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.

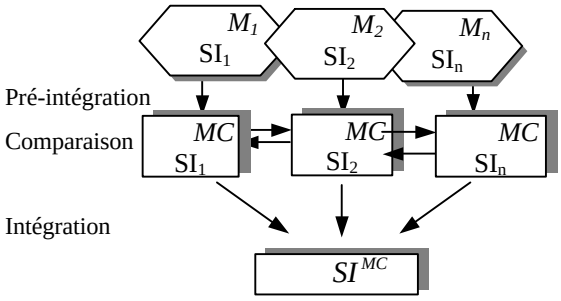

Figure 1: Etapes de l'intégration conceptuelle

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.

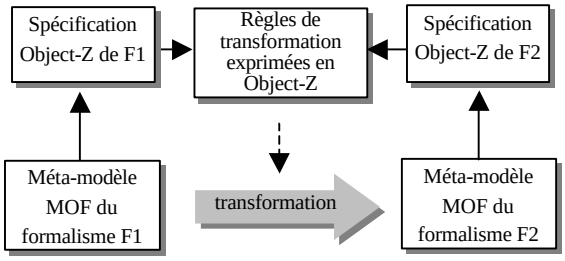

Figure 2: Cadre pour la transformation de RCs

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* $,_{dyn}$ *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 $,_{dyn}$ *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 $,_{dyn}$ *n* ∧ *n* $,_{dyn}$ *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

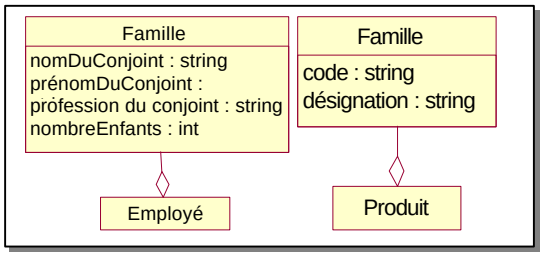

Figure 4. Exemple d'application de la règle 15.

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.

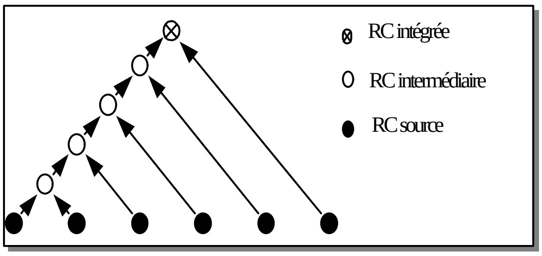

Figure 5. Stratégie à échelle.

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.*