de loisir. L’introduction des applications telles que la visioconférence, le contrôle à distance donne à la technologie sans-fil une nouvelle dimension mais aussi de nouvelles contraintes en termes de délais de livraison et de bande passante allouée. En 1999, l’IEEE a normalisé la technique d’accès sans-fil 802.11b, largement utilisée de nos jours, offrant ainsi un débit théorique de 11 Mbps et assurant une couverture de l’ordre d’une centaine de mètres aux équipements qui intègrent ce standard. Techniquement 802.11b emploie une variante de CSMA, appelée CSMA/CA, comme protocole de niveau lien pour réguler les échanges des différentes entités communicantes. Le CSMA/CA est aussi connu pour ses limitations en termes de garantie de service.

Ce papier présente un algorithme permettant d’améliorer quelques aspects de son fonctionnement tout en offrant un support de la technique de différentiation de service (DiffServ). Une comparaison avec les performances du 802.11b original est présentée pour mettre en évidence les apports en termes de différentiation des différentes classes de trafic que ce soit pour le débit ou pour la variation d’interarrivée.

2. LA TECHNIQUE DIFFSERV

Définie par l’IETF, DiffServ [1] (pour Differenciated Services) est une technique destinée à intégrer les éléments de base de la gestion de la QoS pour les réseaux IP. DiffServ agit au niveau des agrégats de trafic en segmentant le trafic total en plusieurs classes (classes de services) dont le traitement est ensuite différentié dans les équipements d’interconnexion. L’allocation des ressources se fait par classe et non plus par application. Suivant la classe à laquelle un paquet appartient, un traitement plus ou moins privilégié lui est appliqué (ceci inclut la priorité d’acheminement, l’élimination sélective des paquets en cas de congestion). Cette solution est beaucoup moins lourde que l’approche à intégration de service (IntServ) qui suppose l’établissement explicite d’une réservation pour chaque flux de données. DiffServ définit 3 types de classes de services, l’administrateur d’un réseau ayant la liberté de n’implémenter que les classes qu’il juge nécessaires. Les différents types de classes sont les suivants :

EF (Expedited Forwarding) : c’est la classe d’excellence. Les paquets marqués EF doivent être acheminés avec un délai, une gigue et un taux de perte minimum. Des moyens techniques (contrôle d’accès, sur réservation,…) doivent être mis en oeuvre pour assurer le bon fonctionnement de celle-ci.

AF (Assured Forwarding) : quatre classes AF ont été défini, chacune d’elles comporte 3 sous-classes. Les paquets sont marqués AFxy tel que x dans l’intervalle [1,4] est le numéro de la classe AF et y dans [1,3] la précédence à l’écartement. Les paquets d’une même classe empruntent toujours la même file d’attente pour éviter le déséquencement. La précédence à l’écartement définit la priorité relative de rejet (ou un traitement spécifique) des paquets en cas de congestion.

BE (Best Effort) : c’est l’équivalent de l’Internet actuel où aucun traitement particulier ne vient améliorer le relayage des paquets appartement à cette classe.

Au total nous dénombrons 14 comportements possibles. Ils sont notés PHBs (Per Hop Behaviour). Un PHB est la manière avec laquelle un routeur traitera les paquets entrants (c’est à dire la mise en file d’attente plus le traitement en cas de congestion). Le PHB est déterminé à partir de DSPCs codés directement dans le champ TOS du paquet IP.

La norme recommande l’utilisation d’un ensemble précis de DSCPs pour marquer les

paquets IP. Le PHB par défaut (BE) est défini pour les paquets non marqués.

	Précédence 1	Précédence 2	Précédence 3
EF	46		
AF4	34	36	38
AF3	26	28	30
AF2	18	20	22
AF1	10	12	14
BE	0		

La norme recommande l'utilisation d'un ensemble précis de DSCPs pour marquer les paquets IP. Le PHB par défaut (BE) est défini pour les paquets non marqués.

3. LA NORME IEEE 802.11

L' IEEE (Institute of Electrical and Electronics Engineers) a normalisé plusieurs catégories de réseaux locaux : une variante de Ethernet (802.3), le Token Bus (802.4) et le Token Ring (802.5). En 1990 le projet d'un réseau local sans fil, nommé 802.11 [2], est lancé. Il a pour but de fournir une liaison sans fil à un ensemble d'utilisateurs fixes ou mobiles.

La norme 802.11 s'adresse essentiellement aux niveaux lien et physique du modèle OSI. En fait, elle introduit des modifications sur la couche basse du niveau lien (donc niveau MAC) et sur le niveau physique avec le support de plusieurs méthodes d'accès radio (donc la définition de plusieurs couches physiques). Il est à noter que la nouvelle couche MAC est commune à toutes les couches physiques.

3.1 La couche physique

L'IEEE a initialement défini trois couches physiques initiales :

FHSS : pour Frequency Hopping Spread Spectrum, c'est une technique d'accès radio qui consiste en l'émission des symboles sur une bande de fréquence de largeur fixe et de porteuse variable dans le temps. L'émetteur doit se mettre en accord avec le récepteur sur la séquence de porteuses à utiliser. En mode infrastructure, c'est le point d'accès qui annonce les fréquences à utiliser pendant les émissions.

DSSS : pour Direct Sequence Spread Spectrum, le fonctionnement est similaire à FHSS sauf qu'il n'y a pas de changement de porteuse dans le temps, d'où un plus large spectre d'émission et donc un meilleur débit.

IR : pour Infrared, les infra rouges sont utilisés pour le transfert des données. Cette méthode impose que les distances entre émetteurs/récepteurs soient limitées. Elle offre un débit de 1 Mbps.

L'amendement 802.11b [3], qui définit une quatrième couche physique, utilise la méthode DSSS, couplé avec un encodage utilisant la modulation de phase améliorée, pour la transmission des données. 802.11b permet d'atteindre un débit théorique de 11 Mbps. 802.11a, la cinquième couche, fait encore mieux en autorisant des débits jusqu'à 54 Mbps (codage OFDM). Toutefois, 802.11a utilise la gamme des 5 GHz et, est de facto, considérée comme étant un cas particulier de la norme. Enfin, l'IEEE vient de standardiser une nouvelle couche physique, nommée 802.11g, qui utilise les techniques de 802.11a pour obtenir le même débit dans la bande ISM (2,4 GHz).

3.2 La couche Liaison

Dans 802.11, la couche liaison est divisée en deux sous-couches : LLC (Logical Link Control) et MAC (Medium Access Control). La couche LLC a les mêmes propriétés que celles définies pour la couche LLC 802.2. Il est donc concevable de relier un réseau sans fil (WLAN) à n'importe quel réseau qui adopte cette couche pour gérer ses accès. La couche MAC est responsable de la procédure d'allocation du support (accès), de l'adressage des paquets, du formatage des trames, du contrôle d'erreur (via un CRC, Cyclic Redundancy Check), ainsi que de la fragmentation et du réassemblage.

3.3 Accès au support

L'accès au support est déterminé par une fonction dite fonction de coordination : c'est une fonction logique qui détermine l'instant d'émission/réception d'une station associée à un Basic Service Set (BSS). Le standard définit deux méthodes d'accès au support : DCF (Distributed Coordination Function) et PCF (Point Coordination Function). La première utilise la technique CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) pour arbitrer l'accès au support. Elle est conçue de façon à ce que tous les utilisateurs aient une chance égale d'accéder au médium. La deuxième technique est basée sur l'interrogation régulière, par le point d'accès, de l'ensemble des stations pour leur demander s'ils ont des données à transmettre. Chaque station, pour qu'elle soit interrogée, doit s'enregistrer et réserver préalablement un temps d'émission auprès du point d'accès. Actuellement, la plupart du matériel Wi-Fi disponible sur le marché n'implémente que la méthode d'accès DCF, que ce soit pour les points d'accès ou les stations clientes.

3.4 La fonction de coordination DCF

Dans un environnement radio, une station qui émet n'a pas la possibilité de vérifier, en parallèle,

l'intégrité des données qu'elle émet (s'il y a eu collision ou pas car en radio il est très difficile/coûteux d'émettre et de recevoir au même instant), d'où la nécessité de tenir compte de cette limitation dans la conception du contrôle d'accès au medium. DCF met en oeuvre la technique CSMA/CA pour résoudre ce problème, en instaurant une démarche particulière à exécuter avant et pendant l'émission d'une trame. De plus comme il n'y a pas de détection de collision, CSMA/CA utilise des trames ACK pour valider l'envoi des données. La norme définit 3 temps inter-trames destinés à harmoniser le dialogue entre les machines et à diminuer les collisions. Ce sont des temps qui espacent les transmissions inter et intra dialogue(s). Ces temps sont :

SIFS : pour Short Inter-Frame Spacing, qui vaut 28 ms. Ce temps est utilisé pour espacer les émissions d'un même dialogue. (machine A à machine B).

PIFS (PCF IFS) : pour Point Inter-Frame Spacing, égal à un SIFS + 78 ms. C'est un temps utilisé par le point d'accès pour assurer l'acquisition du support. (accès prioritaire)

DIFS (DCF IFS) : pour Distributed Inter-Frame Spacing, temps utilisé par une station pour commencer une nouvelle transmission. Il vaut un PIFS + 128 ms.

Chaque station possède un temporisateur, appelé NAV (Network Allocation Vector), qui sert à retarder le temps d'émission d'une éventuelle trame. Ce temporisateur est mis à jour dès le début de l'émission d'une trame par une autre station. NAV est alors incrémenté par la valeur du deuxième champ « Durée » situé dans l'entête MAC. La station procède alors à sa décrémentation. Une fois le champ NAV à zéro, cette dernière continue à attendre un DIFS. Si après cette période le support est libre (et si elle a des données à émettre), elle envoie ses données sinon elle continue à écouter le support jusqu'à qu'il soit libre. Elle retransmettra ses trames après un temps de reprise (back off) déterminé par l'algorithme BEB (Binary Exponential Back off).

Ce temps de reprise est déterminé par la quantité [22+i*rand()]*timeslot, avec i le nombre de tentatives consécutives d'émission, rand() un nombre aléatoire uniforme compris entre 0 et 1. Ce temps est toujours compris entre les deux valeurs CWmin et CWmax spécifiés par le point d'accès. Si une trame, envoyée par une station A, est reçue correctement par B, B doit répondre par une trame d'acquittement ACK après un SIFS pour indiquer à A le succès de la transmission. Si un ACK n'est pas reçu, la station A reprend la transmission. Les performances de cet algorithme peuvent être améliorées en utilisant un mécanisme optionnel basé sur la réservation de temps d'accès. Ce mécanisme s'apparente de la méthode RTS/CTS utilisée depuis longtemps dans des équipements comme les modems et les cartes réseaux. L'idée de base est d'envoyer une trame RTS (Request To Send) à la station destination avant l'émission d'une (ou plusieurs) trame. La durée totale d'émission est indiquée dans la trame RTS. Ceci inclut, dans notre cas (802.11b), le temps d'émission des données, des SIFS et des ACKs. La machine destination répond par un CTS (Clear To Send). En écoutant cette signalisation, les autres stations mettent à jour leur temporisateur NAV. Cette technique permet d'éviter une éventuelle collision lors d'un transfert de données de longues durées.

L'algorithme CSMA/CA est déroulé par l'ensemble des stations d'un BSS, ce qui permet de garantir une chance égale aux différentes stations pour accéder au support. En tant que tel, il n'offre pas de possibilités pour la gestion de la qualité de service.

4. SUPPORT DE DIFFSERV

Plusieurs mécanismes, tels que [4], [5] et [6] ont été proposés pour introduire la différentiation de service dans 802.11b. En effet, [5] et [6] proposent de jouer sur les temps d'inter-frame (les IFS) pour différentier les classes de service : une station avec un trafic prioritaire utilise des IFSs plus courts qu'une station ayant un trafic de basse priorité. Utiliser des IFSs plus courts revient à réduire la durée d'attente et par là même augmenter la probabilité d'accès au support. Il est à noter que toutes ces techniques présentent des compromis du genre ratio de bande passante utilisée, degrés de différentiation entre les classes de priorités hautes et faibles qu'il faut prendre en compte.

Pratiquement toutes ces approches proposent des solutions de niveau lien où il faut modifier la couche MAC pour les intégrer, ce qui rend assez difficile leur faisabilité dans le cadre réel et opérationnel actuel.

Dans des conditions idéales (toutes les stations sont en vue directe, distance raisonnable entre le point d'accès et les stations, pas d'obstacles, etc.), un réseau 802.11b offre un débit efficace de l'ordre de 5 Mbps. La bande passante restante est utilisée par les signaux de contrôle de CSMA/CA. Pire encore, sachant que les performances de CSMA/CA dépendent aussi de la charge imposée [8] par les

clients du BSS, il arrive dans la pratique que le canal soit saturé (CSMA/CA passe son temps à "backoffer") et qu'aucune transmission ne soit possible : il faut bien éliminer de l'information pour éviter ce genre de scénario.

L'idée mise en œuvre ici est d'attribuer des probabilités d'émission à chaque classe de trafic présente dans le BSS pendant une période de longueur *T* secondes. Ces probabilités, appelées ***Pe[k]*** k allant de 1 à 14, seront utilisées par les différentes stations pour décider d'émettre ou de détruire (voir de mémoriser temporairement) un paquet de données lorsqu'il est encore au niveau de la couche réseau, c'est à dire, avant que le système ne le passe à l'interface réseau en question pour l'émettre physiquement). Ces probabilités seront construites par une entité centrale (le point d'accès par exemple) sur la base d'informations envoyées par les stations du BSS pour chaque période *T*. Ces informations décrivent la charge par classe de service (un vecteur comportant 14 entrées) en termes de bande passante. Une fois déterminé, le vecteur ***Pe*** sera transmis aux stations clientes pour être utilisé pendant la prochaine période. On jouera sur ces probabilités pour différencier la probabilité d'accès à la ressource pour les différentes classes de service. A titre d'exemple, nous affecterons à la classe EF la probabilité d'émission 1 et 0.5 à BE. Ainsi, un paquet BE sur deux sera transmit soit un gain de la moitié de la bande passante attribuée à la classe BE qui sera réaffectée à la classe EF. Voici un exemple d'algorithme pour le calcul du vecteur ***Pe*** :

Algorithme *: Allocation des Pe[k]*

bp_eff=802.11(bp efficace)

pour *i* ***de*** *EF* ***à*** *BE* ***faire***
s[i] = bande passnate[i] demandé
si *(s[i] != 0)* ***alors***
si *(bp_eff > s[i])* ***alors*** *bande_allouée = s[i]*
sinon *bande_allouée = bande_efficace*
bp_eff = bp_eff - bande_allouée
Pe[k] = bande_allouée / s[i]
fin si
si *(Pe[k] < 1)* ***alors break***
fin pour

remise-en-échelle(Pe[k])

L'utilisation de la bande passante se fera en fonction du vecteur ainsi calculé et transmis aux clients du BSS. Cette allocation se fait par ordre de priorité, c'est à dire, en traitant les besoins de la classe EF, puis AFxy et enfin BE. Dès qu'il n'y a plus de bande passante disponible, on arrête le processus. Sous 802.11b, il s'agit donc de distribuer une bande passante efficace assez limitée (de l'ordre de 6 Mbps [10]) aux différentes classes de service par ordre de priorité. Il faut garder en esprit que la composante essentielle de notre algorithme est la fonction de calcul des ***Pe***. Elle utilise les structures de données décrivant le trafic présent dans le BSS et calcule efficacement le vecteur des probabilités d'émission. Dans notre cas, nous avons basé les calculs uniquement sur la bande passante requise pour chaque classe de service (sans plus de détails tels que la moyenne de la taille des paquets, la moyenne de l'interarrivée). C'est la fonction donnée en exemple qui sera utilisée pour la simulation de l'algorithme. La dernière étape consiste à appliquer une remise en échelle des différents ***Pe***. Il est ainsi possible d'affiner d'avantage ces probabilités afin de mieux différencier les classes de service et même d'offrir à un opérateur externe la possibilité de les modifier directement.

5. SIMULATION SOUS NS

Le but de la simulation est de comparer les performances en termes de répartition de la ressource radio d'un BSS entre les différents flux. L'algorithme proposé est ainsi comparé avec la solution classique (c'est à dire CSMA/CA pur). Le BSS en question est constitué de 9 stations dont une (la station 0) fait office d'un point d'accès. Quatre trafics CBR (la durée qui sépare deux émissions est constante) sont simulés :

2 vers 7	1er trafic marqué EF à 128 Kbps.
4 vers 8	2eme trafic marqué AF42 à 1.5 Mbps
1 vers 6	3emetrafic marqué AF22 à 2 Mbps
3 vers 5	4eme trafic marqué BE à 5 Mbps

Les différentes stations sont disposées sur un cercle de rayon 35 mètres ayant pour centre le point d'accès. La couche MAC mise en oeuvre est basée sur un modèle simulant le standard 802.11 (ns/mac/802_11.cc [9]). Une couche de niveau liaison est introduite (ns/mac/ll.cc) pour simuler la latence d'accès au support entre 25 µs et 50 µs.

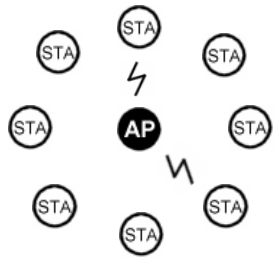


Fig 1 : disposition des nœuds pour la simulation

Finalement, le mécanisme de protection RTS/CTS a été désactivé vu que tous les paquets échangés ont une taille inférieure au seuil au delà duquel le mécanisme est déclenché (RTS_Threshold). Des simulations ont été faites pour des paquets de 192, 256, 384, 512, 768, 1024, 1512 et 2048 octets (les performances de CSMA/CA dépendent de la taille des paquets échangés, de la densité de la BSS en terme du nombre des stations actives, etc.). Chaque simulation dure 120 secondes. Chaque trafic possède une date de début et une date de terminaison. Ces dates sont spécifiées dans le tableau suivant :

Début	Fin	Priorité	Type
0	90	EF	Audio
0	90	AF42	Video
15	120	BE	CBR
40	105	AF22	CBR

Les courbes qui suivent ont été synthétisées à partir des traces recueillies après chaque simulation. Elles représentent la bande passante utilisée par chaque type de trafic (en CSMA/CA pur et modifié) en fonction du temps ainsi que l'interarrivée pour la classe EF, pour des paquets de 512 octets. Finalement, une dernière figure présente le taux d'utilisation (efficacité) du support en fonction de la taille des paquets.

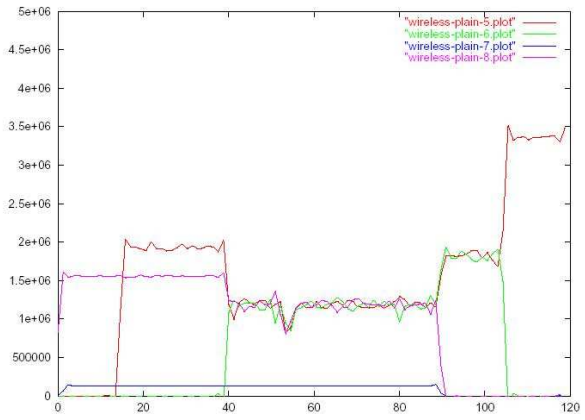


Fig 2 : bande passante utilisée par classe de trafic, 802.11b original, bits/s = f(t)

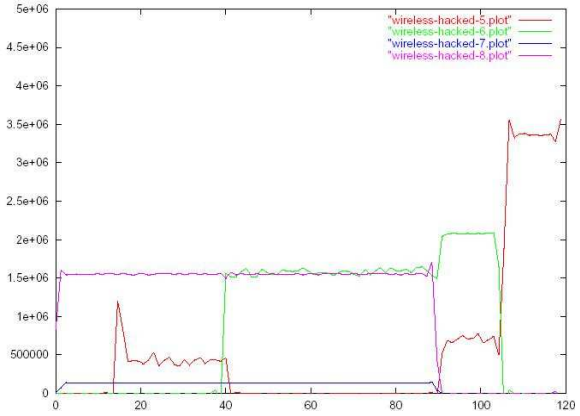


Fig 3 : bande passante utilisée par trafic, 802.11b modifié, bits/s = f(t)

Nous constatons un gain en bande passante significatif pour les classes de service AF22 et AF42. Nous arrivons même à satisfaire, pendant toute sa durée d'existence, le besoin en bande passante de ce dernier au détriment de la classe Best Effort. Un gain relativement important est décelable aussi au niveau de l'interarrivée. En effet, la simulation montre qu'elle est plus "bornée" pour les stations qui implémentent l'algorithme que pour celles qui utilisent le 802.11b original. Ce qui se traduit par une gigue plus faible.

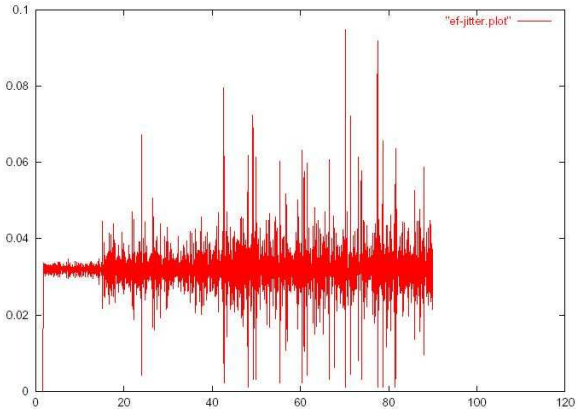


Fig 4 : interarrivée des paquets de la classe EF, 802.11 original, sec = f(t)

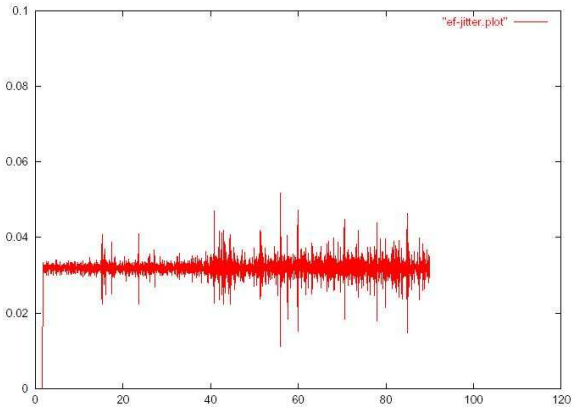


Fig 5 : interarrivée des paquets de la classe EF, 802.11 modifié, sec = f(t)

Cette amélioration est due à la libération du support au profit de la classe EF : en attribuant des probabilités inférieures à 1 pour les classes AF22 et BE, nous empêchons préventivement l'émission de paquets non prioritaires ce qui permet de libérer le support au profit de la classe EF dont la probabilité d'émission vaut 1. Cette amélioration est fonction de la taille des paquets échangés. En effet, en augmentant cette taille, il est possible d'améliorer significativement la qualité du trafic en limitant sensiblement la variation de l'interarrivée.

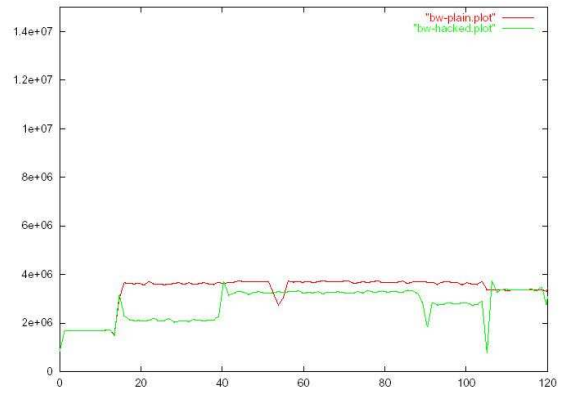


Fig 6 : bande passante totale utilisée par le 802.11 original/modifié, bits/s = f(t)

Cette courbe représente l'utilisation totale de la bande passante par les deux versions du protocole. Nous constatons que ce qui est utilisé par notre algorithme est inférieur à ce qui est normalement offert par 802.11b. Nous avons donc des gains en différentiation par classe de trafic et une perte, de l'ordre de 700 Kbps, sur l'ensemble de la bande passante ; c'est le prix de la différentiation introduite par notre algorithme. Ce prix dépend de la taille des paquets échangés. En faisant varier ce paramètre lors de nos simulations, nous avons obtenu les courbes suivantes. La première exprime le ratio de la bande passante inutilisée en fonction de la taille des paquets (l'activité sur le support varie inversement à la taille des paquets). La seconde courbe traduit la différence entre la moyenne des temps d'arrivées et la moyenne des temps d'inter-émission (trafics CBR -> temps d'inter-émission constant). L'intersection des deux courbes représente le meilleur compromis, en termes de taille de paquet, entre la perte de bande passante introduite par notre algorithme et la déviation par rapport à la valeur d'interarrivée idéale.

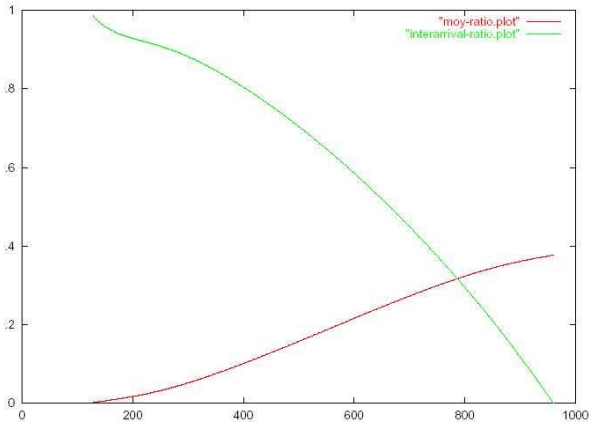


Figure 7 : ratio de bande passante gaspillée en fonction de la taille des paquets échangés / déviation entre interarrivée simulée et interarrivée théorique

Dans notre configuration, la taille des paquets ne doit pas dépasser 800 octets sous peine de perdre une partie importante de la bande passante efficace. De même des paquets de taille inférieure engendrent une variation d'interarrivée, certes moins importante qu'avec du 802.11b pur, mais qui reste toutefois assez importante. La perte en bande passante efficace dépend de l'algorithme de calcul des ***Pe*** ainsi que du nombre des paquets échangés sur une période de temps (i.e l'activité engendrée sur le support). Nous constatons une perte significative pour des paquets de 1024 octets. Ceci est dû à des rejets excessifs de paquets et donc de blocks de 1024 octets (mauvais ***Pe***).

Il faut noter que l'activité sur le support est plus importante avec des paquets de petites tailles qu'avec des paquets de tailles plus importantes (fréquence d'accès au médium plus élevée) étant donné un débit total constant. Les performances de notre algorithme sont affectées par cette activité. En effet, avec des paquets assez petits (inférieur à 192 octets), l'algorithme n'améliore que très légèrement la différentiation. Il sera donc nécessaire de prendre en compte la taille des paquets échangés dans le calcul des ***Pe***.

6. CONCLUSION ET PERSPECTIVES

Dans ce papier, nous avons présenté un mécanisme basé sur le principe d'un rejet sélectif permettant d'assurer une certaine différentiation entre les classes de service définies par DiffServ. Les performances du mécanisme observées lors de simulations sous NS sont discutées. Quelques points tels que l'efficacité de la génération des ***Pe*** et la minimisation de la perte de la bande passante sont à améliorer. Une deuxième alternative, pour le rejet définitif des paquets, basée sur la bufférisation temporaire et la réémission avec un débit adapté est en cours d'étude. Cette bufférisation servira surtout à améliorer et à protéger les performances de TCP.

Cet algorithme, même s'il est assez simple, permet entre autre de prévenir la pénurie en bande passante lorsqu'un nombre important d'utilisateurs se connectent à un BSS donné. En effet, une fois la charge limite (0.8) dépassée, CSMA/CA passera le plus clair de son temps à exécuter des backoff ce qui se traduit, pour les utilisateurs, par des taux de service s'approchant de 0. Notre algorithme permet d'affecter à chaque utilisateur une portion de la bande passante efficace tout en respectant les classes de services utilisées par le BSS. De fait il a une double fonction : éviter l'effondrement des performances de CSMA/CA et protéger les classes de service même dans des conditions de charge extrêmes.

Enfin, une modélisation formelle, basée sur l'augmentation du modèle proposé en [7] ainsi qu'une implémentation sous la forme d'un patch pour le driver HostAP [11], tournant sous Linux, sont en cours d'étude.

7. RÉFÉRENCES

[1] IETF, An expedited Forwarding PHB (Per Hop Behaviour) RFC 3246, Assured Forwarding Group PHB RFC 3260, Per Hop Behaviour Identification Codes RFC 3140. http ://www.ietf.org

[2] *ANSI/IEEE Std 802.11 Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[3] *ANSI/IEEE Std 802.11b Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications : Higher-Speed Physical Layer Extension in the 2.4 GHz Band*, 1999-2001.

[4] A. Veres, M. Barry, L. H. Sun, A. Campbell, "*Supporting Service Differentiation in Wireless Packet Networks using Distributed Control*", 2001.

[5] W. T. Chen, Y. M. Lin, S. C. Lo, "*Priority-Based Contention Control in IEEE 802.11 Wireless LANs*", Departement of Computer Science, National Tsing Hua University, Hsin-Chu, Taiwan 30043, R.O.C.

[6] Dr. J. Deng, R. S. Chang, "*A Priority scheme for IEEE 802.11 DCF Access Method*", IEICE Trans. Commun., Vol. E82-B, No.1, January 1999.

[7] G. Bianchi, "*Performance analysis of the IEEE 802.11 Distributed Coordination Function*", IEEE Journal on Selected Areas in Communications", Vol. 18 Issue : 3, March 2000.

[8] M. Natkaniec and A.R. Pach, "*An Analysis of the Backoff Mechanism used in IEEE 802.11 networks*", Proc. the fifth IEEE Symposium on Computers and Communications, 2000.

[9] NS, Network simulator v2.26, www.isi.edu/nsnam/ns/

[10] J. Jun, P. Peddabachagari, M. Sichitiu, *"Theoretical Maximum Throughput of 802.11 and its applications"*, Second IEEE International Symposium on Network Computing and Applications, April 16 - 18, 2003 Cambridge, Massachusette.

[11] HostAP, http://www.hostap.org, open source driver for Prism2 based WiFi cards for Linux.

APPROCHE FORMELLE INTEGREE POUR LA SPECIFICATION DES ARCHITECTURES DYNAMIQUES ORIENTEES COMPOSANTS

Imen Loulou
Etudiante doctorante, Faculté des Sciences Economiques et de Gestion de Sfax, Tunisie
Imen.loulou@tunet.tn

Ahmed Hadj Kacem
Maître assistant à la Faculté des Sciences Economiques et de Gestion de Sfax, Tunisie
Ahmed@fsegs.rnu.tn

Mohamed Jmaiel
Maître de conférences à l'Ecole Nationale d'Ingénieurs de Sfax
Mohamed.jmaiel@enis.rnu.tn

Khalil Drira
Chargé de recherches au LAAS-CNRS
Khalil@laas.fr

Adresses professionnelles
Université de Sfax, LARIS, B.P. 1088, 3018 Sfax, Tunisie
LAAS-CNRS, 7 avenue de Colonel Roche, 31007 Toulouse Cedex 4, France

Résumé : Nous proposons dans cet article une approche formelle intégrée pour la spécification des architectures dynamiques orientées composants. Nous procédons par intégration d'une approche fonctionnelle et d'une approche structurelle basée sur les grammaires de graphes. Nous visons par là, la simplification de la spécification, l'amélioration de la lisibilité et de la compréhension et la prise en charge de la dynamique. En effet, nous utilisons le langage formel *Z* pour décrire le style architectural qui doit être préservé durant l'évolution d'une architecture, d'une part. D'autre part, nous décrivons la dynamique via des règles de transformation de graphes gardées dont le corps décrit les contraintes structurelles et dont les gardes décrivent principalement les contraintes fonctionnelles du système. Nous exprimons la sémantique avec la notation Z également, obtenant ainsi une approche unifiée qui prend en charge l'aspect statique et l'aspect dynamique. Nous utilisons pour cela l'outil d'analyse Z-EVES pour valider nos spécifications *Z*.

Summary : This paper proposes an integrated formal approach for the specification of dynamic component-based architectures. So, we integrate a functional approach and a structural approach based on graph grammars. The purpose is to express dynamism while offering a simple specification easy to read and to understand. On the one hand, we use the specification language *Z* to describe the constraints made on the architectural style. These constraints have to be maintained during the system evolution. On the other hand,

we describe the dynamic evolving of architecture via guarded graph-rewriting rules whose body describe the structural constraints and whose guards mainly describe the functional constraints of the system. We express the rules entirely with the Z notation also, obtaining in this way, a unified approach which treats the static as well as the dynamic aspect. To validate our specifications, we use Z-EVES an advanced analysis tool supporting the Z-specification language.

Mots clés : architecture logicielle, style architectural, architecture dynamique, réécriture de graphes, applications orientées composants, spécification formelle.

Approche formelle intégrée pour la spécification des architectures dynamiques orientées composants

1 INTRODUCTION

Les systèmes logiciels ont par le passé été développés dans deux communautés séparées : Wet et Dry tel que c'est décrit par Goguen [Goguen, 1992]. La Communauté Wet est caractérisée par la mentalité "Hacker", dans laquelle le développeur veut créer le système le plus vite possible, et utilise pour cela des modèles heuristiques de conception basés sur l'expérience antérieure. Le Système peut (ou non) être documenté, et sa validation (par rapport à une spécification soft des besoins) peut être très difficile à mener. La deuxième alternative de développement est la communauté Dry dans laquelle uniquement les méthodes formelles sont d'usage permettant ainsi une conception rigoureuse. Néanmoins, les modèles utilisés sont souvent théoriques. L'objectif est donc de faire le lien entre les deux philosophies en termes de leurs principes de conception (heuristique et formelle) [Gamble et al., 1999]. Ceci permettra, d'une part, de raisonner sur des systèmes complexes en les caractérisant à un haut niveau d'abstraction (apport de la communauté Dry). D'autre part, ce lien permettra aux concepteurs d'exploiter des modèles récurrents d'organisation de systèmes. De tels modèles connus sous le nom de styles architecturaux [Monroe et al., 1997] facilitent le processus de conception en recourant à des solutions connues pour certaines classes de problèmes (apport de la communauté Wet). Ils peuvent mener à une réutilisation significative du code, facilitent la communication entre concepteurs et/ou utilisateurs et supportent l'interopérabilité. Dès lors que la conception d'architectures logicielles émerge comme une discipline du Génie Logiciel, cette démarche revêt une importance toute particulière.

En outre, l'industrie et la recherche informatiques, en général, et logicielles en particulier, doivent actuellement affronter des applications logicielles dont la complexité est sans cesse en croissance. L'évolution oblige les concepteurs des nouvelles applications à faire face au renforcement des exigences applicatives traditionnelles et à l'introduction de nouvelles exigences liées à une réadaptation des applications nécessitant une création et une distribution dynamique des composants et de leurs supports d'interaction. L'adaptabilité des applications à ces changements est une exigence récente qui requiert une configuration dynamique et une évolution fréquente des architectures des nouvelles applications. Ceci constitue depuis quelques années un domaine de recherche et de développement articulé autour de l'étude des architectures logicielles dynamiques.

L'objectif principal est donc de contrôler l'évolution et la reconfiguration de l'architecture en exprimant leur conformité par rapport à un style architectural.

Dans ce papier, nous proposons une approche qui profite des avantages du pouvoir expressif des langages fonctionnels. Elle exploite également les privilèges qu'offre les grammaires de graphes, notamment en terme de manipulation d'architectures. Nous décrivons donc le style architectural d'un système logiciel avec la notation Z [Abrial, 1985] et les opérations de reconfiguration via des règles de réécriture de graphe. Ces règles sont en fait exprimées via une intégration du langage fonctionnel Z et de la notation Δ [Kaplan et al., 1993] (notation inspirée des grammaires de graphes). Cette intégration possède l'avantage d'offrir au développeur une technique de spécification facile à appréhender tout en présentant un haut pouvoir d'expression. Ces règles prennent en considération les contraintes structurelles et fonctionnelles, définies pour le système, dans leurs conditions d'application assurant ainsi sa consistance durant son évolution. Nous décrivons toute la sémantique en notation Z, obtenant ainsi une approche unifiée. La notation Z est l'un des rares langages formels accepté par l'industrie et les communautés Génie Logiciel et Architecture Logicielle. Nous pouvons ainsi utiliser un démonstrateur de théorèmes qui supporte les spécifications Z tels que *Z/EVES* [ZEV,] et *Z-HOL* [Kolyang et al., 1996].

Ce papier est organisé comme suit : Dans la section 2, nous présentons brièvement les éléments clés d'une spécification en notation Z. la section 3 décrit l'approche proposée et ses principaux éléments. La section 4 présente une étude de cas qui illustre notre approche. Elle explique également le processus de vérification qui doit être appliqué. la section 5 présente une discussion des travaux reliés. Enfin, dans la section 6, nous donnons quelques remarques de conclusion et les perspectives de ce travail.

2 La notation Z

La notation Z, telle qu'elle est présentée dans [Spivey, 1992], offre un langage formel de spécification orienté modèle qui se base sur la théorie des ensembles et la logique des prédicats de premier ordre. L'élément clé de ce langage est la définition de l'espace d'état du système et des opérations possibles qui transforment un état à un autre. Une spécification Z type est un ensemble de schémas d'état et d'opérations. Un schéma d'état contient une partie permettant la déclaration de variables et une autre, prédicative, définissant une relation entre leurs valeurs :

$Schema - Name$

$Declaration$

$Predicate;\ ...;\ Predicate$

Les relations entre un état *avant* et un état *après* sont définies par les schémas d'opérations. Une opération peut être décrite en incluant les deux états : supposons que l'état d'un système est modélisé par un schéma *State*, il existe, en Z, une convention $\Delta State$ pour inclure deux copies du même schéma d'état, *State* et *State'*, pour l'état $avant$ et l'état *après*, respectivement. Un schéma d'opération s'écrit alors :

$Operation$

$\Delta State$

$...$

3. APPROCHE PROPOSEE

Compte tenu de la pertinence des graphes dans la représentation des structures [APP, 2001] et leur fondement mathématique, nous les avons adoptés pour représenter l'architecture d'un système logiciel de façon similaire aux approches de [Métayer, 1998] où les noeuds représentent les composants du système et les arcs décrivent les liens de communication entre ces composants. Un graphe d'architecture représente une configuration donnée d'un système. Ce dernier peut être amené à évoluer en modifiant son architecture. En fait, ces configurations ou architectures représentent des instances du style architectural d'un système. Celui-ci définit les types de composants pouvant intervenir et les types de relations qui peuvent relier ces composants d'une part, ainsi que l'ensemble des propriétés architecturales qui doivent être satisfaites par toutes les configurations appartenant à ce style [Allen et al., 1998]. Nous donnerons dans ce qui suit comment spécifier un style architectural en Z à partir de la définition générale d'un graphe d'architecture. Nous nous intéresserons plus particulièrement aux graphes orientés, étiquetés et typés. En général, un graphe d'architecture s'écrit: G=$\langle N, L, E, T, f\rangle$ où:

1. $E \subseteq N \times L \times N$, est un ensemble d'arcs ;
2. N est un ensemble de noeuds ;
3. L est un ensemble d'étiquettes ;
4. T=$\{t_1, t_2, ..., tn\}$ est un ensemble de types ;
5. f:$N \longrightarrow T$ est une fonction qui associe à chaque noeud un seul type.

Partant de cette définition, le méta modèle Z associé est représenté par le schéma suivant :

$system\_name$

$N_1 : \mathbb{F}\, N\_t_1;\ N_2 : \mathbb{F}\, N\_t_2;\ ...;\ N_n : \mathbb{F}\, N\_t_n$

$R_1 : N\_t_i \leftrightarrow l_j \leftrightarrow N\_t_k$

$R_2 : N\_t_s \leftrightarrow l_k \leftrightarrow N\_t_j$

$...$

$R_m : N\_t_u \leftrightarrow l_v \leftrightarrow N\_t_s$

Avec:

- $N = \bigcup_{i=1}^{n} N_i$
- $E = \bigcup_{i=1}^{m} R_i$
- T=$\{t_1, t_2, ..., t_n\}$
- L=$\{l_1, l_2, ..., l_n\}$

Notons qu'à ce niveau d'abstraction, nous n'avons pas besoin de considérer la représentation des noeuds de type t_i; nous les introduisons donc comme étant des types basiques de spécification: $[N\_t_1, N\_t_2, ..., N\_t_n]$. Ainsi, N_t_1, par exemple, dénote l'ensemble de noeuds de type t_1.
Prenons par exemple le style architectural Producteur/Consommateur. La définition d'un graphe d'architecture appartenant à ce style est comme suit:

$$G = \langle N, L, E, T, f\rangle$$

où:

- L=$\{pushP, pushS, pullC, pullS\}$;
- T=$\{Producer, Service, Consumer\}$;
- $E = \bigcup_{i=1}^{4} E_i$;
- $E_1 \subseteq \{x \in N \mid f(x) = Producer\} \times \{pushP\} \times \{y \in N \mid f(y) = Service\}$;
- $E_2 \subseteq \{x \in N \mid f(x) = Service\} \times \{pushS\} \times \{y \in N \mid f(y) = Consumer\}$;

- $E_3 \subseteq \{x \in N \mid f(x) = Consumer\} \times \{pullC\} \times \{y \in N \mid f(y) = Service\}$;
- $E_4 \subseteq \{x \in N \mid f(x) = Service\} \times \{pullS\} \times \{y \in N \mid f(y) = producer\}$;

Partant de cette définition, la spécification de ce style en Z est donnée par le schéma suivant:

$$
\begin{array}{|l}
\hline
Prod\_Cons[N\_Producer, N\_Service, N\_Consumer] \\
P : \mathbb{F}\, N\_Producer \\
S : \mathbb{F}\, N\_Service \\
C : \mathbb{F}\, N\_Consumer \\
pushP : N\_Producer \leftrightarrow N\_Service \\
pushS : N\_Service \leftrightarrow N\_Consumer \\
pullC : N\_Consumer \leftrightarrow N\_Service \\
pullS : N\_Service \leftrightarrow N\_Producer \\
\hline
\end{array}
$$

Notons que pour des raisons de simplification, nous écrivons : $pushP : N\_Producer \leftrightarrow N\_Service$, au lieu d'écrire : $R_1 : N\_Producer \leftrightarrow pushP \leftrightarrow N\_Service$.
Cette spécification constitue un schéma générique pour une architecture logicielle. Ainsi, $N\_Producer$, $N\_Service$ et $N\_Consumer$ peuvent être instanciés avec n'importe quels autres types de composants.
Les quatre opérations fondamentales fournies par la plupart des langages sont la création et la suppression des composants et des connexions [Wermelinger, 1999]. Les grammaires de graphes ont montré leur pertinence pour exprimer ces opérations de reconfiguration notamment dans les travaux décrits dans [Wermelinger, 1999] et [Métayer, 1998]. Nous proposons donc de les décrire via des règles de réécriture de graphe. Suite à l'étude effectuée sur les différentes notations graphiques proposées pour la réécriture de graphe (Y [Göttler, 1982], X [Göttler, 1992], Δ [Kaplan et al., 1993]), nous suggérons une nouvelle notation qui profite des avantages de la description visuelle et qui couvre les limites que nous avons soulevées et que nous détaillerons dans la suite. Notre description se compose de deux parties : une partie structurelle et une partie fonctionnelle. La partie structurelle est exprimée par la notation Δ qui est composée de cinq sections comme le montre la figure 1.

- *Rétraction* : le fragment de graphe enlevé durant la réécriture.
- *Insertion* : le fragment qui est créé et connecté dans le graphe hôte durant la réécriture.
- *Contexte requis* : le fragment identifié mais non modifié durant la réécriture.
- *Garde* : exprime des conditions sur les étiquettes qui apparaissent dans le graphe de la règle.
- *Restriction* : le fragment de graphe qui doit être absent pour que la réécriture ait lieu. Si le sous graphe correspondant au contexte et la rétraction peut être étendu pour correspondre à la restriction, alors la règle ne peut pas être appliquée à ce sous graphe.

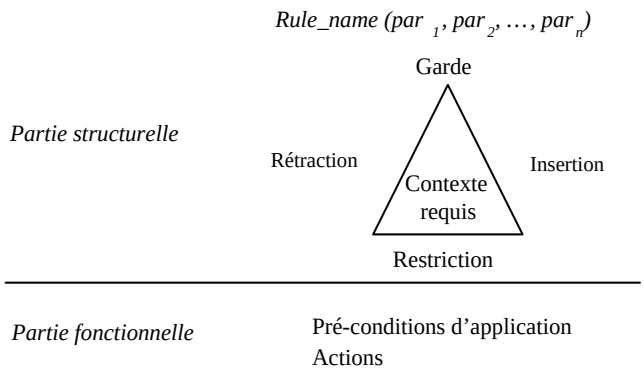


Figure 1. Notation mixte proposée

Une règle r est appliquée à un graphe g selon les étapes suivantes:

1. Un sous graphe isomorphe à $g_l = contexte\ requis \cup retraction$ est identifié dans g.
2. S'il n'existe pas de sous graphe isomorphe ou si le sous graphe isomorphe peut être étendu pour correspondre à g_l plus la Restriction (si elle existe), r ne peut pas être appliquée à g.
3. La garde, si elle existe, est évaluée. si elle est évaluée à faux, r ne peut pas être appliquée à g.
4. Les éléments du sous graphe isomorphe à la Rétraction sont retirés de g.
5. Un graphe isomorphe à l'Insertion est attaché au graphe hôte g via les arcs décrits entre le Contexte et l'Insertion dans r.

Quant à la partie fonctionnelle, elle exprime des préconditions d'application qui ont pour objectif le contrôle de l'évolution conformément aux propriétés du système. Elle exprime également les actions d'ordre fonctionnel à exécuter si la règle s'avère applicable. Nous exprimons cette partie en notation Z.
L'introduction de la partie fonctionnelle permet de résoudre les limites présentes dans les notations de transformation de graphes notamment leur pouvoir expressif en ce qui concerne les conditions et les actions non structurelles d'application. En effet, ces notations ne peuvent pas exprimer des conditions sur les valeurs des attributs des

noeuds du graphe par exemple ou des conditions faisant intervenir la disjonction et toute autre contrainte fonctionnelle. D'autre part, le fait de spécifier tout le système en utilisant les graphes uniquement entraîne dans certains cas l'augmentation de la complexité du graphe de la règle ce qui compliquerait ce graphe de point de vue lisibilité et compréhension, entravant de ce fait l'avantage principal de ces notations. La partie fonctionnelle permet au contraire d'exprimer par une simple expression logique, une condition assez compliquée si exprimée graphiquement. Nous proposons de décrire la sémantique de nos descriptions par le langage Z. Ainsi, chaque règle est traduite par un schéma d'opération comme suit :

$$
\begin{array}{|l}
\hline
Rule\_name \\
par_1?;\ par_2?;\ \ldots;\ par_n? \\
\Delta system\_name \\
\hline
Pré - conditions : \\
contraintes\ fonctionnelles : formules\ sur\ les \\
\ paramètres, formules\ sur\ les\ noeuds\ des\ attributs \\
,\ etc. \\
contraintes\ structurelles : formules\ sur\ les \\
relations :\ identification\ de\ g_l = contexte\ requis \\
\cup\ rétraction\ dans\ le\ graphe\ du\ système \\
Actions : \\
Opérations\ sur\ les\ ensembles\ des\ noeuds/arcs \\
(insertion/rétraction) \\
Actions\ fonctionnelles \\
\hline
\end{array}
$$

Où $Par_i?$ représentent les paramètres d'entrée de la règle et $\Delta system\_name$ indique que la règle peut changer l'état du système modélisé par le schéma $system\_name$.
L'application d'une règle de transformation s'effectue alors comme suit : si les contraintes fonctionnelles sont vérifiées et si les contraintes structurelles sont également satisfaites (existence du graphe décrit dans $g_l = contexte\ requis \cup retraction$ et l'absence du graphe décrit dans la partie Restriction) alors les actions peuvent être exécutées (insertion et/ou rétraction des noeuds et/ou arcs et autres actions fonctionnelles).

4 ETUDE DE CAS

Pour détailler davantage notre approche, nous présentons dans cette section un exemple simple : le système de contrôle de patients PMS (Patient Monitoring System) qui a été utilisé pour illustrer les travaux de [Métayer, 1998] et [Warren and Sommerville, 1995]. Pour représenter l'architecture de communication de ce système, nous avons choisi le style architectural Producteur/Consommateur.
A chaque service de la clinique (pédiatrie, cardiologie, maternité,) est associé un service d'événement pour gérer les communications entre les infirmières et les contrôleurs de lit rattachés au service en question. Chaque infirmière demande des informations relatives à ses patients en envoyant une requête au service d'événement auquel elle est liée. Ce service prend en charge cette demande et la transmet aux contrôleurs de lit des patients concernés. Lorsque l'état d'un patient est jugé anormal, son contrôleur de lit envoie un signal d'alarme au service d'événement auquel il est connecté. Ce service transmet alors ce signal à l'infirmière responsable. L'infirmière joue ainsi le rôle du composant consommateur et le contrôleur de lit celui du composant producteur.

4.1 Spécification du système

En plus des contraintes du style architectural, une application peut avoir des propriétés spécifiques qui doivent être toujours satisfaites durant l'évolution de son architecture. Nous allons prendre quelques propriétés du système PMS telles que:

- Le système doit contenir au maximum 3 services.
- Un service contient au maximum 5 infirmières et 15 patients.
- Un patient doit être toujours affecté à un et un seul service et ce service doit contenir au moins une infirmière pour pouvoir s'occuper de ce patient.
- Une infirmière doit être attachée à un seul service.

La spécification du système doit donc considérer les contraintes du style producteur/Consommateur et les propriétés spécifiques citées. Dans la notation Z, on peut combiner les informations contenues dans deux schémas en incluant un schéma dans la partie déclaration de l'autre. Les déclarations sont fusionnées et les prédicats sont conjoints. La spécification est donc comme suit: $[BED\_MONITOR, EV\_SER, NURSE]$

$$
\begin{array}{|l}
\hline
PMS1 \\
Prod\_Cons[BED\_MONITOR, EV\_SER, NURSE] \\
\hline
\end{array}
$$

PMS

$PMS1[PB/P, ES/S, CN/C]$
$NB\_CN : EV\_SER \rightarrow \mathbb{N}$
$NB\_PB : EV\_SER \rightarrow \mathbb{N}$

$\#ES \leq 3$
$\forall s : ES \bullet NB\_CNs \leq 5 \land NB\_PBs \leq 15$
$\forall x : PB \bullet$
$\exists_1 s : ES \bullet (x, s) \in pushP \land (s, x) \in pullS$
$\land NB\_CNs \geqslant 1$
$\forall s, z : ES;\ y : CN \bullet$
$(y, s) \in pullC \land (s, y) \in pushS \land (y, z) \in pullC$
$\land (z, y) \in pushS \Rightarrow s = z$

Notez que $N\_Producer$, $N\_Service$ et $N\_Consumer$ sont instanciés respectivement par $BED\_MONITOR$, EV_SER et $NURSE$ et nous avons renommé également quelques variables du schéma $Prod\_Cons[N\_Producer, N\_Service, N\_Consumer]$ pour faciliter leur utilisation; En notation Z, si $Schema$ est un schéma, alors on écrit $Schema[new/old]$.
La fonction NB_CN calcule pour un service donné le nombre d'infirmières connectées et la fonction NB_PB donne pour un service donné le nombre de patients affectés.
Il est donc évident que le graphe de la figure 2 est une instance possible du système PMS.

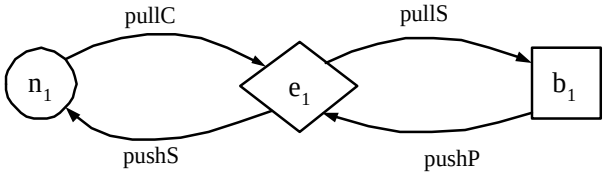


Figure 2. Une configuration possible du système PMS

4.2 Spécification de l'évolution

Dans cette section, nous présenterons la spécification des différentes règles permettant de faire évoluer notre système PMS tout en tenant compte des propriétés décrites.

- *Insertion d'un service d'événement:* cette règle permet d'insérer une instance de composant de type EV_SER à condition que le système ne contient pas déjà trois services d'événement comme l'impose le premier prédicat du schéma PMS. La représentation syntaxique de cette règle est donnée par la figure 3 et la sémantique par le schéma $insert\_ES$.

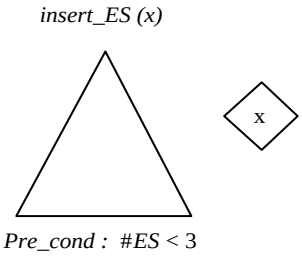


Figure 3. Règle d'insertion d'un service d'événement

$insert\_ES$

$x? : EV\_SER$
ΔPMS

$\#ES < 3$
$ES' = ES \cup \{x?\}$

Si nous appliquons cette règle au graphe de la figure 2 avec e_2 comme paramètre, nous obtiendrons le graphe donné par la figure 4.

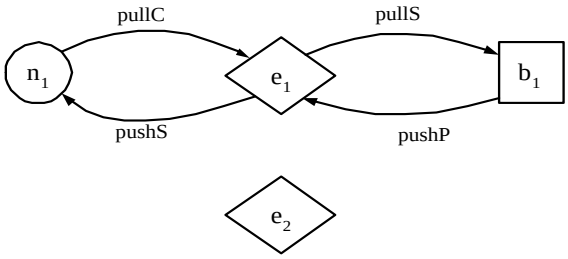


Figure 4. Graphe obtenu suite à l'application de la règle *insert_ES* **au graphe de la figure 2**

- *Insertion d'un patient:* l'insertion d'un nouveau patient dans le système se traduit par l'insertion d'un nouveau contrôleur de lit. Ainsi, cette règle permet d'insérer une instance de composant de type $BED\_MONITOR$ et de la lier à un service d'événement sous deux conditions : il faudrait d'abord qu'il y ait au moins une infirmière appartenant à ce service pour pouvoir s'occuper du nouveau patient. En plus, il faudrait vérifier que ce service ne contient pas déjà quinze patients, et ce conformément au deuxième prédicat du schéma PMS. La représentation syntaxique de cette règle selon la notation purement graphique est donnée par la figure 5. Cette représentation complique considérablement le graphe et rend difficile la compréhension de la règle. Tandis qu'une simple expression logique nous épargne cette

complexité et facilite la tâche de spécification comme le montre la figure 6. De plus, les actions d'ordre fonctionnel peuvent être exprimées. La sémantique est spécifiée par le schéma $insert\_PB$.

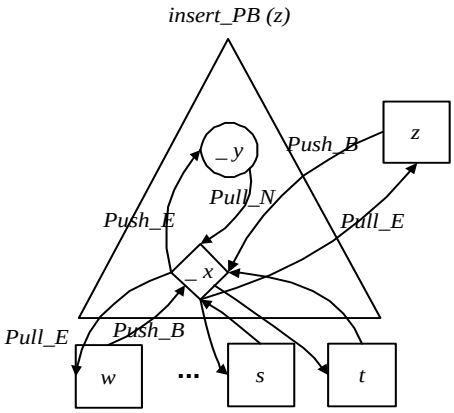


Figure 5. Règle d'insertion d'un patient (notation Δ)

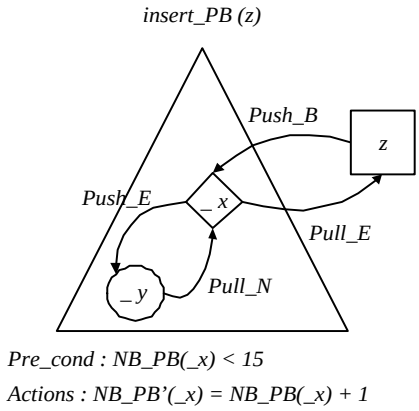


Figure 6. Règle d'insertion d'un patient (notre notation)

insert_PB

$z? : BED\_MONITOR$
ΔPMS
$x : EV\_SER$

$\exists\, y : CN \bullet x \in ES \wedge$
$(x, y) \in pushS \wedge (y, x) \in pullC \wedge$
$NB\_PBx < 15$
$PB' = PB \cup \{z?\}$
$pushP' = pushP \cup \{(z?, x)\}$
$pullS' = pullS \cup \{(x, z?)\}$
$NB\_PB'x = NB\_PBx + 1$

L'application de cette règle au graphe de la figure 4 avec b_2 comme paramètre, génère le graphe d'architecture donné par la figure 7.

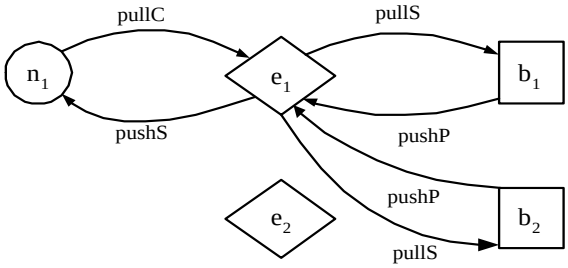


Figure 7. Graphe obtenu suite à l'application de la règle $insert\_PB$ au graphe de la figure 4

- *Insertion d'une infirmière:* cette règle permet d'insérer une instance de composant de type $NURSE$ et de la lier à un service d'événement quelconque (devant exister dans le système) via les liens précisés dans la partie déclaration du schéma PMS. Pour appliquer cette règle, on doit vérifier que le service en question ne contient pas déjà cinq infirmières conformément au deuxième prédicat du schéma PMS. La représentation syntaxique est donnée par la figure 8 et la sémantique est exprimée avec le schéma $insert\_CN$.

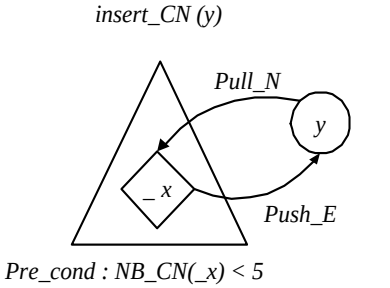


Figure 8. Règle d'insertion d'une infirmière

insert_CN

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$x \in ES \land NB\_CNx < 5$
$CN' = CN \cup \{y?\}$
$pushS' = pushS \cup \{(x, y?)\}$
$pullC' = pullC \cup \{(y?, x)\}$
$NB\_CN'x = NB\_CNx + 1$

Si nous appliquons cette règle au graphe de la figure 7 avec n_2 comme paramètre, il y aura deux choix possibles : l'infirmière peut être affectée au service e_1 ou au service e_2 ; le choix est non déterministe. On peut obtenir, par exemple, le graphe donné par la figure 9.

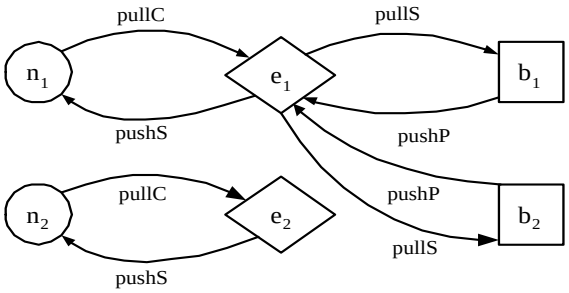


Figure 9. Graphe obtenu suite à l'application de la règle $insert\_CN$ au graphe de la figure 7

- Suppression d'une infirmière: une infirmière peut quitter le système si elle n'a aucun patient en charge. Cette règle supprime donc un composant de type $NURSE$ si et seulement s'il n'est pas connecté à un service contenant des patients et ne contenant pas d'autres infirmières. La syntaxe est donnée par la figure 10 et la sémantique par le schéma $supp\_CN$.

supp_CN

$y? : NURSE$
ΔPMS

$y? \in CN$
$\forall x : ES \bullet (y?, x) \notin pullC \land (x, y?) \notin pushS$
$CN' = CN \setminus \{y?\}$

Si on voudrait par exemple appliquer cette règle au graphe de la figure 9 avec n_1 comme paramètre, nous remarquerons que le deuxième prédicat de la règle n'est pas vérifié ; la règle est par conséquent inapplicable.

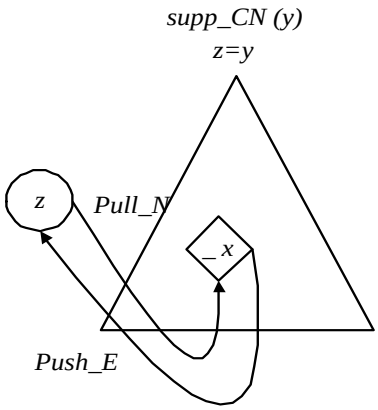
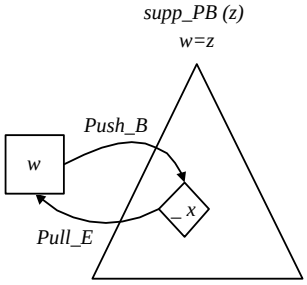


Figure 10. Règle de suppression d'une infirmière

- *Suppression d'un patient:* lorsqu'un patient quitte le système, le contrôleur de lit correspondant est déconnecté du service en question et supprimé du système. La syntaxe de cette règle est exprimée par la figure 11 et la sémantique par le schéma $supp\_PB$.



Actions : NB_PB '(_x) = NB_PB(_x) - 1

Figure 11. Règle de suppression d'un patient

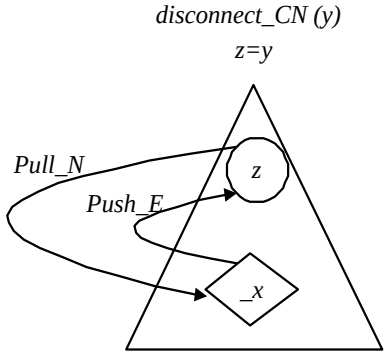
supp_PB

$z? : BED\_MONITOR$
ΔPMS
$x : EV\_SER$

$z? \in PB$
$x \in ES \land (x, z?) \in pullS \land (z?, x) \in pushP$
$PB' = PB \setminus \{z?\}$
$pushP' = pushP \setminus \{(z?, x)\}$
$pullS' = pullS \setminus \{(x, z?)\}$
$NB\_PB'x = NB\_PBx - 1$

- Déconnection d'une infirmière: une infirmière peut quitter son service (sans pour autant être supprimée

du système) seulement si ce service ne contient pas de patients ou alors il contient d'autres infirmières qui pourraient s'occuper des patients. Ainsi, cette règle ne permet de déconnecter un composant de type $NURSE$ que si l'une des deux conditions citées est satisfaite. La représentation syntaxique de cette règle selon notre notation est donnée par la figure 12. Nous pouvons relever dans ce cas précis une autre limite de la notation purement graphique : ce qui est exprimé simplement par l'opérateur de disjonction ne peut pas être exprimé graphiquement.



Pre_cond: NB_CN(_x) > 1 ∨ *NB_PB(_x) =* 0

Actions : *NB_CN'(_x) = NB_CN(_x) -* 1

Figure 12. Règle de déconnection d'une infirmière

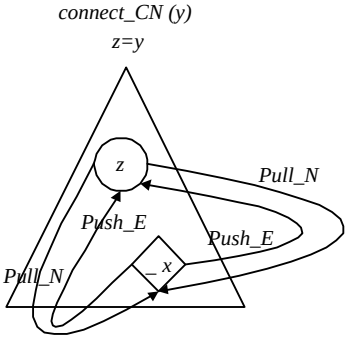
Nous exprimons la sémantique avec le schéma $disconnect\_CN$.

$disconnect\_CN$

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$y? \in CN \land x \in ES \land$
$(x, y?) \in pushS \land (y?, x) \in pullC$
$NB\_CNx > 1 \lor NB\_PBx = 0$
$pushS' = pushS \setminus \{(x, y?)\}$
$pullC' = pullC \setminus \{(y?, x)\}$
$NB\_CN'x = NB\_CNx - 1$

- *Connexion d'une infirmière:* Il faudrait d'abord vérifier que l'infirmière est libre, autrement dit, elle n'est pas attachée à un service. En plus, elle ne pourrait être connectée à un service que si ce dernier ne contient pas déjà cinq infirmières conformément au deuxième prédicat du système. La représentation syntaxique selon notre notation est donnée par la figure 13 ; notons qu'il s'agit ici de la même remarque de complexité faite pour la règle 5. La sémantique est spécifiée par le schéma $connect\_CN$.



Pre_cond : NB_CN(_x) < 5

Actions : NB_CN'(_x) = NB_CN(_x) + 1

Figure 13. Règle de déconnection d'une infirmière

$connect\_CN$

$y? : NURSE$
ΔPMS
$x : EV\_SER$

$y? \in CN$
$\forall z : ES \bullet (y?, z) \notin pullC \land (z, y?) \notin pushS$
$x \in ES \land NB\_CNx < 5$
$pushS' = pushS \cup \{(x, y?)\}$
$pullC' = pullC \cup \{(y?, x)\}$
$NB\_CN'x = NB\_CNx + 1$

En conclusion, nous pouvons remarquer la pertinence des grammaires de graphe en ce qui concerne la façon avec laquelle l'architecture est manipulée.

4.3 Vérification des propriétés

Dans cette section, nous allons montrer que la spécification proposée préserve la consistance du système et assure sa sécurité durant son évolution. Pour ce faire, nous devons prouver le théorème suivant:

$$\forall\, c, c' \in Conf, \forall\, r \in \textit{Rules}.(c\ \mathbf{sat} P) \land (c \xrightarrow{r} c') \Rightarrow c'\mathbf{sat} P$$

Informellement, ce théorème peut être traduit comme suit: c étant une configuration satisfaisant les propriétés requises P, et r étant une règle pouvant être appliquée sur c, alors la nouvelle configuration c' obtenue suite à l'application de r sur c, satisfait à son tour les dites propriétés. Pour cela il

suffit d'identifier pour chaque propriété les règles susceptibles de générer des architectures qui violent les propriétés correspondantes.
Prenons par exemple la propriété qui indique qu'un service contient au maximum cinq infirmières. Les seules règles pouvant éventuellement enfreindre cette propriété sont celles qui relient une infirmière à un service. Les règles concernées sont donc $insert\_CN$ et $connect\_CN$. Or celles-ci possèdent la condition suivante : $Nb\_CN(\_x) < 5$, et qui les empêche d'être exécutées si le service en question contient déjà cinq infirmières. Ainsi la propriété ne sera jamais violée.
Parmi les contraintes du PMS, il existe une qui stipule qu'un patient doit être toujours affecté à un service. Seules les règles qui manipulent les connexions des composants de type $BED\_MONITOR$ avec les composants de type EV_SER peuvent violer cette propriété. Dans notre cas, il s'agit des règles $insert\_PB$ et $supp\_PB$. Or, la règle $insert\_PB$ permet d'insérer une instance de composant de type $BED\_MONITOR$ tout en la connectant à un service. Quant à la règle $supp\_PB$ elle permet de supprimer complètement une instance $BED\_MONITOR$. Ainsi, cette contrainte ne risque pas d'être violée.
Nous pouvons vérifier également la contrainte qui stipule que le service auquel est affecté le patient doit contenir au moins une infirmière. Le risque réside dans les règles pouvant supprimer ou déconnecter une infirmière ou encore connecter un composant $BED\_MONITOR$ à un service. Les règles concernées sont donc: $insert\_PB$, $supp\_CN$ et $disconnect\_CN$. Or la règle $insert\_PB$ n'affecte un contrôleur de lit à un service que si ce dernier contient au moins une infirmière, ce qui est conforme à la propriété. La règle $supp\_CN$ ne supprime une infirmière que si elle n'est pas connectée à un service contenant des patients et ne contenant pas d'autres infirmières. Enfin, la règle $disconnect\_CN$ ne procède à la déconnection d'une infirmière de son service que si ce dernier ne contient pas de patients ou alors il contient au moins une autre infirmière. De ce fait, la propriété est effectivement préservée.

5 DISCUSSION

Notre étude des techniques de description et de reconfiguration des architectures logicielles nous a permis d'identifier deux courants de recherche. Le premier s'intéresse à développer des notations spécialisées appelées "langages de description d'architecture" (ADLs) afin de remplacer les diagrammes informels de type Box-and-line. Les ADLs tels que Darwin [Dulay et al., 1993], Rapide [Luckham and Vera, 1995, RAP, 1997], Wright [Allen et al., 1998] et LEDA [Canal et al., 1999] fournissent un support de modélisation pour aider les concepteurs à structurer le système et à composer les différents éléments. Les ADLs se limitent à la description de la dynamicité prédéfinie : ils s'intéressent uniquement à des systèmes ayant un nombre fini de configurations connues à priori. Par ailleurs, à l'exception du langage Wright, les approches basées sur les ADLs ne permettent pas de différencier les styles architecturaux et ne sont pas adaptées à la vérification des propriétés de l'architecture spécifiée.

Quant au deuxième courant, il consiste à appliquer des méthodes formelles existantes à la conception architecturale dans un but d'analyse et de vérification. Nous avons constaté que les travaux de ce courant sont pratiquement articulés autour de deux formalismes: les langages fonctionnels et les grammaires de graphes. Dans le contexte des langages fonctionnels, la logique temporelle a été utilisée dans [Aguirre and Maibaum, 2002] pour son grand pouvoir expressif et surtout pour ses possibilités de vérification. Néanmoins, cette logique est un formalisme axiomatique difficile à appréhender qui requiert des experts qualifiés. En outre, avec la logique temporelle, la reconfiguration n'est pas bien explicitée. Dans [Abowd et al., 1995], les auteurs ont développé des modèles formels avec Z pour les styles filter-pipe, et systèmes orientés évènements et dans [Gamble et al., 1999], Gamble et al. ont utilisé Z pour spécifier un style dit à base de règles. Toutefois, ces travaux n'évoquent pas les problèmes liés à l'évolutivité des architectures des applications.

Quant au principe du deuxième formalisme, il consiste à utiliser les graphes pour la représentation des structures, étant donné qu'ils constituent le moyen mathématique le plus naturel et intuitif pour la représentation de situations complexes. Dans ce contexte, [Métayer, 1998] propose de représenter l'architecture d'un système logiciel par un graphe et le style architectural via une grammaire de graphe à contexte libre. Ce type de grammaires ne permet pas de décrire certaines propriétés logiques permettant par exemple de raisonner sur le nombre d'instances d'un composant donné. Quant à la reconfiguration, elle est décrite par un ensemble de règles de réécriture dont la description est assez simple et compréhensible, et qui explicitent bien le changement topologique effectué. Néanmoins, ces règles ne permettent pas d'exprimer certaines conditions logiques telles que l'absence d'un lien de communication entre deux entités logicielles.

Notre approche, quant à elle, porte sur le problème de l'évolution en considérant la dynamique comme une séquence d'opérations non connues à priori. D'autre part, les transformations structurelles sont bien explicitées grâce à l'utilisation des grammaires de graphes, et toutes les contraintes et les actions d'ordre fonctionnel et parfois même d'ordre structurel peuvent être exprimées en notation Z. De plus, nous exprimons toute la sémantique en notation Z, permettant par conséquent une analyse rigoureuse des styles architecturaux.

6 CONCLUSION

Nous avons présenté dans cet article une approche de spécification formelle des architectures dynamiques des systèmes orientés composants. Cette approche repose sur une intégration des sémantiques basées sur les graphes dans le contexte de la notation Z. Ceci facilite la tâche du développeur en lui offrant une technique de spécification facile à appréhender tout en lui permettant de raisonner rigoureusement sur les styles architecturaux. De plus, notre approche permet de décrire formellement la dynamique d'une architecture logicielle via des règles de réécriture de graphe. Ces règles prennent en considération les propriétés spécifiant les contraintes pour appliquer des opérations de reconfiguration, ce qui assure la consistance du système durant son évolution. Nous utilisons actuellement $Z/EVES$ [ZEV,], un outil qui supporte la notation Z, pour spécifier les architectures logicielles et raisonner sur leur dynamique. En perspective, nous envisageons de prendre en considération la description de l'aspect comportemental des composants logiciels. Pour cela nous sommes en cours d'étudier l'intégration d'un langage de calcul de processus avec la notation Z.

Bibliographie

[ZEV,] Z/eves. http://www.ora.on.ca/Z-eves.

[RAP, 1997] (1997). *Guide to the Rapide 1.0 Language Reference Manuals*. Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University.

[APP, 2001] (2001). Appligraph. applications of graph transformations. Technical Report 22565, Hans-jörg Kreowski and Detlef plump editors.

[Abowd et al., 1995] Abowd, G. D., Allen, R., and Garlan, D. (1995). Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364.

[Abrial, 1985] Abrial, J. R. (1985). *Programming as a mathematical exercise*. In C.A.R. Hoare, editor, Mathematical Logic and Programming Languages. Prentice-Hall International.

[Aguirre and Maibaum, 2002] Aguirre, N. and Maibaum, T. (2002). A temporal logic approach to component-based system specification and reasoning. In *5th ICSE Workshop on component-based software engineering*, Orlando, Florida, USA.

[Allen et al., 1998] Allen, R., Douence, R., and Garlan, D. (1998). Specifying and analysing dynamic software architectures. *Journal of Fundamental Approaches to Software Engineering*, 1382:21–37.

[Canal et al., 1999] Canal, C., Pimentel, E., and Troya, J. (1999). Specification and refinement of dynamic software architectures. *Journal of Software Architecture*, pages 107–125.

[Dulay et al., 1993] Dulay, J., Kramer, N., and Magee, J. (1993). Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2):73–82.

[Gamble et al., 1999] Gamble, R., Stiger, P., and Plant, R. (1999). Rule-based systems formalised within a software architectural style. *Elsevier Knowledge-Based Systems*, 12:13–26.

[Goguen, 1992] Goguen, J. (1992). The dry and the wet. In Falkenberg, E., Rolland, C., and El-Sayed, E.-S. N.-E.-D., editors, *proceedings of IFIP Working Group 8.1 Conference*, Information Systems Concepts, pages 1–17. Alexandria, Egypt, Elsevier North-Holland.

[Göttler, 1982] Göttler, H. (1982). Attributed graph grammars for graphics. graph grammars and their application to computer science. *Computer Sciences*, 153:130–142.

[Göttler, 1992] Göttler, H. (1992). Diagram editors = graphs + attributes + graph grammars. *International Journal of Man-Machine Studies (IJMMS)*, 37:481–502.

[Kaplan et al., 1993] Kaplan, S. M., Loyall, J. P., and Goering, S. K. (1993). Specifying concurrent languages and systems with δ-grammars. *Research Directions in Concurrent Object-Oriented Programming*.

[Kolyang et al., 1996] Kolyang, Santen, T., and Wolff, B. (1996). A structure preserving encoding of z in isabelle/hol. In von Wright, J., Grundy, J., and Harrison, J., editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag.

[Luckham and Vera, 1995] Luckham, D. C. and Vera, J. (1995). An event-based architecture definition language. *IEEE transactions on Software Engineering*, 21(9):717–734.

[Métayer, 1998] Métayer, D. L. (1998). Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553.

[Monroe et al., 1997] Monroe, R., Kompanek, A., Melton, R., and Garlan, D. (1997). Architectural styles, design patterns, and objects. *IEEE Software*, pages 43–52.

[Spivey, 1992] Spivey, M. (1992). *The Z notation*. Prentice Hall International.

[Warren and Sommerville, 1995] Warren, I. and Sommerville, I. (1995). Dynamic configuration abstraction. In Schäfer, W. and Botella, P., editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 173–190. Springer-Verlag.

[Wermelinger, 1999] Wermelinger, M. (1999). *Specification of software architecture reconfiguration*. PhD thesis, Université Nova de Lisbon.

AN EVALUATION OF THE FBDM FRAMEWORK DESIGN METHOD

N. Bouassida,
Assistant en Informatique de gestion
Nadia.Bouassida@isimsf.rnu.tn

H. Ben-Abdallah
Maître Assistant en Informatique
Hanene.benabdallah@fsegs.rnu.tn

A. Ben Hamadou
Professeur en Informatique
Abdemajid.Benhamadou@isimsf.rnu.tn

Adresse professionnelle
Laboratoire LARIM, ISIMS, Route Mharza km 1.5, 3018, Sfax, Tunisie

Résumé : Cet article présente des travaux pertinents à la réutilisation architecturale. D'une part, il présente la méthode de conception de frameworks FBDM (Framework Based Design Method). D'autre part, il évalue FBDM à travers une étude de cas dans le domaine des éditeurs graphiques. La méthode FBDM offre un langage de conception, un processus de conception et un outil CASE. Le langage de FBDM est un profile UML, qui étend UML avec des annotations graphiques afin de distinguer entre le noyau du framework et les points de variation. Le processus de conception de FBDM génère un framework d'une manière semi-automatique en unifiant un ensemble d'applications dans le domaine du framework; le développeur du framework doit décider de la complétude de quelques relations faisant partie des points de variation. La méthode FBDM est évaluée à travers un framework d'éditeurs graphiques. Elle compare le framework généré par FBDM avec le framework populaire JHotDraw. De plus, l'évaluation utilise des métriques de conception pour examiner la qualité du framework généré et l'apport de la notation de FBDM.

Summary : This paper presents work pertinent to architecture reuse. On one hand, it presents the framework design method FBDM (Framework Based Design Method) which offers a design language, a design process and a CASE toolset. On the other hand, it reports on an experimental evaluation of FBDM in the domain of graphical drawing editors. The design language of FBDM is a UML profile that extends UML with graphical annotations to distinguish between a framework core and hot-spots. The FBDM design process generates a framework semi-automatically by unifying a set of applications in the framework domain; the developer is probed to decide on the completeness of some relations. The second part of this paper uses a graphical drawing editor framework to evaluate FBDM. It compares the framework generated by FBDM to the popular JHotDraw framework. In addition, it uses design metrics to examine the quality of the generated framework and the usefulness of the notation of FBDM.

Mots clés : Réutilisation, conception de frameworks, métriques de conception, FBDM.

Keywords : Reuse, framework design, design metrics, FBDM.

An Evaluation of the FBDM Framework Design Method

1 – INTRODUCTION

As computerized systems became larger and increasingly complex, the benefits of reuse became irrefutable. In fact, the software engineering community has been applying several object-oriented based reuse techniques ranging from class and component libraries (c.f., (Meyer, 1988) (Szyperski, 1996)) to, more recently, generic architectures (a.k.a. frameworks (Johnson, 1998)) and evolving models (c.f., MDA (OMG, 2001)).

The work presented in this paper is an academic contribution to architectural reuse through frameworks. An object-oriented framework offers the skeleton of applications in a given domain, that can be customized by an application developer (Fayad, 1998). Hence, it allows the reuse of design models, code and valuable domain expertise (Johnson, 1998). Therefore, it ensures an increased productivity, a shorter development time and a higher quality of applications.

Overall, a framework is composed of immutable parts, called *frozen-spots* or *core*, and adaptable parts, called *hot-spots*. A frozen-spot describes the typical software components and is, therefore, present in any application generated from the framework. On the other hand, a hot-spot allows the framework extension to a particular application, and helps in tracing the evolution of a given design.

One of the problems impeding the widespread use of frameworks is the difficulty of their development, combined with the lack of a standard notation that helps to identify the hot-spots. This problem motivated several design notations based on UML (c.f., (Pree, 1994) (Fontoura et al., 2000)) and a few design processes (c.f., (Schmid, 1997)). However, none of the proposed notations distinguishes between the core of the framework and the hot-spots and none of them relies on a formal semantics. Moreover, none of the design processes proposed in the literature gives rules that guide the generation of the framework and that determine its different components.

Clearly identifying (both finding and denoting) the core and the hot-spots of a framework is essential in understanding and reusing the framework. This motivated our work in proposing a framework design method, called FBDM (Framework Based Design Method, that offers a design language, called F-UML with a precise semantics (Bouassida et al., 2003), a design process (Bouassida et al., 2002) and a CASE tool, called F-UMLTool (Bouassida et al., 2004). The F-UML language is a UML profile that increases the expressiveness of UML with framework specific concepts and that guides a framework design reuse. It adds tags and graphical annotations to UML diagrams; the extensions help to distinguish visually between the core of a framework and its hot-spots and guide the user in deriving a specific application in the framework domain. On the other hand, the FBDM design process is a bottom-up process that generates a framework design by applying a set of unification rules to concrete application designs. In addition, it generates a framework with a minimum intervention from the developer; this latter is probed only to decide on the completeness of some structural relations (inheritance, composition,…), a decision often requiring domain-specific knowledge.

After an introduction to the FBDM method, this paper presents an experimental evaluation of FBDM in the domain of graphical drawing editors; the choice of the domain was motivated by the presence of a popular, mature and open source framework in this domain (JHotDraw (Gamma, 2000)).

The FBDM evaluation was conducted in two phases. First, an intrinsic evaluation tested the completeness of the design unification rules and the quality of the generated framework. Secondly, a comparative evaluation tested the correctness of the design process and evaluated the usefulness of the F-UML notation.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the framework design method FBDM. Section 4 presents the three graphical drawing applications as well as the framework generated through FBDM. Section 5 presents the evaluation metrics used and reports on the results of our two-phase evaluation.

Section 6 concludes with a summary of the presented work and outlines future works.

2. RELATED WORK

Classical design languages (e.g., UML) and processes (e.g., OMT (Rumbaugh, 1991)) lack concepts to determine and express the variability in frameworks. This motivated several researchers to propose design languages and processes specifically for frameworks. We next review framework design languages and processes that are based on the *de facto* standard language UML.

2.1 – UML-based Framework Design Languages

Fontoura et al. (Fontoura et al., 2000) propose a UML profile for frameworks, called UML-F, where a design is expressed by a class diagram and a set of sequence diagrams. This notation extends the two UML diagrams by *presentation tags* (e.g., complete, incomplete), *basic modeling tags* (e.g., fixed, application, framework) and *essential pattern tags* (e.g., FacM-Creator, FacM-ConcreteCreator). The added tags are used to mark, essentially, the complete/incomplete parts and the variable parts in the diagrams and the roles of diagram elements. In this notation, the extended class diagram represents the framework classes and relations. However, according to the tag definitions, this notation represents only the whitebox hot-spots; the second type of hot-spots, called blackbox (Schmid et al., 1997), are therefore not expressible in this notation. In addition, several tags are complementary and thus are redundant (e.g., complete and incomplete, application and framework). Furthermore, the pattern tags combined with the presentation tags could overcharge the class diagram and impede the design understanding.

Sanada et al. (Sanada et al., 2002) present an UML extension that "aims to be comprehensive and well defined". Most of their proposed extensions have already been defined by Fontoura (Fontoura, 2000); the essential difference is the constraint *covariant* which shows that adding a subclass to a certain class might result in adding a subclass to another class in the class diagram.

Riehle (Riehle, 1998) proposes a role modeling language that adapts the OORAM methodology (Reenskaug, 1996). The proposed language represents a framework through a class model with an *extension-point class set* (places of extension), *a built-on class set* (the framework interface) and *a free role type set* (usages of the framework by other frameworks). This language focuses more on framework composition than framework adaptation. For instance, it does not visually distinguish between extension-point classes and frozen classes in the framework. Therefore, one cannot easily recognize the whitebox and blackbox hot-spots.

Overall, the proposed UML-based design languages for frameworks lack the visual distinction between the core and the two hot-spot types. This distinction is essential in understanding and reusing a framework. In addition, none of the proposed languages relies on a formal semantics. This latter is vital in analyzing a framework and validating its reuses.

2.2 – Framework Design Processes

A design process for frameworks should systematically help to identify and derive a framework core, blackbox and whitebox hot-spots. It could follow either a top-down or bottom-up strategy. Bottom-up design works well where a framework domain is already well understood, for example, after some initial evolutionary cycles. In this case, the design process starts from a set of existing applications and generalizes them to derive a framework design (c.f., (Koskimies, 1995), (Fontoura, 2000)). On the other hand, top-down design is preferred when the domain has not yet been sufficiently explored. In this case, the design process starts from a domain analysis and then constructs the framework design (c.f., (Aksit, 1999)).

Koskimies and Mossenback (Koskimies et al., 1995) propose a two-phase bottom-up framework design process. The first phase, called *problem generalization*, incrementally generalizes a representative application in the framework domain into "the most general" form. In the second phase, called *framework design*, the generalization levels of the previous phase are considered in a reverse order, leading to an implementation for each level. The last step in the second phase applies the resulting framework to the initial application. This design process does not provide for reuse guidelines; that is, it does not clearly identify nor does it guide the designer in finding the framework core and hot-spots.

Schmid (Schmid et al., 1997) decomposes the framework design process into three steps: 1) design of a class model for an (arbitrary) application in the framework domain; 2) analysis and specification of the domain variability and flexibility, i.e., identification of the hot-spots; and 3) generalization of the class model by applying a sequence of transformations that incorporate the domain variability. This design process leaves the identification of hot-spots, during the second step, to the developer's expertise.

Fontoura et al. (Fontoura et al., 2000) propose a design process that considers a set of applications

as viewpoints (i.e., perspectives) of the domain. The process informally defines a set of unification rules that describe how the viewpoints can be combined to derive a framework. This design process neither distinguishes between the two hot-spot types, nor does it specify the object interactions. In addition, this process does not address semantic issues in the unified applications (e.g., synonyms, homonyms,…); it supposes that all the semantic inconsistencies between the viewpoints have been solved beforehand.

3. THE FRAMEWORK DESIGN METHOD FBDM

A framework design consists in a framework design notation and a framework design process that guides the development of a framework. On one hand, the framework design notation must provide for a means to describe the framework static structure (the classes and their relations), the core, and the whitebox and blackbox hot-spots. In addition, it has to show explicitly the collaborations between objects instantiated from the framework classes, and to clarify object responsibilities. On the other hand, the framework design process has to provide for design rules helping in the design of frameworks and to provide for a methodical, systematic way to design frameworks. In addition, it must guide the user in determining the core and hot-spots of the framework.

These reasons motivated us to propose the FBDM design method with its design language F-UML and process.

3.1 - The Framework Design Language F-UML

A framework design expressed in the F-UML language is composed of four UML extended diagrams:

- A use case diagram that specifies the framework scope, objectives and domain limits. Table 1 summarizes and explains the F-UML extensions added to UML.
- A class diagram that describes the static architecture of a framework. Table 2 presents the F-UML notation for the class diagram.
- A pattern diagram that shows the design patterns (Gamma et al., 1995) and meta-patterns (Pree, 1994) in order to delimit the roles of the framework classes.
- A set of sequence diagrams that describe object interactions. The extensions to the UML sequence diagram are described in Table 3.

In F-UML, a blackbox hot-spot represents a component that can be selected (without modification) in an application derived from the framework. On the other hand, a whitebox hot-spot can be selected and modified to meet the special requirements of an application derived from the framework. For example, a whitebox hot-spot class can be modified by specializing it (i.e., adding an inheriting class) adding/removing an attribute/operation, or redefining one of its operations. Note that, since the use cases represent the design at a high level of abstraction, they can not include whitebox hot-spots.

Being a UML profile, the F-UML language could be adopted easily by the UML community. In addition, having a precise, formal semantics (Bouassida et al., 2003) F-UML can be combined with a formal method that allows the verification and validation of framework designs. Currently, the F-UML semantics is expressed in Object-Z (Smith, 2000) and verification is conducted with the Z/Eves theorem prover (Bouassida et al., 2005).

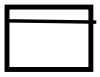
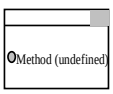
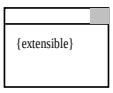
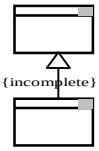
Notation	Explanation
	a framework core
	a framework blackbox hot-spot
	a framework whitebox hot-spot
	a re-definable (virtual) method in a whitebox hot-spot
	a method with an undefined signature
	an adaptable class interface (whitebox hot-spot)
	the framework may be adapted by adding other related classes (inheritance, composition,…)

Table 1. F-UML class diagram notation

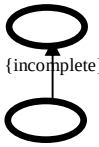
Notation	Explanation
	a framework core use case (actor)
	a framework hot-spot use case (actor)
	to show that it is possible to add an inheriting actor (use case) in an application adapting the framework

Table 2. F-UML use case diagram notation

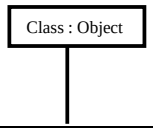
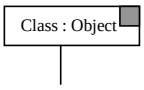
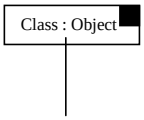
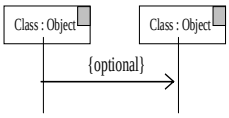
Notation	Explanation
	a framework core object
	a framework whitebox hot-spot
	a framework blackbox hot-spot.
	the message may not exist in an application reusing the framework

Table 3. F-UML sequence diagram notation

3.2 - The FBDM design process

The FBDM design process is a bottom-up process that takes several applications in a given domain and applies a set of unification rules to derive a framework in the F-UML notation. (Roberts (Roberts 1996) states that three applications sufficiently represent their domain). In addition, the FBDM design process helps the framework designer to structure the framework by determining automatically its core and hot-spots.

Overall, the design process is based on architectural and noun comparisons. Architectural comparisons examine the structure of the applications defined through the different relations (composition, aggregation …). Since a framework offers a basic architecture for the derived applications, architectural comparison of the unified applications is, therefore, a reasonable approach.

The second type of comparisons used in the FBDM design process is noun comparisons. These latter examine the nouns attributed to the different entities (actors, use cases, classes, attributes, methods …) and, thus, deal with the semantics included in the applications through naming. These comparisons rely on the hypothesis that nouns are well chosen to reflect the roles of software entities. More specifically, they use semantic relationships (e.g., equivalence, generalization-specialization, composition) between nouns of classes, attributes and methods that appear in the applications to be unified. In addition, the unification rules compare the method signatures, which, combined with the noun comparison, give an insight on the dynamic behavior (or roles) of the classes.

In the remainder of this section, we note a class C in an application A_i as C_{Ai}, a use case U in an application A_i as U_{Ai}, and an actor A in an application A_i as A_{Ai}.

3.2.1 Unification of the use case diagrams

To design the framework use case diagram, the unification process first extracts use cases (actors) *common* to all of the applications and puts them as the framework core. Secondly, it extracts the different use cases (actors) and puts them as blackbox hot-spots. For this, it uses the following four semantic relations:

- *N_equiv(A_{A1},...,A_{An})* means that the actor names are either identical or synonym.
- *N_var(A_{A1},...,A_{An})* means that the actor names are a variation of a concept, e.g., employee-contractual, employee-permanent, employee-vacationer.
- *Gen_Spec(A_{A1};A_{A2},...,A_{An})* means that the name of the actor A_{A1} is a generalization of the names of A_{A2} ,…, A_{An}, e.g., $Person_{A1}$-$Employee_{A2}$.
- *N_dist(A_{A1},...,A_{An})* means that the name of the actor A_{A1} has neither an equivalent nor a variation, nor a generalization in the other applications.

The design of the framework use case diagram through unification is guided by the six rules depicted in Figure 1.

3.2.2 - Unification of the class diagrams

To design the framework class diagram, the unification process determines the semantic correspondence between the classes, using the following criteria:

- class name comparison criteria to compare semantically the names;
- attribute comparison criteria to compare the names and types of the attributes; and
- operation comparison criteria to compare the names and signatures of the operations.

Note that, when comparing the set of attributes and methods of two classes, the unification rules take into account the interpretation of the inheritance relation. Thus, they consider the set of attributes (operations) of a class as those contained in the given class augmented with the set of attributes (operations) of the super classes from which it inherits.

Class name comparison criteria: They express the linguistic relationship among class names that belong to different applications. We defined five types of relations between classes:

- *N_equiv(C_{A1},...,C_{An})*, *N_var(C_{A1},...,C_{An})*, *Gen_Spec(C_{A1};C_{A2},...,C_{An})*, and *N_dist(C_{A1},..., C_{An})* are defined in a similar manner to their counter parts for the use case diagram; and
- *N_comp(C_{A1};C_{A2},...,C_{An})* which means that the class name C_{A1} is linguistically a composite of the components C_{A2},…,C_{An}, e.g., $House_{A1}$-$Room_{A2}$.

Attribute comparison criteria: They express the relationship between the attribute names and the attribute types of application classes. They are grouped into four categories:

- *Att_equiv(C_{A1},..., C_{An})* implies that the classes have identical or synonym attribute names with the same types.
- *Att_int(C_{A1},..., C_{An})* implies that C_{A1},..., C_{An} have attributes in intersection.
- *Att_dist(C_{A1},..., C_{An})* implies that no attribute of C_{A1},..., C_{An} is common with the attributes of the others.
- *Att_conf(C_{A1},..., C_{An})* implies that there exists at least one attribute of C_{A1} with a name equivalent to the attributes of C_{A2},..., C_{An} but their types are different.

Operation comparison criteria: Operation comparison consists in comparing the operation names and signatures (returned types and parameter types). The operation comparison criteria *Op_equiv(C_{A1},...,C_{An})*, *Op_int(C_{A1},...,C_{An})*, *Op_dist(C_{A1},...,C_{An})*, and *Op_conf(C_{A1},...,C_{An})* are defined in a similar manner to the attribute comparison criteria.

The class diagram unification process is depicted in Figure 2. (For a detailed description of the unification rules, the reader is referred to (Bouassida et al., 2002). As illustrated in Figure 2, the unification rules determine automatically the framework core and hot-spots. The designer is probed only to decide on the completeness of some relations (Rule 3.d and 7). This information is, in fact, application dependant and requires domain knowledge.

In addition, in Figure 2, **Rule *5*** deals with the generalization-specialization relation in a manner similar to *N-Comp* relation of **Rule *4***. Furthermore, **Rule *3.b*** may add new inheriting classes to the framework. The addition depends on the significance of the number of attributes and methods in an inheriting class *C* with respect to another class *C'*. This is defined using the ratio R_{sig}:

$$R_{sig}(C, C') = \frac{\text{number of attributes and methods that belong to C and C'}}{\text{number of attributes of C + number of methods of C}}$$

Informally, a class *C* has a significant number of attributes and methods with respect to a class *C'*, if R_{sig} is greater than a fixed threshold (e.g., 50%) that can be fixed by the framework designer. This ratio fixes the level of details the framework designer would like to include in the framework.

Further, ***Rule 6*** adds or ignores hot-spot classes according to the domain coverage ratio R_{dc} :

$$R_{dc}(C) = \frac{\text{number of occurrences of C(or its variations or its equivalents) in A1,.., An}}{\text{number of applications}}$$

Informally, this ratio is used to determine the reuse potential of a class. If a class is present in several applications, then it covers an important space of the framework domain; thus, it must be present in the framework hot-spot. On the other hand, if a class is present in few applications, it is too application specific; thus, if it is added to the framework, it may complicate unnecessarily the framework comprehension.

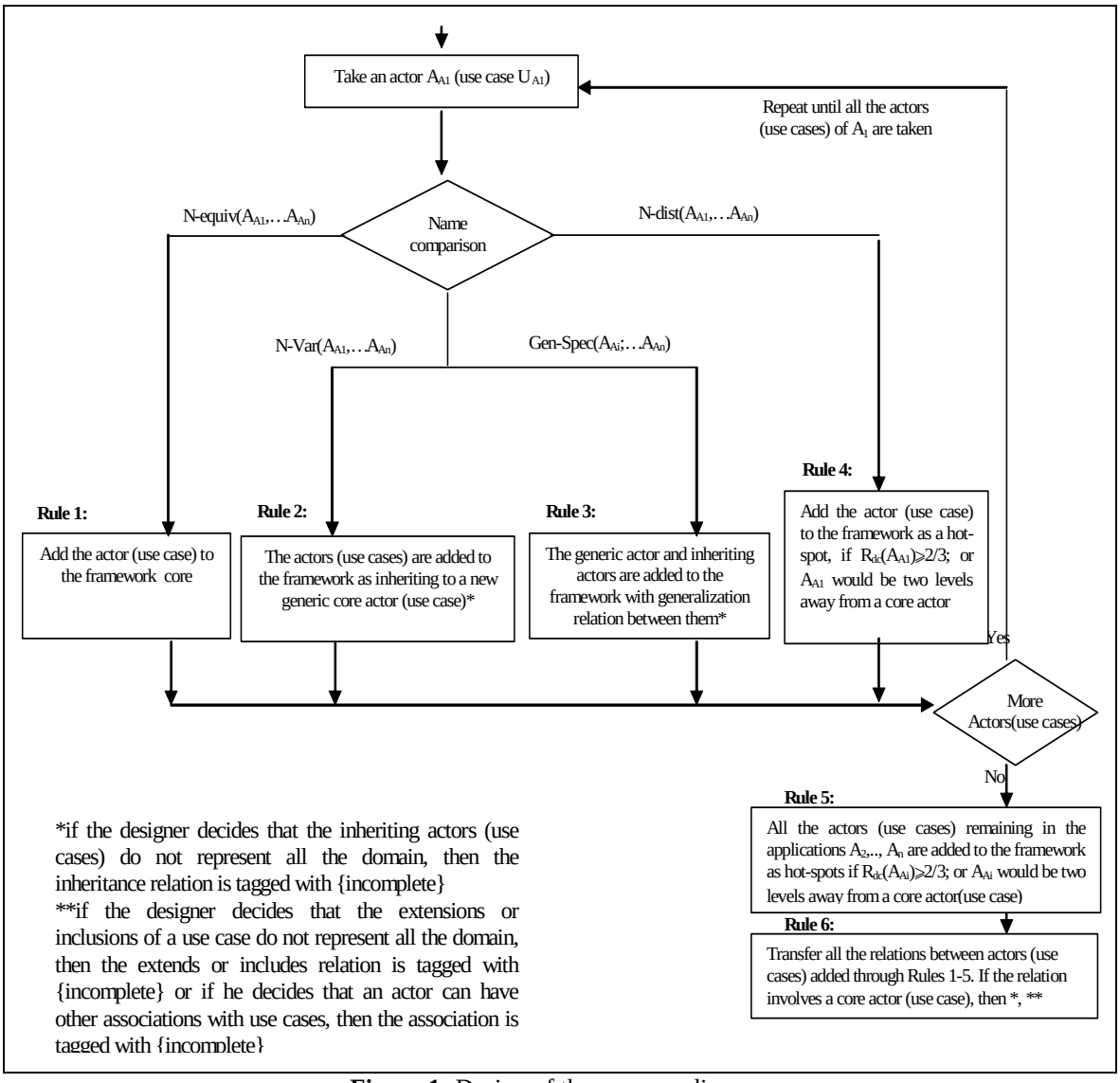


Figure 1: Design of the use case diagram

3.2.3 - Unification of the sequence diagrams

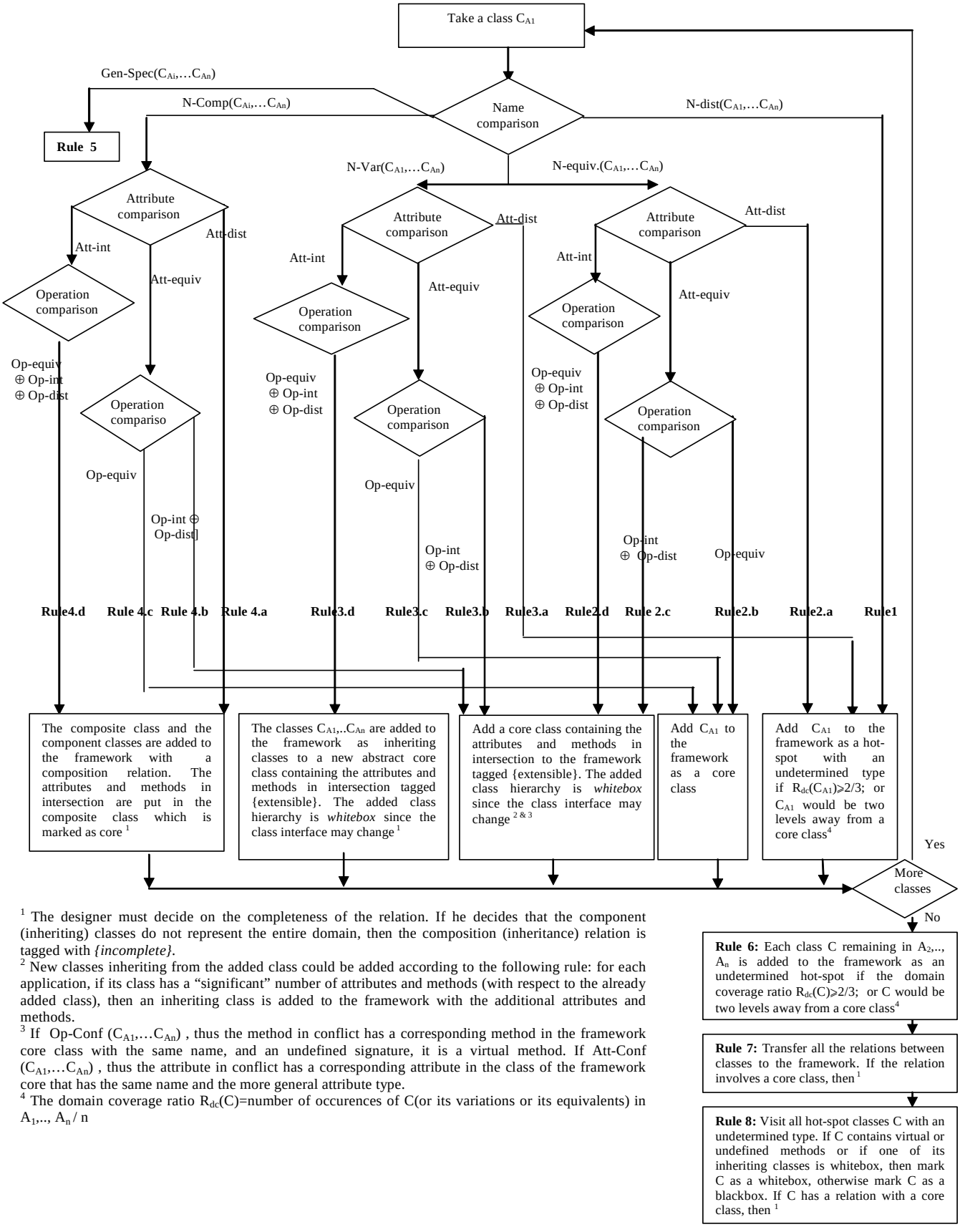
The framework sequence diagrams are obtained by the unification of the sequence diagrams which represents semantically equivalent scenarios. Briefly, the unification process takes the "union" of all the sequence diagrams of the given applications and marks any message as *{optional}* if it does not appear in all the applications. Any sequence diagram that does not have equivalent diagrams in the other applications being unified is transferred to the framework with all its messages marked *{optional}*.

For more details on the six unification rules of the sequence diagrams, the reader is referred to (Bouassida et al., 2002).

Finally, we note that the FBDM design process can be optimized in two ways. First, the unification can start with the application having the minimum number of elements (i.e., use cases/actors, classes, and messages). Secondly, in the unification rules of the class diagrams, the comparisons can be limited to "significant" attributes and operations, i.e., those that are not omni present in the classes; for example, the creator and destructor operations in a class.

4 - THE GRAPHICAL DRAWING EDITOR CASE STUDY

Two motivations were behind our choice of the graphical drawing editor domain. First, an open-source and mature framework already exists (JhotDraw (Gamma et al., 2000)) and, secondly, several derived applications also exist. The availability of the framework and its applications allowed us to conduct both an intrinsic and a comparative evaluation of FBDM.



[1] The designer must decide on the completeness of the relation. If he decides that the component (inheriting) classes do not represent the entire domain, then the composition (inheritance) relation is tagged with *{incomplete}*.

[2] New classes inheriting from the added class could be added according to the following rule: for each application, if its class has a "significant" number of attributes and methods (with respect to the already added class), then an inheriting class is added to the framework with the additional attributes and methods.

[3] If Op-Conf (C_{A1},…C_{An}) , thus the method in conflict has a corresponding method in the framework core class with the same name, and an undefined signature, it is a virtual method. If Att-Conf (C_{A1},…C_{An}) , thus the attribute in conflict has a corresponding attribute in the class of the framework core that has the same name and the more general attribute type.

[4] The domain coverage ratio R_{dc}(C)=number of occurences of C(or its variations or its equivalents) in A_1,.., A_n / n

Figure 2: Class diagram design

Recall that the FBDM design process starts from existing applications. For this, we have chosen the following three applications (due to the availability of their source code): JARP, a graphical composer for petri nets (JARP, 2001); JOONE, a Java framework to create, train and run neural networks (JOONE, 2001); and RENEW, an editor for drawing reference nets (RENEW, 2001).

Before starting our evaluation, we have re-engineered the Java source code of the three graphical drawing editor applications with a help from the Rational Rose tool (Rational, 2004). In addition, we manually specified their use case diagrams and sequence diagrams by examining their documentation. Due to space limitations, we next give a quantitative description of the three applications and focus on describing the framework generated through FBDM.

4.1 – The Unified Applications

JOONE has a use case diagram composed of one actor and 14 use cases related by 2 *"extends"* relations, 5 *"includes"* relations and 6 associations. Its class diagram is composed of 242 classes interrelated by 132 inheritance relation, 35 associations and 48 aggregation. It has 15 sequence diagrams.

On the other hand, the JARP application has a use case diagram composed of one actor and 13 use cases related by 5 "extends" relations, 3 "includes" relations and 5 associations. Its class diagram is composed of 175 classes related by 123 inheritance relations, 33 associations and 24 aggregations. It has 12 sequence diagrams.

As for the application RENEW, it has a use case diagram composed of one actor and 21 use cases related by 3 "extends" relations, 11 "includes" relations and 5 associations. Its class diagram is composed of 328 classes related by 243 inheritance relations, 191 associations and 17 aggregations.

4.2 – The Generated Framework

Figure 3 shows the use case diagram of the framework generated by unifying the three applications in F-UMLTool. Similar to the unified applications, it has one core actor. In addition, it has 27 use cases marked as blackbox hot-spots (e.g., "create a neural network Figure" and "create a petri Net Figure"); these use cases were not present in all of the unified applications and, thus, represent particular application-specific functionalities. Their presence in the use case diagram gives the designer an idea on possible adaptations of the framework and guides him/her in the reuse. Furthermore, the generated use case diagram contains seven use cases marked as core (e.g., "create figure"); these latter represent the basic functionalities of any graphical drawing editor application, which also the case of the three unified applications.
Figure 4 partially shows the generated class diagram of the framework. The complete class diagram contains 220 classes, among which 38 are core, 4 are both core and whitebox, 76 are blackbox and 102 are whitebox classes.

Examining closely the generated class diagram, we found that among the classes present in the three applications, we have the classes Figure, AbstractFigure, AttributeFigure and CompositeFigure; thus, these classes are part of the framework core. In addition, among these classes, the classes AbstractFigure, AttributeFigure and CompositeFigure contain virtual methods; that is, they could be refined either via inheritance or by overriding some of their methods. For this, they are marked as both whitebox and core classes.

During the generation of the framework class diagram, F-UML Tool decided automatically on the type of the hot-spot classes (e.g., AttributeFigure, PolylineFigure are whitebox, PetriPlace and PetriTransition are blackbox). It prompted us to decide on the completeness of 16 inheritance and composition relations. Among these relations, we cite the inheritance relation between AttributeFigure and RectangleFigure, EllipseFigure, TextFigure; since the inheriting classes do not represent the entire domain, we have decided to tag this relation {*incomplete*}.

Figure 5 presents an example of the generated sequence diagrams, for the "Create a figure" use case. Some objects (e.g., PetriPlaceImpl and LayerFigure) are blackbox in conformance with the class diagram. The message LayerFigure is tagged optional since it appears only in one application among the three unified ones.

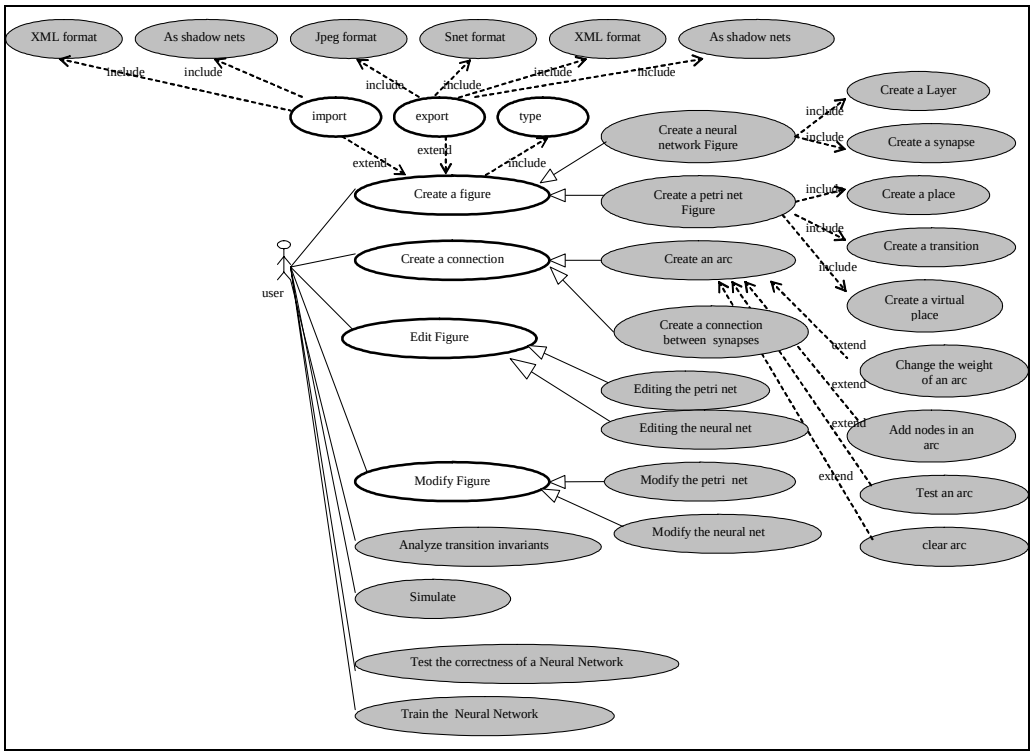


Figure 3: Use case diagram of the generated graphical drawing editor framework

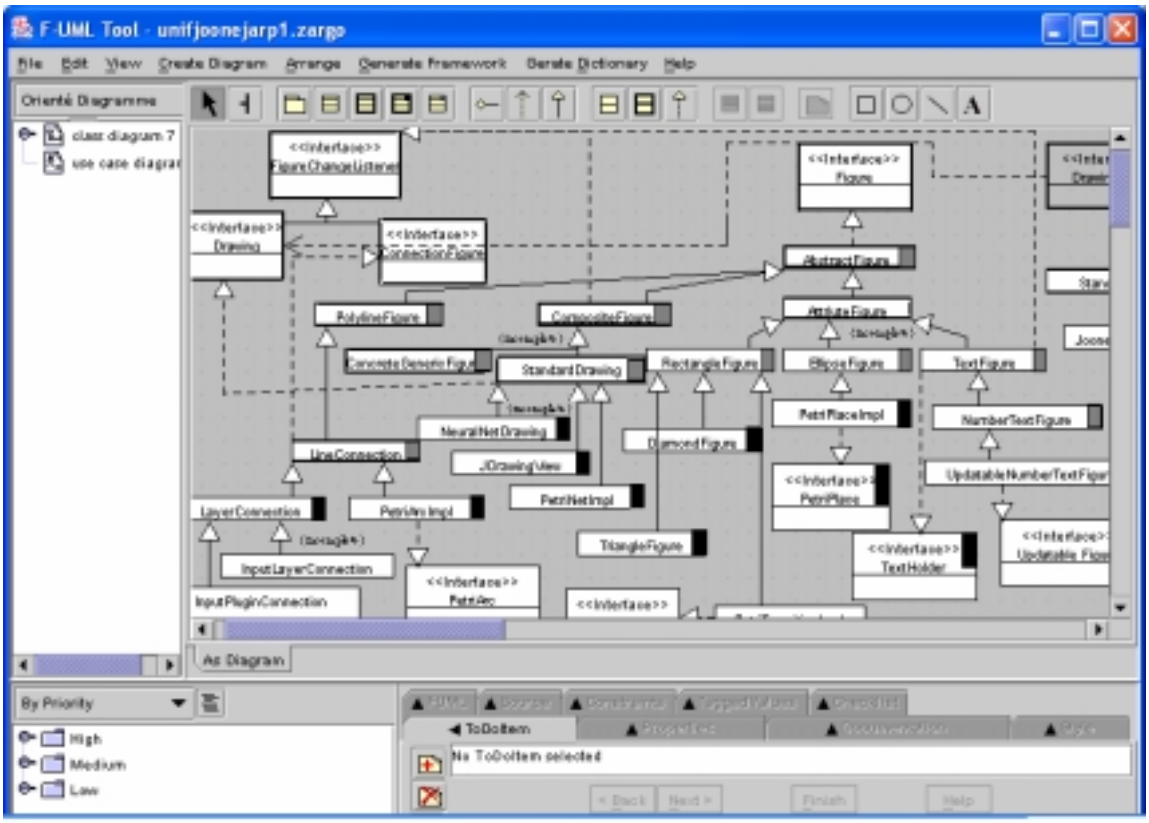


Figure 4: Class Diagram of the generated Graphical Drawing Editor Framework (partial view)

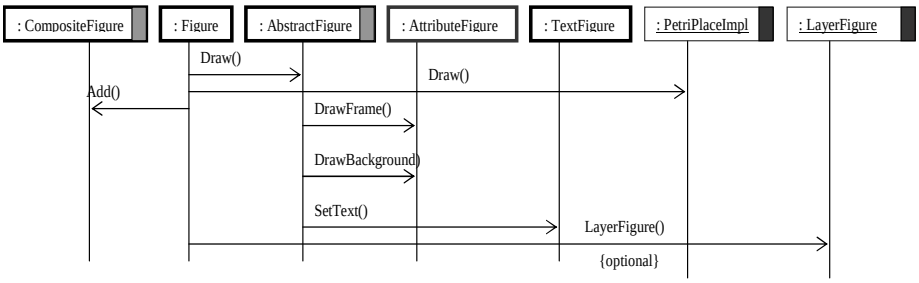


Figure 5: Generated Sequence diagram for "Create Figure"

5 - EVALUATION OF THE FBDM METHOD

The evaluation of FBDM is divided into two phases. In the first phase, the quality of the generated framework is examined with respect to the quality of the original applications; the quality is evaluated using several object-oriented design metrics. In the second phase, the generated framework is compared with the existing framework (JHotDraw) to examine its completeness, consistency and non-redundancy. This comparative evaluation allows us, in addition, to evaluate the usefulness of the F-UML notation.

5.1 - Evaluation Metrics

Many researchers have been interested in software metrics (c.f., (Kim,2002), (Shepperd, 1993)) and particularly in object-oriented specific metrics (c.f., (Chidamber, 1994), (Lorenz, 1994), (Xenos, 2000)). Xenos (2000) presents a state of the art of object-oriented metrics and classifies them essentially into three categories: class, method and inheritance metrics.

Kim (2002) adapted these object-oriented metrics for UML diagrams. In our evaluation, we have selected a set of most pertinent metrics for frameworks from the works of Xenos (2000) and Kim (2002). To these metrics, we added a set of framework specific metrics.

As illustrated in Table 4, the selected metrics for each diagram can be divided into two categories, depending on their usage. The first category of metrics can be used to measure the quality of a diagram in terms of its structure. The second category of metrics (marked with ^{R}) helps to identify the reuse degree of a diagram, e.g., the number of classes in the core, the number of methods that must be redefined when reusing a whitebox class, etc. We note that the metrics listed in Table 4 can be used to calculate other metrics, e.g., the number of hot-spot use cases (NHU) and the number of hot-spot actors (NHA) can be can be derived by subtracting NCU from NUM and NCA from NAM, respectively.

5.2 – An Intrinsic Evaluation

In this intrinsic evaluation of the generated framework, we aimed at testing the efficiency of the FBDM process. More precisely, we wanted to answer the following question: staring from "good" quality applications, does the FBDM process generate an "as good" quality framework? For this, we used the set of quality metrics to examine the generated framework.

Table 5 summarizes the values of the metrics used to answer this question. Overall, as for the class diagrams of the applications, similar results were confirmed for the quality of the generated framework class diagram since it is the largest, we next examine it more closely.

Number of classes

The generated class diagram has a fewer classes than the largest application (RENEW). The excluded classes are due to **Rule 6**, which uses the heuristic R_{dc} to decide whether a class covers the domain enough to be added to the generated framework. Recall that R_{dc} determines the importance of a class in the domain as a function of the number of appearances of the class in the unified applications. In other words, this heuristic lets us eliminate the classes that are too application specific. For instance, the class PetriEditor which belongs to the JARP application was omitted in the generated framework, since both R_{dc}(PetriEditor) < 2/3 and this class is far from a core class by more than two inheritance levels.

Number of relations

In addition, the number of relations (association, aggregation and inheritance) slightly increased in the generated class diagram due to **Rule 7** which automatically transfers all relations between classes added to the framework class diagram. For instance, according to the FBDM process, the DIT of the generated framework will be at most one plus the maximum of the DITs of the original applications. The additional depth level results from **Rule 3.d** and justifies the value of NIM for the generated framework. For example, consider the DIT of the class AbstractFigure: in the JOONE

application, it is equal to 4, in the JARP and RENEW applications it is equal to 3, while it is equal to 4 in the generated framework.

Similar results were confirmed for the quality of the generated use case diagram and sequence diagrams.

Metric		***Definition***
Class diagram	NCM	Number of classes in a model (Kim, 2002)
	NCC	Number of core classes[R]
	NBBC	Number of blackbox hot-spot classes[R]
	NWBC	Number of whitebox hot-spot classes[R]
	NWBCC	Number of whitebox core classes[R]
	NAsM	Number of associations in a model (Kim, 2002)
	NAgM	Number of aggregations in a model (Kim, 2002)
	NIM	Number of inheritance relations in a model (Kim, 2002)
	Class metrics	
	NAtC	Number of attributes in a class (Kim, 2002)
	NOpC	Number of operations in a class (Kim, 2002)
	NROC	Number of re-definable operations in a class[R]
	Inheritance metrics	
	DIT	Number of ancestors of a class (Chidamber,1994)
	NOC	Number of children of a class (Chidamber,1994)
Use case diagram metrics	NAM	Number of actors in a Model (Kim, 2002)
	NCA	Number of core actors[R]
	NUM	Number of use cases in a Model
	NCU	Number of core use cases[R]
Sequence diagram	NOM	Number of objects in a Model (Kim, 2002)
	NMM	Number of messages in a Model
	NOpM	Number of optional messages in a Model[R]

Table 4. Evaluation metrics used ([R]: reuse metric)

Metric	**Generated Framework**	**JOONE**	**JARP**	**RENEW**
NCM	220	242	175	328
NAsM	93	35	33	191
NAgM	8	48	24	17
NIM	231	132	123	243
NAtC	0⋯10	0⋯43	0 ⋯ 16	0 ⋯ 10
NOpC	1 ⋯ 58	1 ⋯ 80	1 ⋯ 31	1 ⋯ 103
DIT	0 ⋯ 17	0 ⋯ 8	0 ⋯ 15	0 ⋯ 17
NOC	0 ⋯ 3	0 ⋯ 5	0 ⋯ 3	0 ⋯ 3
NAM	1	1	1	1
NUM	34	14	13	21
NOM (“CreateFigure”)	7	7	4	6
NMM (“CreateFigure”)	8	6	4	5

Table 5. Values of quality metrics

Metric	**Result**
NCC	38
NBBC	76
NWBC	102
NWBCC	4
NROC	0⋯7
NCA	1
NCU	7
NOpM(“CreateFigure”)	2

Table 6. Values of reuse metrics

Overall, as for the class quality, the case study illustrates that the generated classes maintain the quality metric values below the largest values of the unified classes.

While there are no fixed thresholds for the above metrics, it is clear that their values reflect the ease of understanding a design and the benefits of reusing it. For example, classes with a large

number of attributes (NAtC) and methods (NOpC) would be of a higher complexity (Chidamber,1994). Another example is the DIT of a class, which measures the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number of methods it inherits and, thus, the more complex the class is. However, while deeper trees yield a greater design complexity, they offer better reuse levels through the multiple inherited methods.

A third example is the metric NOC, which both measures the number of a class children of and indicates its possible reuses: A large number of children implies a greater reuse potential. However, this measure alone may not convey a precise reuse measure. For instance, consider the NOC of the class *AbstractFigure*; it is equal to 3 both in JHotDraw and the generated framework, which indicates that *AbstractFigure* offers a medium potential of reuse. However, in the generated framework, *AbstractFigure* is the root of an *incomplete* hierarchy (see Figure 4), which indicates that the generated framework can be adapted to a particular application by adding other inheriting classes. Thus, with the F-UML notation, the number of incomplete relations stemming from a class can be combined with NOC for a better indication of the reuse measure.

5.3 – A Comparative Evaluation

In this evaluation, we aimed at testing the completeness and correctness of the generated framework, and the usefulness of the F-UML notation.

Consistency and completeness of the use case diagram

Comparing the use case diagram of the generated framework (Figure 3) with the manually designed use case diagram of JHotDraw, we noted that:

- The same main actor (user) is present in both frameworks.
- The 27 use cases representing application-specific functions (e.g., create a neural net figure, create an arc) are generated as hot-spots that illustrate possible adaptations of the framework.
- The set of use cases in the generated framework contains most of those of JHotDraw. More specifically, NUM is 9 in JHotDraw and it is 34 in the generated framework from which 27 are hot-spot use cases (i.e., shown only to give an idea on possible reuses) and 7 are core use cases (see Table 6). Thus, comparing JHotDraw with the generated framework, only two use cases ("create animation" and "use construction tools for figure") were not detected by the design process. Since they were not used by any of the three applications, these use cases do not represent essential functions of JHotDraw. In addition, relying on the F-UML notation, the missing use cases can be easily added as hot-spots.

Overall, this case study confirmed the completeness and correctness of the unification rules for the use case diagrams, since the JHotDraw use case diagram is contained in the generated use case diagram, modulo the hot-spots.

Consistency and completeness of the class diagram

Comparing the class diagram of the generated framework (Figure 4) with the class diagram of JHotDraw, we noted that:

- All of the abstract classes which define the generic structure and behavior of any application in the framework domain (e.g., Figure, AbstractFigure, CompositeFigure, FigureChangeListener, Drawing) are completely and correctly derived by the FBDM design process.
- The concrete classes that could be reused in specific applications (e.g., EllipseFigure, RectangleFigure) are generated as hot-spot classes.
- The NCM of JHotdraw is 216 while it is 220 in the generated class diagram. Looking closely at both diagrams, we found out that:
  - The additional classes are marked as hot-spots in the generated framework. This may lead us to conclude that the design process produces a framework with (possibly too many) application specific increments, e.g., PetriArc, PetriPlace, etc. These latter could be considered as details that may complicate the comprehension of a framework and hence impede its reuse. However, the F-UML notation helps by visually distinguishing these details from the core. Thus, when reusing a framework, the designer can first focus on the core, and later he/she can choose to understand or ignore the hot-spots.
  - Some classes in JHotDraw were not derived in the generated framework because they were absent in the original applications (e.g., *ImageFigure* in JHotDraw). However, the FBDM design process puts the tag {incomplete} wherever the designer can add the

missing classes. For instance, *RoundRectangleFigure* can be added as an inheriting class of *AttributeFigure* in the generated framework; this possibility is marked by the tag {incomplete} on the generalization out of AttributeFigure (see Figure 4). These missing classes may reduce the number of reused classes (as a reuse metric); however, thanks to the F-UML notation, the designer is advised of the places he is expected to focus his design effort.

- The large number of classes in the derived framework is justifiable since the original applications have an average NCM of 248 classes. In addition, a large number of the framework classes is a blackbox hot-spot (NBBC=76); that is, they give an idea about possible framework adaptations thus, one would only select some of these classes without having to modify them.
- The relations (generalization, association and aggregation) derived in the framework are consistent with their counter parts in JHotDraw. The generated framework contains additional relations taken with the application-specific classes.

Similar results were noted when comparing the sequence diagrams of the generated framework with the sequence diagrams of JHotDraw. The generated sequence diagram includes the corresponding sequence diagram in JHotDraw. In addition, it complements it with an optional message (*LayerFigure*()) relating the core object *Figure* and the hot-spot object *LayerFigure*.

Overall, the generated framework is consistent with and contains more details than JHotDraw; In addition, the degree of details produced in the generated framework depends on the level of domain coverage of the unified applications. F-UML helps identifying the details, which is essential both to measure the degree of reuse and to guide a reuse. However, it lacks concepts to express a conditional instantiation of two alternative hot-spots. For example, there is no indication in the class diagram of the framework of Figure 4 that the class *NeuralNetDrawing* cannot be taken with *PetriNetImpl*. This leads us to consider the use of OCL to express such constraints in future work. Finally, the F-UMLTool was vital in the design process, especially in managing the complexity of applying the rules on the large class diagrams.

Finally, we note that in addition to the JHotDraw experiment reported in this paper, we have evaluated the FBDM method in the case of a framework for e-commerce brokers (Bouassida et al., 2002). This latter case study contains all three diagrams. The framework was generated from three independent e-broker applications. Overall, the e-commerce broker experiment confirms the above reported results for the graphical editor domain. Unlike the JHotDraw case study, we do not have access to an e-broker framework to compare the generality and degree of reuse of the derived framework.

6 - CONCLUSION

This paper first presented the FBDM design method for frameworks. It then presented an experimental (intrinsic and comparative) evaluation of FBDM in the domain of graphical drawing editors.

On one hand, the case study showed that the F-UML notation facilitates the distinction between the core of the framework, which must be present in any application derived from it, and its variable parts. This, in turn, can guide the designer in estimating the degree of reuse the framework can offer. In addition, the case study highlighted the fact that the F-UML notation might need to be augmented with the Object Constraint Language (OCL) to express certain reuse constraints on the hot-spots.

On the other hand, the case study showed that the design process generates a framework that contains the whole framework core, certain hot-spots present in JHotDraw, and other application-specific hot-spots.

We are currently working on three research axes. The first examines how to add the generation of the pattern diagram to the F-UMLTool. The second consists of automating the collection of the semantic comparison criteria in the dictionary used by the F-UMLTool during the design process. Finally, our third research axis examines how F-UML can be integrated in the OMG Model Driven Architecture (OMG, 2001); in particular, we are examining how to define the PIM-PIM and the PIM-PDM transformations in terms of hot-spots. The integration of F-UML in MDA provides MDA with a reuse validation capability through the formal semantics of F-UML (Bouassida et al., 2003) (Bouassida et al., 2005).

BIBLIOGRAPHY

Aksit, M., Tekinerdogan, B., Marcelloni, F., Bergmans, L. (1999), "Deriving Object-Oriented Frameworks from Domain Knowledge", in *Building Application Frameworks: Object-Oriented*

Foundations of Framework Design, M. Fayad, D. Schmidt, R. Johnson (Eds.), John Wiley & Sons Inc., pp169-198.

Bouassida, N., Ben-Abdallah, H., Gargouri, F., Ben-Hamadou, A. (2004), "F-UMLTool for the formal design of frameworks", *XXII^{ème} Congrés INFORSID*, Biarritz-France 25-28 Mai.

Bouassida, N., Ben-Abdallah, H., Gargouri, F., Ben-Hamadou, A. (2002), "A stepwise Framework Design Process", *IEEE International Conference on Systems Man and Cybernetics*, Hammamet, Tunisia, 07-09 October.

Bouassida N., Ayadi, T., Ben-Abdallah, H., Gargouri, F. (2002), "Design of a framework for electronic commerce brokers", *IEEE International Conference on Cognitive Informatics*, Calgary, Canada, 27-29 August .

Bouassida, N., Ben-Abdallah H., Gargouri F., Ben Hamadou A. (2003), "Formalizing the framework design language F-UML", *International conference on Software Engineering and Formal methods (SEFM'03)*, Brisbane-Australia.

Bouassida, N., Ben-Abdallah, H., Gargouri, F., Ben-Hamadou, A. (2005), "Towards a rigorous architectural reuse", *Arab International conference on computers Software and Applications*, Egypt.

Chidamber, S. R., Kemerer, C. F. (1994), "A metrics suite for object-oriented design", *IEEE Transactions on Software Engineering*, Vol 20, N° 6, pp 476-493.

Erni, K., Lewrentz, C. (1996), "Applying design metrics to object-oriented frameworks", *Proceedings of the 3rd International Software Metrics Symposium* (Metrics'96).

Fayad, M., Schmidt, D., Johnson, R. (1998), *Building Application Frameworks*, Wiley.

Fontoura, M.F., Pree W., Rumpe B. (2000), "UML-F: A Modeling Language for Object-Oriented Frameworks", *European Conference on Object Oriented Programming*, Springer-Verlag.

Fontoura, M. F., Crespo S., Lucena C.J., Alencar P., Cowan D. (2000), "Using viewpoints to derive Object-Oriented Frameworks: A case study in the web education domain", *The Journal of Systems and Software (JSS)*, vol 54, n°3 Elsevier Science.

Gamma, E., R., Helm, R. Johnson, Vlissides, J. (1995), *Design patterns: Elements of reusable Object Oriented Software*, Addisson-Wesley, Reading, MA.

Gamma, E., Eggenschwiler, T. (2000), http://www.jhotdraw.org

Johnson, R. E., Foote, B. (1998), "Designing reusable classes", *Journal of Object Oriented Programming*, vol. 1, n°2.

Kim, H., Boldyreff C. (2002), "Developing Software Metrics Applicable to UML Models", Workshop *QAOOSE, Malaga-Spain.*

Koskimies, K., Mossenback, H. (1995), "Designing a framework by stepwise generalization", *5th European software Engineering Conference,. Lecture Notes in Computer Science 989*, Springer-Verlag.

Lorenz, M., Kidd, J. (1994), *Object-Oriented Software Metrics: A Practical Approach*, Prentice Hall.

Meyer, B., (1988) *Object Oriented software construction*, Edition Prentice-Hall International.

OMG. Model-Driven Architecture Home Page, http://www.omg.org/mda (2001).

Pree, W. (1994), "Meta-patterns: a means for capturing the essentials of object-oriented designs", *European Conference on Object Oriented Programming*, Bologna, Italy.

Rational, www.rational.com, 2004.

Riehle, D., Gross, T. (1988) “Role model based framework design and integration”, Proceedings of OOPSLA’98, Vancouver.

Reenskaug, T., (1996), *Working with objects*, Greenwich : Manning.

Roberts, D., Johnson, R. (1996), “Evolving Frameworks : A pattern language for Developing Object Oriented Frameworks”, *Proccedings of the third conference on pattern languages and programming,* Montecilio, Illinois.

Rumbaugh, J., (1991), *Object Oriented Modelling and design*, Prentice Hall.

Sanada, Y., Adams, R. (2002), “Representing Design Patterns and Frameworks in UML-Towards a Comprehensive Approach”, *Journal of Object Technology*, vol. 1, n°2, July-August.

Shepperd, M.J., Ince, D. (1993), *Derivation and Validation of software Metrics*, Clarendon Press, Oxford, UK.

JARP (2001), Petri Nets Analyzer: JARP, SourceForge, http://www.jarp.org,.

JOONE (2001), Java Object Oriented Neural Engine: Joone, SourceForge, http://www.joone.org,.

Schmid, H. A. (1997), “Systematic framework design by generalization”, *Communications of the ACM, Special issue on Object Oriented Application frameworks*, Vol 40, N°10.

Smith, G. (2000), *The object-Z specification Language*, Advances in Formal methods, Kluwer Academic Publishers.

Szyperski, C., Pfister, C. (1996), Workshop on Component Oriented Programming in Mülhaüser M. (editions), *Special issue on Object Oriented Programming, ECOOP (96),* Vrelag, Heideberg.

Renew, (2001) Wienberg, F., Kummer, O., Duvigneau M., http://www.renew.org.

Xenos, M., Starvrinoudis, D., Zikouli, K., Christoudalis, D. (2000), “Object-Oriented Metrics- A Survey”, *Proceedings of the Federation of European Software Measurement Associations*, Madrid, Spain.

MODELISATION DE LA REPARTITION DES DONNEES D'UN DATA WAREHOUSE

Karima TEKAYA
Assistante en informatique
Karima.Tekaya@.isi.rnu.tn

Abdelaziz ABDELLATIF
Maître-assistant en Informatique
abdelaziz.abdellatif@fst.rnu.tn

Adresse professionnelle
Faculté des sciences de Tunis, Département informatique,
Campus universitaire - 2092 Manar II

Résumé : Les utilisateurs des data warehouses ne cessent d'augmenter. A l'image des entreprises, ces utilisateurs sont de plus en plus répartis géographiquement sur plusieurs sites. Les data warehouses centralisés ne sont donc plus adaptés à ce genre d'entreprises. Pour répondre à ce nouveau besoin, nous avons proposé une démarche de modélisation de la répartition des données d'un Data Warehouse. Celle-ci, se base essentiellement sur un ensemble de matrices permettant la modélisation de l'intégration logique des données du Data Warehouse d'un côté et leur répartition entre les différents Data Marts de l'organisation d'un autre côté.

Summary: The users of Data Warehouses do not cease increasing. With the image of the companies, these users are divided more and more geographically on several sites. Centralized Data Warehouses thus are not adapted more to this kind of companies. To meet this new requirement, we proposed a methodology of modelling the distribution of the data of a Data Warehouse. This one is based primarily on a set of matrices allowing the modelling of the integration of the data in a Data Warehouse. Secondly, their distribution between different Data Marts.

Mots clés : Data warehouse, Data mart, Modélisation, Répartition, Intégration.

1- INTRODUCTION

Un Data Warehouse (DW) répond aux problèmes de données surabondantes et localisées sur de multiples systèmes hétérogènes. Le DW est un entrepôt de données permettant un stockage intermédiaire des données issues des applications de production, dans lesquelles les utilisateurs finaux puisent avec des outils de restitution et d'analyse.

L'intégration du DW dans une structure unique a pour but d'éviter aux données concernées par plusieurs sujets d'être dupliquées. Le DW est fragmenté en plusieurs bases appelées Data Mart(DM). Un Data Mart est l'implémentation d'un DW pour un domaine bien spécifique. En effet, c'est un sous ensemble d'un DW [1].

On peut avoir plusieurs Data Mart au sein d'une même entreprise [2]. Ces data marts peuvent être répartis par département, les données utilisées sont extraites à partir du DW principal (centralisé).

2- PROBLEMATIQUE

Un système d'information est composé d'une composante décisionnelle et d'une composante opérationnelle. Le système d'information opérationnel englobe toutes les informations concernant l'activité de l'entreprise, ces données sont stockées dans une base appelée base de production.

Le système d'information décisionnel englobe des informations provenant de bases de production ou de sources diverses et externes à l'entreprise servant comme support d'aide à la décision. L'ensemble de ces informations est stocké dans le DW.

Le système d'information est en évolution, il fait face aujourd'hui aux problèmes de décentralisation des entreprises, les utilisateurs sont de plus en plus nombreux, ils exercent des activités hétérogènes et appartiennent généralement à des sites éloignés géographiquement. Ceci a eu comme conséquence la décentralisation du système décisionnel.

Les besoins informationnels et les utilisations des données peuvent être différentes d'un site à un autre. De ce fait, une organisation centralisée des données peut être non adéquate à cette nouvelle architecture répartie. Un DW réparti pourra répondre plus efficacement aux besoins des utilisateurs. Les données peuvent être organisées par sujet et une meilleure utilisation du DW est garantie. La répartition d'un DW en plusieurs DM est la solution la plus adéquate pour un système différé puisqu'elle permet de rapprocher les données aux utilisateurs et améliorer l'organisation des données.

Plusieurs contraintes techniques peuvent être rajoutées :

- La communication des informations stratégiques aux différents décideurs s'avère de plus en plus coûteuse de point de vu financier (coût des accès) et temporel (temps d'accès).
- Le DW est centralisé dans une base unique, le stockage des données sur un ordinateur central peut souffrir d'une très longue charge de traitement ce qui peut influer sur sa performance.
- En plus, le volume du DW augmente très rapidement ce qui ralentit les accès et gonfle le stockage [7], [8] et [9].
- D'autre part, la centralisation des données pourrait devenir le point sensible du système informatique.

De ces faits, la centralisation d'un DW peut se refléter négativement sur sa performance et ses fins. Pour faire face à ces différents problèmes, le système opérationnel opte pour l'adaptation des bases de données réparties. Le système d'information décisionnel opte pour la répartition du DW en DM. Plusieurs démarches de modélisation ont été proposées pour modéliser les bases de production réparties. Par contre, aucune

démarche exhaustive n'a été proposée pour la modélisation de la répartition des données d'un DW.

3- CONTRIBUTION

La contribution apportée par cet article est de proposer une démarche de modélisation de la répartition des données d'un DW. Celle-ci se base essentiellement sur les niveaux de modélisation classiques, en ajoutant un ensemble de concepts de base, intégrer de nouveaux modèles et proposer un formalisme de présentation.

Dans la section suivante, nous allons citer l'état de l'art. Dans la section 5, nous allons proposer les concepts de base de notre démarche, les modèles nécessaires et le formalisme proposé.

4- ETAT DE L'ART

Les méthodologies trouvées dans la littérature ont généralement pour objectif d'intégrer le DW dans une structure unique et ont comme résultat un entrepôt de données centralisé [3] et [4]. Cet entrepôt est appelé DW, s'il est généralisé aux activités de l'entreprise, ou bien DM s'il est spécifique à un département particulier.

On a constaté dans l'état de l'art que tous les travaux concernés par la modélisation de la réparation des données des DW sont orientés vers la modélisation physique [5] et [6]. Des algorithmes de répartition verticale des données ont été proposées dans [14] et [15]. L'idée de répartition des données d'un DW a été évoquée par Noaman, A.Y. et K. Barker dans [7] et [8]. Ils se sont basés sur l'architecture ANSI/SPARC pour la modélisation des données des DW. La démarche proposée par ces auteurs se base essentiellement sur l'approche Top/Down. Ils ont aussi développé un algorithme de fragmentation horizontale des tables de faits dans [9].

Dans [4] une démarche exhaustive a été proposée pour modéliser l'intégration des données d'un DW (Figure 1). Celle-ci se base essentiellement sur l'ajout d'un modèle d'intégration des données permettant de modéliser l'intégration des données (MID) dans le DW. Ce modèle sert à identifier pour les données du modèle logique de données obtenu : leurs sources de données, les transformations éventuelles qu'elles doivent subir, leurs modes de rafraîchissement dans le DW et leurs fréquences d'utilisation. Dans [12] une démarche exhaustive de modélisation de la répartition des données d'une base de production a été bien développée (Figure1). Celle-ci se base essentiellement sur l'ajout d'un modèle de répartition des données (MRD) en tenant compte d'un processus de répartition et en intégrant un programme d'optimisation des différentes allocations en fonction des débits binaires échangés, les fiabilités des échanges et les caractéristiques du réseau. Dans [16], une adaptation du modèle ASM (Abstract State Machines) a été éffectuée pour modéliser un data warehouse réparti.

5- SOLUTION PROPOSEE

5.1- Concepts de base

Nous visons par cette démarche le côté logique et organisationnel des données qui n'a pas été bien mis en évidence dans l'état de l'art. L'objectif visé est, donc, de modéliser les données contenues dans un DW central et en même temps leur répartition entre plusieurs bases de données distantes qui seront les futurs DM de l'entreprise.

Pour généraliser notre démarche, nous proposons un formalisme que nous pourrons adapter à n'importe quelle approche de conception. Généralement la modélisation d'un système d'information se base sur trois niveaux :

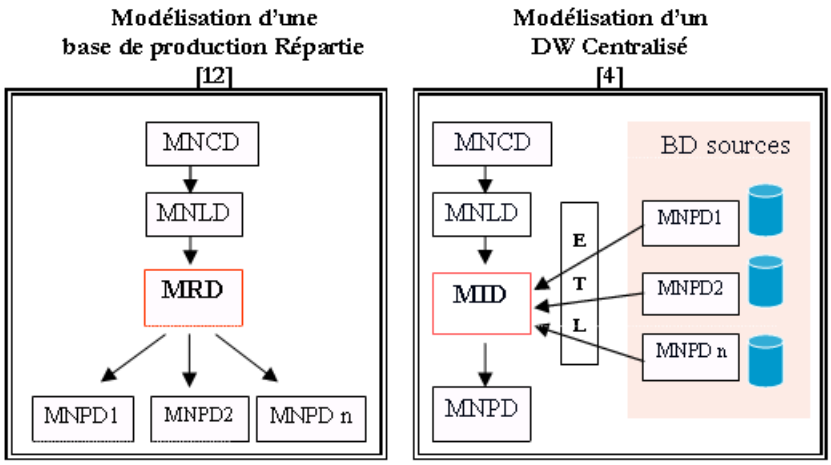


Figure 1 : Démarches de modélisation (Etat de l'art)

1- Modélisation du Niveau Conceptuel des Données (MNCD)
2- Modélisation du Niveau Logique des Données (MNLD)
3- Modélisation du Niveau Physique des Données (MNPD)

En effet, nous allons nous baser sur les deux démarches proposées dans [4] et [12]. Le but est l'adaptation des processus de répartition à la modélisation d'un DW.

Une fois la modélisation du niveau conceptuel des données (MNCD) est réalisée, on entame la modélisation du niveau logique des données, celle-ci peut s'effectuer en deux axes. Deux besoins fondamentaux sont à satisfaire : (1) Il faut tenir compte tout d'abord des besoins d'intégration des données modélisées dans le DW vis-à-vis de leurs sources de données. Elles peuvent subir des transformations pour leur adaptation à la base. (2) Un autre besoin est à satisfaire, c'est celui de la répartition des données entre les différents sites, ceci en tenant compte des débits binaires échangés, des fiabilités des échanges et des besoins de fragmentation des tables. Ces deux axes de modélisation sont indépendants et peuvent être effectués en parallèle. Ainsi, deux équipes de modélisation peuvent travailler en même temps. Une première équipe qui se charge de la répartition des données entre les différents DM et une deuxième équipe qui se charge de l'intégration logique des données sources du DW global. Cette méthode de modélisation permet de garantir un espace de travail partagé, accélérer le rythme du travail et réduire la complexité de la modélisation. Les deux équipes peuvent se réunir ensuite, pour une organisation finale des données intégrées et allouées.

5.2- Modèles

Cette méthodologie se base sur six modèles (Figure 2) répartis selon trois niveaux : niveau conceptuel, niveau logique (enrichi) et niveau physique.

Pour choisir l'architecture à mettre en place, nous proposons un modèle introductif appelé Modèle Structurel et Organisationnel de l'Entreprise (MSOE).

Au niveau conceptuel nous gardons le Modèle Conceptuel de Données (MCD) proposé dans les approches classiques.

Au niveau logique, Le modèle logique de Données (MLD) sera généré à partir du MCD.

A ce niveau, nous proposons un enrichissement à travers deux modèles : Un Modèle d'Intégration Logique des Données (MILD) et un Modèle de Répartition Logique des Données (MRLD).

Le MILD permet d'identifier pour chaque donnée du MNLD, la source correspondante et (si nécessaire) les transformations qu'elle doit subir.

Le MRLD permet d'identifier pour chaque donnée du MNLD global le DM auquel elle sera affectée.

Le MILD et le MRLD seront fusionnés pour créer un dernier modèle englobant toutes les informations nécessaires pour la modélisation de la répartition des données d'un DW. Celui-ci est appelé Modèle d'Intégration Logique des Données Réparties (MILDR).

Au niveau physique, plusieurs Modèles Physiques de Données (MPD) seront déduits. Ces derniers, représentent l'organisation physique des différents DM.

5.3- FORMALISME PROPOSE

5.3.1- Le MSOE

Pour établir le MSOE, nous proposons le formalisme suivant :

– Un Tableau de Structure (TS)

– Une matrice de liaison inter-site (MLIS)

La MLIS permet de décrire la structure de l'entreprise et son organisation de point de vue géographique. Cette description est nécessaire pour choisir la meilleure architecture à mettre en place.

Pour élaborer le MSOE, nous allons commencer tout d'abord par analyser la structure de l'entreprise. Ceci revient à trouver des réponses aux questions suivantes :

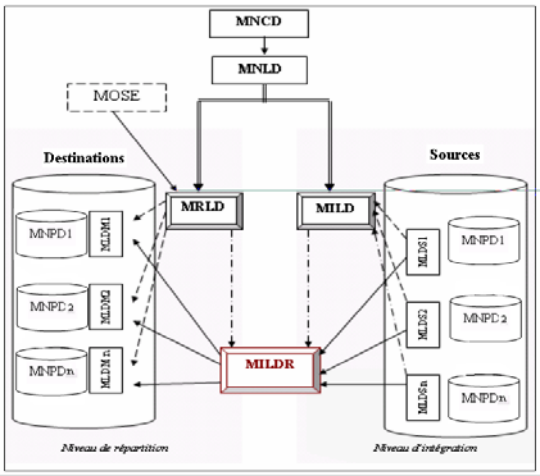


Figure 2 : Modèles proposés

1- L'entreprise est elle répartie géographiquement ?
2- Si oui, quels sont les sites qui la forment ?
3- Comment ces sites sont reliés les uns aux autres ?
4- Quels sont les moyens et les caractéristiques des supports de communication entre les sites (type de réseau, support, protocole, débit binaire de transmission des données, fiabilité…etc.)?

Les réponses à ces questions sont résumées dans le TS (Figure 3). Le tableau est une simple description de la structure de l'entreprise. Il permet de visualiser la liste des groupes de sites de l'entreprise, triée par ordre de priorité décisionnelle. Pour chaque site, on identifie son type et le groupe auquel il appartient. Le tableau de structure est

important pour les étapes suivantes puisqu'il détermine la liste des sites décisionnels de l'entreprise et peut faire l'objet d'une documentation pour l'entreprise. Les sites de priorité 3 seront supprimés de la liste puisqu'ils ne détiennent aucun pouvoir décisionnel.

Priorité (1, 2,3)	Groupes de sites	Types de sites	Liste des sites
1	G1	T1	$S_{1.1},...,S_{1.n}$
2	G2	T2	$S_{2.1},...,S_{2.k}$
3	G3	T3	$S_{3.1},...,S_{3.j}$

Figure 3 : Tableau De Structure

Après avoir décrit la structuration de l'entreprise, il est important d'étudier son infrastructure réseau. Cette étude consiste à identifier le type du réseau, les moyens et les caractéristiques des supports de communication entre les sites. En effet, il s'agit d'identifier pour chaque couple de sites s'il existe une portion du réseau qui les relie. Si cette portion existe, il faut se renseigner sur les caractéristiques des communications entre ces deux sites. Les caractéristiques qui nous semblent les plus importantes à identifier sont la fiabilité et le débit binaire.

Ces deux caractéristiques diffèrent d'une portion de réseau à une autre. En effet, ceci dépend des supports de transmission au sein du réseau, de la distance qui sépare les deux sites, ...etc. Des statistiques sont utilisées afin de déterminer les valeurs de ces caractéristiques.

Pour la présentation de ces caractéristiques nous pouvons utiliser une matrice carrée d'ordre n, où n est le nombre de sites. Cette matrice résume les liaisons entre les différents sites de l'entreprise. Chaque cellule de cette matrice contient le débit binaire échangé et la fiabilité de la portion du réseau liant les deux sites correspondants. Elle représente donc, l'existence d'une liaison entre deux sites quelconques. Une cellule vide indique l'absence de liaison entre les deux sites. Nous pouvons résumer ces différentes données dans la matrice qu'on a choisi d'appeler Matrice des Liaisons Inter-Sites1 (MLIS1) (Figure 4).

	S1	S2	…	Sn
S1		DB	DB	DB
		F	F	F
S2	DB		DB	DB
	F		F	F
…	DB	DB		DB
	F	F		F
Sn	DB	DB	DB	
	F	F	F	

Figure 4 : La MLIS1

Cette matrice peut être améliorée (Figure 5) par l'ajout des indicateurs caractérisant les débits binaires.

Un débit binaire (DB) peut être : un débit élevé (DE), un débit moyen (DMoy), un débit faible (DF).

Cette classification est basée sur la définition de trois intervalles de débit binaire. Ensuite, suivant l'appartenance du DB à un intervalle parmi ces trois, un

débit peut être classé : (DE), (DMoy) ou bien (DF).

Les modèles logiques des Data Marts décrivent l'allocation logique des différentes données du DW vers les sites correspondants. Cette description ne prend pas en considération les besoins d'intégration des données vis-à-vis de leurs sources. Les informations données par le MSOE sont insuffisantes pour décider l'allocation d'une information vers un site donné. Plusieurs critères sont à prendre en considération pour la répartition des données. Le critère le plus important est celui de la fréquence d'utilisation. On rappelle à cet effet, que les données du DW sont utilisées seulement en consultation.

De ce fait, il faut tout d'abord ressortir les différents traitements possibles qui seront exécutés par les sites de l'entreprise. Pour chaque site, on va énoncer les utilisations possibles des différentes données par les traitements. Les données sont des tables ou bien fragments de tables.

	S1	S2	…	Sn
S 1		DE/DMoy/DF	DE/DMoy/DF	DE/DMoy/DF
		F	F	F
S 2	DE/DMoy/DF		DE/DM/DF	DE/DM/DF
	F		F	F
…	DE/DMoy/DF	DE/DM/DF		DE/DM/DF
	F	F		F
S n	DE/DMoy/DF	DE/DMoy/DF	DE/DMoy/DF	
	F	F	F	

Figure 5 : MLIS2

5.3.2- Le Modèle de Répartition Logique des Données

Pour modéliser la répartition des données d'un DW, la première tâche consiste à identifier pour chaque site mentionné dans le MSOE, les traitements possibles sur les données de la base. Ceci, est visualisé dans une matrice tridimensionnelle car elle englobe les sites de l'entreprise, les traitements à effectuer par site et les données nécessaires. Cette matrice est appelée : « Matrice Utilisation des Données (MUD) ».

La deuxième tâche consiste à déduire à partir de la matrice précédente, les meilleures allocations possibles des données aux sites de l'entreprise. Et ce, par la construction d'une deuxième matrice visualisant le mode d'allocation des données. Cette matrice est appelée : « Matrice d'Allocation des Données (MAD) ».

LA MUD

La matrice utilisation des données décrit les différentes utilisations possibles des données par les différents sites décisionnels. Pour établir la MUD nous allons identifier tout d'abord les traitements possibles par site. Il est à noter qu'une donnée D_u peut être soit une table ou bien une portion de table, c'est à dire un fragment de table. Les tables sont extraites directement du MLD, par contre, les fragments de tables ne sont pas facilement identifiables. Pour ce faire, nous allons adapter le formalisme de fragmentation des données de production aux tables du MNLD. Le point de départ est la liste des tables du MLD et les différents traitements ti classés par site. Le résultat est l'ensemble des fragments nécessaires pour les différentes utilisations. Pour ce faire, nous proposons un formalisme qu'on a choisi d'appeler Matrice de fragmentation (MF).

Cette matrice a pour objectif l'identification des critères de fragmentation et les fragments nécessaires.

Elle est tridimensionnelle :

- une dimension pour les tables d'origine (T),
- une dimension pour les sites (S_i),
- une troisième pour les traitements susceptibles d'être exécutés sur chaque site (T_i).

Il existe deux types de fragments, un fragment vertical et un fragment horizontal : Le fragment vertical est une sélection d'une colonne d'une table. Le fragment horizontal est une sélection d'une ligne d'une table.

Pour établir la MF, il est indispensable d'identifier les différents fragments horizontaux et verticaux. De ce fait, nous proposons deux matrices préliminaires que nous avons choisis d'appeler Matrice de Fragmentation Horizontale (MFH) et la Matrice de Fragmentation Verticale et Mixte (MFVM).

La matrice d'utilisation des données (Figure 6) décrit pour chaque traitement correspondant à un site de l'entreprise, les données nécessaires pour son accomplissement. C'est une matrice tridimensionnelle par ce qu'elle intègre les dimensions suivantes:

- La liste des sites (S_i)
- La liste des traitements par site (T_i)
- La liste des données nécessaires aux traitements (D_u)

Dans la matrice utilisation des données, nous désignons par D_u:

- une table non fragmentée,
- un fragment horizontal,
- un fragment vertical,

– ou bien, un fragment mixte.

La construction de cette matrice consiste à identifier, pour chaque traitement t_ip_i, les données nécessaires, leur mode et leurs fréquences de consultation par ce traitement. Les opérations de création, de suppression, de modification ne seront pas prises en compte. Ces opérations seront faites par l'administrateur du DW qui s'occupe lui même de toutes les opérations de mise à jour. Chaque case de la matrice indique la fréquence d'utilisation de D_u vis à vis de tipi appartenant à S_i.

La matrice d'utilisation des données peut être simplifiée (Figure 7) en indiquant le total des utilisations par site. Ainsi, nous pouvons pour chaque donnée identifier le site le plus prioritaire, en tenant compte du nombre d'utilisations de celle-ci. La simplification aboutit à une deuxième matrice qu'on a choisi d'appeler MUD2.

Cette matrice nous servira de support pour décider l'allocation des données par site. Nous avons choisi d'enrichir cette matrice par les indicateurs de priorité pour chaque site. Chaque cellule contiendra le total des fréquences d'utilisation par site divisé par la priorité de ce dernier.

Le résultat final est un indicateur efficace pour décrire la nécessité ou non d'allocation de la donnée au site correspondant.

A ce niveau, nous disposons d'une liste de tables, de FH, de FV et de FM, nous avons aussi la fréquence d'utilisation de ces données par site ainsi que leurs priorités. Nous pouvons alors, procéder à la construction de la MAD.

LA MAD

La matrice d'allocation des données (Figure 8) décrit pour chaque donnée, le site dans lequel elle sera allouée ou bien, elle sera consultée. C'est une matrice bidimensionnelle parce qu'elle englobe : Les données utilisées (D_u), Les sites destinataires (S_i).

			TABLES DE FAITS / FRAGMENTS HORIZONTAUX				
			D_1	**…**	**D_u**	**…**	**D_t**
SITES CIBLES	**S1**	$t_{1.1}$	FU		FU		FU
		…					
		$t_{1.p1}$	FU		FU		FU
		…					
		$t_{1.q1}$	FU		FU		FU
	Total des utilisations		$TU_{1.1}$		$TU_{u.1}$		$TU_{t.1}$
	...						
	Si	$t_{1.i}$	FU		FU		FU
		…					
		$t_{i.pi}$	FU		FU		FU
		…					
		$t_{i.qi}$	FU		FU		FU
	...						
	Total des utilisations		$TU_{1.i}$		$TU_{u.i}$		$TU_{t.i}$
	…						
	Sn	$t_{n.1}$					x
		…					
		$t_{n.pn}$	x		x		
		$t_{n.qn}$			x		
	Total des utilisations		$TU_{1.n}$		$TU_{u.n}$		$TU_{t.n}$

Figure 6 : MUD1

Chaque cellule de la MUD2 représente un indicateur du besoin d'allocation. Si cet indicateur est faible, ceci signifie la non nécessité d'allocation de la donnée correspondante.

		TABLES DE FAITS / FRAGMENTS HORIZONTAUX				
		D_1	**…**	**D_u**	**…**	**D_t**
SITES CIBLES	**S_1**	TU_{11}		TU_{u1i}		TU_{t1i}
		P_1		P_1		P_1
	...					
	S_i	$TU_{1.i}$		$TU_{u.i}$		$TU_{t.i}$
		P_i		P_i		P_i
	...					
	S_n	$TU_{1.n}$		$TU_{u.n}$		$TU_{t.n}$
		P_n		P_n		P_n

Figure 7 : MUD2

Pour allouer une donnée on cherche le site qui l'utilise le plus c'est à dire dont le total des fréquences d'utilisation est supérieur à tous les autres sites et qui est le plus prioritaire pour son utilisation. Cette méthode d'allocation peut faire l'objet d'automatisation. Ainsi, on pourra décrire l'algorithme correspondant à l'allocation des données du DW vers les différents Data Marts de l'organisation.

Chaque cellule de la matrice d'allocation des données indique si une donnée est une Donnée Persistante (DP) ou bien Donnée Consultée (DC). Une DP signifie qu'elle est allouée au site correspondant, par contre, une DC veut dire qu'elle sera juste consultée par le site correspondant. Une cellule vide indique que la donnée n'est pas consultée par le site et on l'appelle Donnée Absente (DA).

Ayant réalisé le MALD, la modélisation de la répartition logique des Data Mart est achevée. Toutes les données correspondantes à la répartition des données sont identifiées, ces données nous permettront en partie de construire le MILDR. Mais, il faudra tenir compte en parallèle de l'intégration des données sources vis à vis des sources de données.

		TABLES DE FAITS / FRAGMENTX HORIZONTAUX				
		D_1	…	D_u	…	D_t
DESTINATIONS	DM1	DP/DC/DA		DP/DC/DA		DP/DC/DA
	...					
	DMi	DP/DC/DA		DP/DC/DA		DP/DC/DA
	...					
	DMn	DP/DC/DA		DP/DC/DA		DP/DC/DA

Figure 8 : MALD1

5.3.3- Modélisation de l'Intégration Logique des Données Sources

Le modèle d'intégration logique des données sources décrit les sources de données nécessaires pour les besoins d'intégration (Figure 9).

Chaque donnée du MLD est caractérisée par une source correspondante. Elle subit des transformations selon les besoins.

La modélisation de l'intégration logique des données sources n'intègre en aucun cas les besoins de répartition physique. Le formalisme proposé est une Matrice d'Intégration Logique des Données Sources (MILDS) (Figure 9).

Il s'agit de déterminer pour chaque donnée du MLD la source de donnée qui permet de l'alimenter. Cette dernière subit les transformations nécessaires pour l'adapter à la base.

Une source peut être soit :

- *Interne* : c'est l'ensemble des attributs qui se trouvent dans les tables sources des applications fonctionnelles.
- *Externe* : c'est l'ensemble des attributs spécifiques au DW comme les dates, les catégories, les types,…etc., qui ne proviennent pas des données sources.

Les données peuvent subir plusieurs. Une transformation peut faire l'objet :

- D'une transformation élémentaire (TE): formule, expression ou des programmes permettant d'obtenir le contenu d'un attribut (a) à partir d'une source (Sc). Ce type de transformation est fait dans le cas ou l'attribut est obtenu à partir d'une seule source.
- Une transformation composite (TC) : formule, expression ou programme permettant d'obtenir le contenu d'un attribut à partir de deux ou plusieurs sources.

Au niveau de la phase d'intégration, tout attribut de la base doit être caractérisé par la source qui l'alimente et par les transformations nécessaires qu'il doit subir pour son utilisation par les différents sites correspondants. Pour identifier ces transformations nous proposons une matrice que nous avons choisi d'appeler Matrice de Transformations des Données Sources (MTDS).

Celle-ci est tridimensionnelle car elle renferme les dimensions suivantes :

- une dimension pour les attributs des différentes tables du MLD (A)
- une dimension pour les sources internes (SI)
- une dimension pour les sources externes (SE)

Une fois la modélisation terminée, l'équipe de modélisation de la répartition fournit la MAD. La deuxième équipe, s'occupant de la modélisation logique des données sources, fournit la MILDS. Ces deux équipes, peuvent ensuite se réunir pour préparer la modélisation de l'intégration logique des données réparties.

5.3.4- Modélisation de l'Intégration Logique des Données Réparties (MILDR)

La modélisation de l'intégration logique des données réparties (Figure 10) consiste à caractériser chaque donnée allouée à un DM par :

- la source correspondante pour son alimentation (interne ou externe)
- la transformation nécessaire que la donnée source peut subir (élémentaire ou composite) pour son adaptation à la base.
- le DM auquel elle sera allouée.

Tables du MNLD	**Attributs**	**Sources de données**																**TC**
		MLDS1					**MLDS r**						**MLDS s**					
		A_{11}		**A_{1z}**		**A_{1x}**		**A_{r1}**		**A_{rz}**		**A_{rx}**		**A_{s1}**		**A_{sz}**		**A_{sx}**	
TD1	**a_{11}**	TE						TE						TE					
	…																		
	a_{l1}	x				x		x						x					**TC**
	...																		
	a_{1m}																		
TDj	**a_{j1}**			TE						TE						TE			
	…																		
	a_{j1}																		
	…																		
	a_{jm}																		
TDk	**a_{k1}**	x		x				x		x				x		x			**TC**
	…																		
	a_{kl}					TE						TE						TE	
	…			x		x				x		x				x		x	**TC**
	a_{km}																		

Figure 9: MILDS

Le formalisme proposé est une matrice appelée Matrice d'Intégration Logique des Données Réparties (MILDR). La modélisation de l'intégration logique des données réparties consiste à fusionner les deux matrices réalisées au niveau logique (la MILDS et la MRLD). La fusion consiste à remplacer les colonnes de la MILDS par les données réparties entre les différents Data Mart de l'entreprise, ces données sont celles identifiées dans la matrice allocation logique des données. On gardera par contre les mêmes colonnes de la matrice transformation des données. Le résultat de la fusion sera une nouvelle matrice englobant toutes les informations nécessaires pour la répartition des données d'un DW entre plusieurs DM.

La MILDR représente le dernier niveau de modélisation logique, elle garde pour chaque donnée D_u sa traçabilité vis à vis de sa source, les différentes transformations nécessaires pour son adaptation à la base, le type de la transformation voulu et le site correspondant auquel elle sera allouée.

Cette matrice peut subir des modifications selon le besoin, ceci va simplifier les mises à jour et renforcer la flexibilité de la modélisation surtout avec la fluctuation de l'environnement et avec l'extension du besoin informationnel vis à vis du DW.

Destinations \ Sources			$MLDS_1$			$MLDS_r$			$MLDS_s$			TC
			A_{11}	A_{1z}	A_{1x}	A_{r1}	A_{rz}	A_{rx}	A_{s1}	A_{sz}	A_{sx}	
DM	$MLDM_1$	D_1	TE			TE			TE			
		…										
		D_u	x		x	x			x			TC
		...										
		D_t										
…												
DM	$MLDM_j$	D1		TE			TE			TE		
		…										
		D_u										
		...										
		D_t										
…												
DM	$MLDM_n$	D_1	x	x		x	x		x	x		TC
		…										
		D_u			TE			TE			TE	
		...		x	x		x	x		x	x	TC
		D_t										

Figure 10 : Matrice de l'Intégration Logique des Données Réparties

6- CONCLUSION

Dans cet article nous avons proposé une nouvelle démarche de modélisation de la répartition des données des DW. L'avantage de la démarche proposée est le fait qu'elle constitue une extension au niveau de la modélisation qui peut s'appliquer sur n'importe quelle approche de conception.

L'apport de la démarche proposée est la mise en évidence du coté organisationnel des données d'un DW. Un enrichissement du niveau logique de modélisation est effectué pour garantir une meilleure organisation des données. A ce niveau, un MNLD global du DW est établi. Ensuite, la modélisation logique s'oriente vers deux axes indépendants et qui peuvent se faire en parallèle. Un premier axe consiste à modéliser l'allocation logique des données du DW vers plusieurs DM, le résultat est une MALD. Le deuxième axe consiste à modéliser l'intégration des données vis à vis des sources de données en tenant compte des transformations nécessaires sur les données sources, le résultat est la MILDS. Le résultat donné par le niveau logique est une matrice appelée MILDR du DW et qui n'est autre que la fusion des deux matrices précédentes. Cette dernière permet de visualiser toutes les informations nécessaires pour la répartition des

données d'un DW. Cependant, quelques axes de recherches restent à étudier et à approfondir :

- Jusqu'àprésent il n'y a pas eu de réalisation pour la solution proposée, le travail effectué sera complété ultérieurement par une implémentation.
- Nous envisageons aussi une amélioration du processus d'allocation des données par la prise en compte des caractéristiques du réseau. On pourra intégrer un programme d'optimisation permettant de donner une meilleure allocation possible en tenant compte des caractéristiques du réseau et de sa fiabilité.
- On pourra ensuite, envisager une l'intégration d'une allocation dynamique des données et ceci par l'intégration d'un agent intelligent qui permet de calculer les fréquences d'utilisation des différentes données par les sites de l'organisation.

BIBLIOGRAPHIE

[1] Ralph Kimball, «The Data Warehouse has no centre», Volume 2, Nombre 10. (1999).

[2] Bill Inmon, «Data Mart does not equal Data Warehouse», DM Direct. (1999).

[3] Jean-François Goglin; « La construction du data warehouse, du data mart au data web »; Nouvelles Technologies Informatiques; Ed. HERMES.

[4] KOLSI Nader, « Modélisation de l'intégration des données d'un DW ». Institut Supérieur de Gestion, TunisIII, Tunisie (2000).

[5] Ladjel Bellatreche, Kamalakar Karlapalem, «Some Issues in design of Data Warehousing Systems», Department of computer Science & Technology Clear Water Bay Kowloon, Hong Kong.(1999).

[6] GALACSI, « Conception De Bases De Données : Du schéma conceptuel aux schémas physiques » DUNOD informatique.(1989).

[7] Noaman, A.Y. et K. Barker, "Distributed Data warehouse Design", (under revision for) journal Submission. (2000).

[8] Noaman, A.Y. et K. Barker, "Distributed Data warehouse Architecture and design", the Fourteenth International Symposium on computer and Information Sciences (ISCI'99), Kusadasi, Turki. (1999).

[9] Noaman, A.Y. et K. Barker, "A Horizontal Fragmentation Algorithm for the fact relation in a Distributed Data Warehouse", the Eight International Conference on Information and Knowledge Management (CIKM'99), Kansas, Missouri.(1999).

[10] MESSAOUD Saloua, « Modélisation de la répartition et de la réplication des données». Institut Supérieur de Gestion, TunisIII, Tunisie (2000).

[11] Stefano CERI, Giuseppe PELAGATTI, «Distributed Data base: Principles and systems », McGaw-hill. (1984).

[12] George Gardarin, «Bases de données : Objets et relationnelles », Edition EYRLLES (1997).

[13] P.O'Neil and D.Quass. "Improved query performance with variant indexes. Proceedings of the ACM SIGMOD International Conference on Management of Data. (1997).

[14] P.O'Neil et D.Quass. Improved query performance with variant indexes. Proceedings of the ACM SIGMOD International Conference on Management of Data. (1997).

[15] S. Chaudhuri and V.Narasayya. "Index merging". Proceedings of the International Conference on Data Engineering (ICDE). (1999).

[16] Jane Zaho, Klaus-Dieter Schewe ACM International Conference Proceeding Series, Proceedings of the first Asian-Pacific conference on Conceptual modelling, Dunedin, New Zealand. (2004).