Manipulation d'objets dynamiques dans les bases de connaissances

Valérie Favier

CONSERVATOIRE NATIONAL DES ARTS ET METIERS
CENTRE REGIONAL ASSOCIE DE GRENOBLE

MEMOIRE

présenté en vue d'obtenir

Le diplôme d'Ingénieur C.N.A.M

en

INFORMATIQUE

par

Valérie FAVIER

Manipulation d'objets dynamiques dans les bases de connaissances

soutenu le : 29 septembre 1989

Les travaux relatifs au présent mémoire ont été effectués à l'IMAG, Laboratoire de Génie Informatique, groupe bases de connaissances dynamiques, sous la direction de M. Nguyen et Mme Rieu.

REMERCIEMENTS

Je remercie Monsieur Kaiser, Professeur au Conservatoire National des Arts et Métiers de Paris, d'avoir bien voulu présider le jury de ce mémoire.

J'ai le plaisir de remercier Monsieur Courtin, Professeur à l'Université des Sciences Sociales de Grenoble, pour ses conseils lors de l'élaboration de ce travail.

Je remercie Monsieur Nguyen, Chargé de Recherche INRIA à l'Université Joseph Fourier de Grenoble, et Madame Rieu, Maître de Conférence à l'Université des Sciences Sociales de Grenoble, pour leur encadrement, leur aide et leur amitié tout au long de l'année.

Je remercie Monsieur Vigliano, Directeur de Recherche à la société Syseca-Logiciel, et Madame Gal, Ingénieur en Chef à la société Syseca-Logiciel, d'avoir bien voulu s'intéresser à mon travail et participer au jury.

Je tiens à remercier également toute l'équipe Sherpa pour cette année passée ensemble.

SOMMAIRE

Résumé

Cette étude s'inscrit dans le cadre du projet Sherpa. Ce projet a pour but de concevoir un système de représentation de connaissances basé sur le concept d'objet. L'axe privilégié du projet concerne la gestion de la dynamique des données et des connaissances.
L'étude présentée s'intéresse plus particulièrement à la **dynamique des instances** et la **dynamique des classes.**

La dynamique d'une instance est gérée grâce à des structures. Celles-ci reflètent les différents états possibles d'incomplétude et d'incohérence des instances. Ces structures sont créées à l'aide de règles heuristiques. Si une instance se détériore, une boucle de dégradation permet de gérer cette détérioration. Une instance se détériore si elle devient moins complète et/ou moins cohérente. Cependant, l'utilisateur a la possibilité d'affiner pour une application donnée le concept d'amélioration et de détérioration d'une instance.

La dynamique des classes comprend leur création et leur mise à jour. Plusieurs façons de créer une classe sont envisagées, notamment la création d'une classe à partir d'une structure. Nous étudions, plus particulièrement, l'optimisation de la mise à jour et de la reclassification des instances d'une classe modifiée.

Mots clés : objets - connaissances - dynamique - instances - classes - complétude - cohérence

INTRODUCTION

Cette étude se situe au sein du projet **Sherpa** [REC88]. Ce projet a pour objectif d'implémenter un système de représentation de connaissances à base d'objets. Sherpa réunit deux équipes: l'une travaillant dans le domaine des bases de données et l'autre dans le domaine de l'intelligence artificielle.

Ce projet se décompose en deux parties :

- édification du modèle minimal, appelé SHOOD,
- gestion de la dynamique des données et des connaissances.

L'aspect le plus important de ce projet est la gestion de la dynamique.

Nous nous sommes intéressés plus particulièrement à la gestion de la dynamique des instances et la dynamique des classes.

Les domaines d'applications de ce projet sont divers :

- Conception Assistée par Ordinateur,
- Résolution d'équations aux dérivées partielles,
- Modélisation de bases de séquences dans le domaine de la génétique.

L'étude du modèle minimal a commencé en septembre 1988.

Tout en participant à l'élaboration du modèle minimal, nous avons dans un premier temps étudié la dynamique des instances à partir d'un modèle existant, le modèle CADB (Computer-Aided Design Deductive Data Base) [RIE85]. Nous avons ensuite réalisé l'adaptation de cette étude sur SHOOD et étudié la dynamique des classes uniquement sur ce dernier modèle.

Ce mémoire se décompose en quatre parties.

Le premier chapitre introduit les concepts de l'approche orientée objet. Un état de l'art permet d'avoir un aperçu de quelques systèmes orientés objet existants. Un paragraphe est consacré au projet Sherpa.

Le deuxième chapitre introduit une manière originale de traiter la dynamique des instances, à l'aide du concept de structure. Dans deux modèles CADB et SHOOD, nous étudions la gestion de l'évolution d'une instance.

Le troisième chapitre traite de la dynamique des classes. En particulier, nous abordons la création d'une classe à partir d'une structure, et l'optimisation des mises à jour des instances, lors de la mise à jour d'une classe, à l'aide des structures.

Le quatrième chapitre aborde la partie réalisation. Un prototype a, en effet, été réalisé sur Macintosh II, en Le_Lisp.

1 L'APPROCHE ORIENTEE OBJET

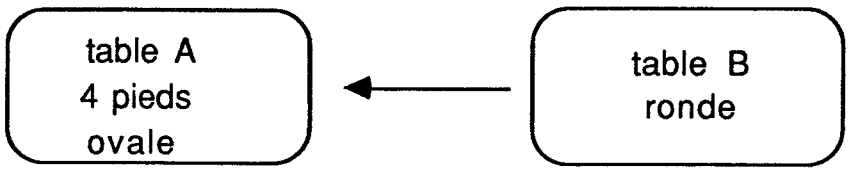
1.1 Introduction

La complexité, la taille et la dynamique des données évoluant, les chercheurs ont été amenés à considérer une nouvelle façon de manipuler et représenter les informations. Dans des domaines tels que le génie logiciel et l'intelligence artificielle des recherches se sont dirigées vers une approche orientée objet. Cette approche a pour avantages : la modularité, la souplesse des systèmes et le partage des connaissances.

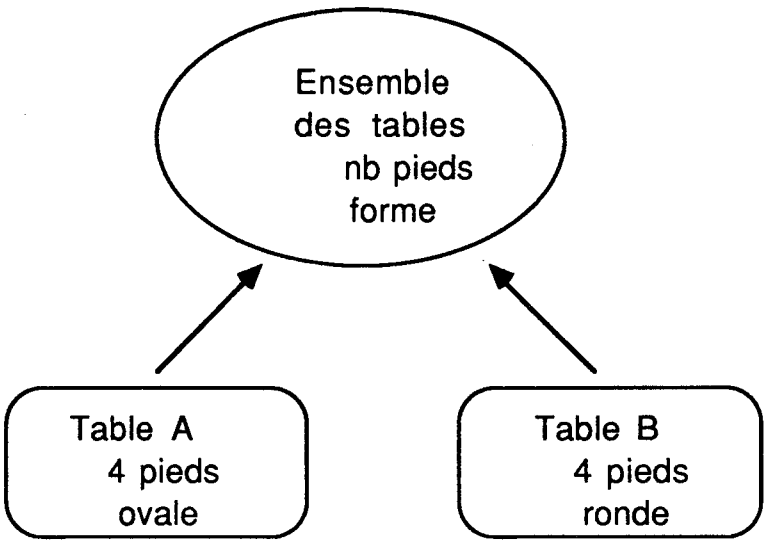
Les premières idées de la programmation orientée objet sont issues du langage SIMULA, conçu pour la simulation (Dahl et Nygaard 1966). On peut également citer SMALLTALK (Goldberg et Robson 1983 [GOL83]), conçu pour le domaine du génie logiciel; les applications de ce langage concernant l'interface homme-machine sont notamment utilisées pour le Macintosh.

Contrairement à une programmation classique, dans l'approche orientée objet, la séparation procédures - données n'existe pas. Une entité de base est considérée : l'objet. Celui-ci contient à la fois des données et des procédures. Par exemple, on peut créer un objet "table" comportant deux données : son nombre de pieds et sa forme, et une procédure : calcul-surface, qui permet de calculer la surface du plateau de la table.
Les différents systèmes orientés objet divergent par la façon de représenter et de partager la connaissance.

Pour représenter la connaissance, deux théories sont utilisées : les **prototypes** et les **ensembles**. Pour plus de compréhension, nous allons aborder ces deux théories à l'aide d'un exemple. Nous voulons représenter une table A et une table B.
Dans la philosophie des prototypes, nous allons définir une table particulière, la table A. Pour créer une autre table B, nous définissons pour B les caractéristiques différentes de A et référençons la table A pour les propriétés communes aux deux tables. Les différents objets sont tous au même niveau, les tables A et B sont deux prototypes.



Une table A ayant 4 pieds et une forme ovale est créée. Si nous souhaitons créer une table B ayant 4 pieds et une forme ronde, seule la forme ronde est spécifiée, le nombre de pieds est pris dans le prototype A.
Dans la philosophie des ensembles, nous allons d'abord créer un **ensemble** contenant les **caractéristiques générales** de toutes les tables. La table A est un membre de l'ensemble. Une nouvelle table B est aussi membre de l'ensemble des tables. Nous distinguons donc deux niveaux : l'ensemble, et les objets particuliers appartenant à cet ensemble, appelés des **instances**.



On crée en premier l'ensemble des tables, avec ses caractéristiques : nombre de pieds et forme. Puis les tables A et B sont créées en donnant une valeur aux caractéristiques de l'ensemble des tables.

L'approche orientée objet est aussi caractérisée par le partage des connaissances. Deux méthodes sont employées. La première méthode est **l'héritage** qui introduit une notion de hiérarchie entre les objets; le partage de la connaissance est rigide et se fait toujours à travers cette hiérarchie; un objet A partage ses données avec toujours le même objet B. La deuxième méthode est la **délégation** : le partage se fait selon les besoins de l'objet; par exemple, l'objet A partage la connaissance "nombre de pieds" avec l'objet B, mais partage la connaissance "forme" avec l'objet C.

Le terme d'approche orientée objet recouvre plusieurs types de systèmes. Le choix d'un de ces systèmes est fonction d'une part des modes de représentation et de partage des connaissances et d'autre part de la philosophie des domaines d'applications (les applications de génie logiciel et d'intelligence artificielle n'ont pas les mêmes besoins). Deux approches seront étudiées, d'une part **les systèmes à base de classes** qui reposent sur la théorie des **ensembles** et l'**héritage**, et d'autre part **les systèmes à base de frames** qui reposent sur la théorie des **prototypes** et l'**héritage**. Nous ne ferons que citer les langages acteurs (Plasma [HEW75]) basés sur les prototypes et la délégation.

Pour plus de compréhension, une même notion est ici toujours mentionnée avec le même terme, bien que selon les systèmes ces notions ne portent pas toujours le même nom. Pour chaque approche nous avons dégagé les caractéristiques "propres" à celle-ci, mais il est évident que beaucoup de modèles hybrides existent (par exemple Loops [STE86], Shirka [REC87]).

1.2 Les concepts des systèmes à base de classes

Le concept fondamental est l'objet qui comporte des propriétés structurées.
L'objet est constitué de procédures et de données. Les objets communiquent par un échange de **messages** qui permettent de manipuler les objets par l'intermédiaire de leurs méthodes.

1.2.1 Classes et instances

Nous allons définir les notions de classe et d'instance.
Une classe est un ensemble d'objets répondant aux mêmes **propriétés**: **méthodes** (procédures) et **attributs** (données simples). Ceci permet une taxonomie des objets.
Nous pouvons, par exemple, créer une classe AVION qui représente l'ensemble des avions. Cette classe a deux attributs : imat et fuse qui permettent respectivement d'identifier un avion par son immatriculation (imat) et de donner la longueur du fuselage (fuse).
Dans certain système [DAN88], une distinction est faite entre la définition de la classe (ses propriétés) qui est appelée "type abstrait" et l'ensemble des objets correspondant à cette classe. Ceci permet d'utiliser l'algèbre associée aux types abstraits. Nous n'utiliserons que le terme de classe (ou schéma) pour désigner l'une ou l'autre de ces notions.

Une instance est un objet qui instancie les attributs d'une classe.
Par exemple, l'avion A est une instance de la classe AVION. Les attributs imat et fuse de la classe AVION prennent respectivement les valeurs tango et 45.

Beaucoup de systèmes tendent vers une méta-circularité, c'est-à-dire vers un système où tout objet est instance d'un ou plusieurs autres objets. Cela permet de construire un système souple pouvant être étendu relativement facilement. Le concept de méta-classe est alors introduit. Une méta-classe permet de créer des classes. Les classes instancient les attributs de la méta-classe : nom, propriétés.... L'ensemble des objets est alors organisé en un graphe, dit d'instanciation, ayant pour racine une méta-classe. Certains systèmes (Loops, Smalltalk-80) autorisent l'emploi de plusieurs méta-classes, permettant ainsi la création de classes de statut différent (une seule méta-classe est racine du graphe d'instanciation). La méta-classe racine est instance d'elle-même; lors de la création d'une base, nous disposons d'un "bootstrap" contenant au moins la création de cette méta-classe. Nous avons donc les instances des méta-classes qui sont des classes, et les instances des classes qui sont dites instances terminales.

Exemple:

Considérons la méta-classe: META, et la classe: AVION.

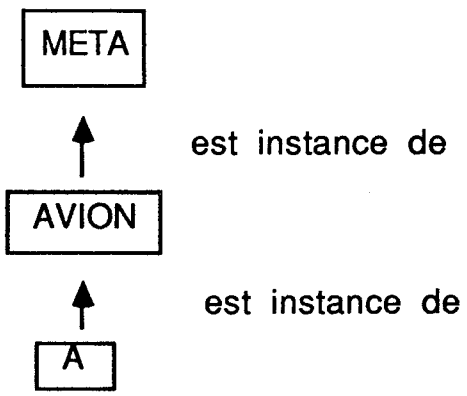
META
nom
propriétés

classe AVION
  nom : avion
  propriétés :
    imat
    fuse

avion A
  imat : tango
  fuse : 45

La classe AVION instancie les attributs de la méta-classe META en donnant comme valeur à "nom" : avion et comme valeur à "propriétés" : imat et fuse.
Un avion A est instance de la classe AVION, s'il instancie les propriétés imat et fuse, par exemple en leur donnant les valeurs tango et 45.

graphe d'instanciation :



Dans certains systèmes (Orion [BAR87], GemStone [PEN87], CADB [RIE85]), un type est associé à chaque attribut. Dans ce cas, les valeurs prises par cet attribut sont restreintes. Ceci permet un contrôle dès l'instanciation de l'attribut. L'ensemble des valeurs possibles s'appelle un domaine. Généralement ce domaine est une classe.
Pour la classe AVION le domaine de imat est, par exemple, "chaîne" et le domaine de fuse est "entier". Quand une valeur est donnée à l'attribut "fuse" le système vérifie que celle-ci est bien un entier.

La classe AVION devient alors:

classe AVION
  nom : avion
  propriétés :
    imat : chaîne
    fuse : entier

1.2.2 Messages et méthodes

Un message est composé d'un sélecteur et d'arguments. Le sélecteur représente la méthode à exécuter. Un des arguments est le nom de l'objet receveur, les autres sont les valeurs des paramètres de la méthode.
exemple de message:
  send (méthode objet-receveur paramètre1 ... paramètre n)

Un message ne doit dépendre ni du type, ni de l'implémentation des données. Quand un objet reçoit un message, il utilise les méthodes; ce sont les codes permettant de déterminer le comportement de l'objet vis à vis du message. Un ensemble de messages s'appelle un protocole. Les utilisateurs manipulent les objets sans en connaître la structure, on parle alors **d'encapsulation** des propriétés des objets. L'utilisateur peut donc raisonner en terme d'abstraction.

Exemple:
Ajoutons une propriété à la classe AVION : la méthode calculer-décollage qui permet de calculer la distance de décollage en fonction de la longueur du fuselage

classe AVION
  nom : avion
  propriétés :
    imat : chaîne
    fuse : entier
    calculer-décollage
      retourner (fuse * 50)

Prenons un message :
  send (calculer-décollage A)
Le sélecteur de ce message est calculer-décollage; l'objet receveur est l'avion A; il n'y a pas de paramètre.
Si ce message est envoyé, l'avion A applique la méthode calculer-décollage et renvoie la valeur 2250 (45 * 50).

1.2.3 L'héritage

Une caractéristique fondamentale de l'approche orientée objet est le concept d'héritage. Celui-ci permet de limiter la redondance d'informations en autorisant le partage de la connaissance; il évite la répétition de toutes les propriétés pour chaque classe d'objets. En effet, l'héritage permet la transmission des propriétés d'une classe vers une ou plusieurs classes de niveau inférieur. Une hiérarchie est ainsi instaurée entre les classes. L'ensemble des classes forme alors soit un graphe d'héritage (on a alors un héritage multiple) (Orion), soit un arbre (l'héritage est simple) (CADB, GemStone). Les classes d'un niveau supérieur sont appelées super-classes, et celles d'un niveau inférieur : sous-classes.
En parcourant le graphe d'héritage vers le bas, une spécialisation est définie. En effet, la connaissance est de plus en plus spécifique dans les sous-classes. Une sous-classe spécialise la connaissance de sa super-classe en possédant des propriétés supplémentaires ou en affinant les propriétés héritées (soit en modifiant la définition de la propriété, soit en restreignant le domaine de celle-ci).

Par exemple, nous pouvons créer une sous-classe de la classe AVION: "LONGCOUR" qui représente la classe des avions long-courrier; à cette classe nous ajoutons une propriété "distance" qui permet de connaître la distance que peut parcourir un avion. Nous affinons donc les propriétés de la classe AVION, en ajoutant un attribut à la classe LONGCOUR. L'attribut distance n'apparaît pas dans la classe AVION, car l'utilisateur n'est intéressé par cette valeur que dans le cas d'un avion long-courrier.

Les propriétés d'une sous-classe sont donc l'union des propriétés de ses super-classes, plus ses propriétés propres. Quand une propriété est re-définie au niveau d'une sous-classe, celle-ci se substitue (override) à la définition héritée de la super-classe.

A la racine du graphe (ou de l'arbre) d'héritage se trouve une classe particulière, généralement appelée OBJET. Toutes les classes héritent donc au moins des propriétés de OBJET. L'arête entre deux noeuds (deux classes) du graphe (ou de l'arbre) représente la relation "is-a" (est-un).
Les héritages de propriétés peuvent être soit automatiques (Orion, GemStone, CADB), soit manuels (Encore [SKA86]).
Certain système (CADB) considère non seulement l'héritage de propriétés, mais aussi l'héritage des instances, on parle de classification automatique.

Exemple:

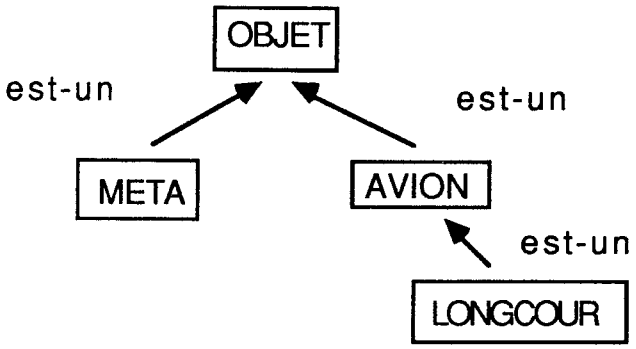
Reprenons l'exemple des avions en ajoutant une autre classe LONGCOUR qui est la classe des avions long-courrier.

```
classe LONGCOUR
  nom : longcour
  propriétés :
     distance : entier     ; distance est la distance maximale que peut parcourir un avion.
```

Si la classe LONGCOUR est une sous-classe de la classe AVION, elle hérite des attributs imat et fuse et de la méthode "calculer-décollage" de la classe AVION. Ses propriétés sont

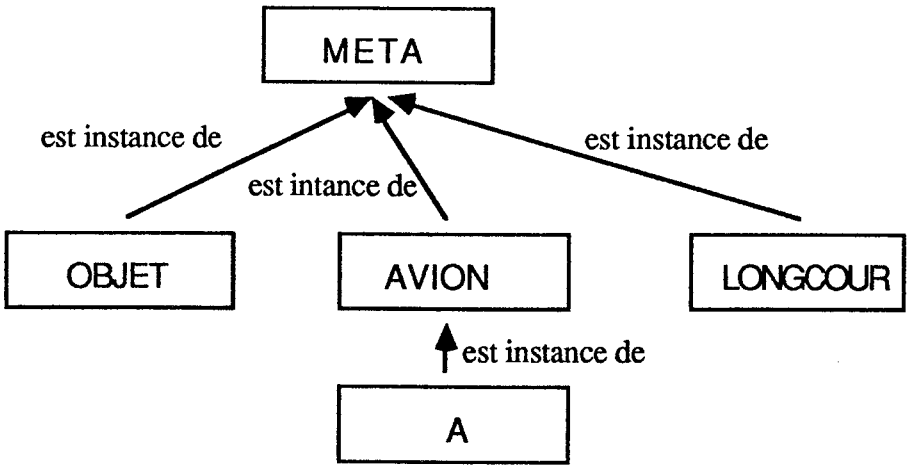
donc : imat, fuse, distance et calculer-décollage. Une instance de LONGCOUR doit donc instancier les attributs: imat, fuse et distance.

graphe d'héritage:



OBJET est la super-classe de META, AVION et LONGCOUR. AVION est la super-classe de LONGCOUR. META, AVION et LONGCOUR sont des sous-classes d'OBJET.

nouveau graphe d'instanciation:



Nous pouvons remarquer que les instances terminales apparaissent seulement dans le graphe d'instanciation.

1.2.4 Mise à jour des schémas

Dans le cadre de l'intelligence artificielle ou de la CAO [RIE86], le graphe d'héritage représente le monde réel, il est donc indispensable que ce graphe puisse être modifié. En génie logiciel, le graphe doit fournir à l'utilisateur un cadre rigoureux pour éviter les risques d'erreurs; dans ce cas le graphe doit rester statique, et ne peut être modifié.

Plusieurs types d'opérations peuvent être effectués sur les schémas. On distingue d'une part les mises à jour des propriétés d'une classe, et d'autre part les modifications effectuées sur le graphe: modification des arêtes et addition ou suppression de classes.

La limite et la résolution de ces opérations, pour conserver la cohérence du graphe, sont différentes selon les systèmes considérés. Par exemple dans Encore [SKA86], à chaque modification, une nouvelle version de la classe est créée : la modification des classes est alors rendue transparente aux programmes.

Lors d'une mise à jour, nous assistons ou non à une propagation des modifications sur les instances et les classes.

1.2.5 Objet composé

On parle d'objets composés dans le cas d'interconnexion entre objets. Le lien entre les objets est le lien "est-partie-de" (is-part-of). La dépendance existant entre ces objets est traitée différemment selon les systèmes, et représente des notions plus ou moins fortes [JEA89].
Par exemple, on peut considérer qu'un avion est composé de moteurs. A la création d'une instance de la classe AVION, il faut créer les moteurs correspondant à cet avion particulier.

1.2.6 Valeur par défaut - Valeur partagée

On parle aussi du concept de valeur par défaut. Si un attribut n'a pas de valeur calculée ou spécifiée, on lui donne la valeur par défaut.
Par exemple, à la création de la classe AVION, la valeur par défaut de l'attribut "fuse" est 30. A la création d'un avion B, si l'utilisateur ne connaît pas la valeur de l'attribut fuse, le système considère 30 comme la bonne valeur; mais l'utilisateur peut modifier à tout moment cette valeur.

La valeur partagée est aussi introduite dans certain système (Orion). C'est la valeur d'un attribut, donnée à toutes les instances d'une classe. Cette valeur ne peut être modifiée au niveau de l'instance, ni par l'utilisateur, ni par le système.
Par exemple, si à la création de la classe AVION, la valeur partagée de l'attribut fuse est 20, toutes les instances de la classe AVION auront 20 comme valeur de l'attribut fuse; l'utilisateur ne peut pas modifier cette valeur au niveau d'un avion particulier.

1.3 Les concepts des systèmes à base de frames

Les systèmes à base de frames sont surtout utilisés dans le domaine de l'intelligence artificielle. Ils supportent la théorie des **prototypes** et utilise l'**héritage**. L'entité de base est le **frame**. C'est Marvin Minsky [MIN75], en 1975, qui a introduit ce concept.

Comme système à base de frames, nous pouvons citer KRL [BOB77] conçu en 1977 dans le cadre de recherche sur la compréhension du langage naturel; FRL [ROB77] conçu pour mettre en pratique les idées de Minsky en 1977; SHIRKA [REC87] qui utilise non seulement l'héritage, mais aussi le concept d'instanciation.

Un frame permet de structurer un objet. Un frame est composé d'**attributs**. A chaque attribut est associé une ou plusieurs **facettes**. Une facette permet de décrire les caractéristiques des attributs; c'est-à-dire la nature des valeurs de l'attribut, la manière de calculer ou d'utiliser cette valeur. Nous distinguons deux types de facettes; d'une part les **facettes déclaratives** qui permettent de déterminer le type ou la valeur (non calculée) d'un attribut; et d'autre part les **facettes procédurales** qui permettent un contrôle ou un calcul de la valeur de l'attribut. A une facette procédurale est associé un attachement procédural, appelé **réflexe**, celui-ci permet de calculer la valeur de la facette.

Exemple:

Nous allons définir le frame AVION correspondant à la classe AVION de l'exemple précédent.
Nous disposons d'une facette "$un" qui permet de spécifier le type de l'attribut; d'une facette "$intervalle" qui spécifie l'intervalle dans lequel l'attribut doit prendre ses valeurs; d'une facette "$valeur" qui spécifie la valeur d'un attribut; et d'une facette "$si-besoin" qui permet de calculer la valeur d'un attribut. La facette "$si-besoin" est une facette procédurale, les autres sont déclaratives.

```
Frame AVION
        imat : $un  chaîne
        fuse : $un entier
               $intervalle [0 300]
    deco : $si-besoin  calculer-décollage
```

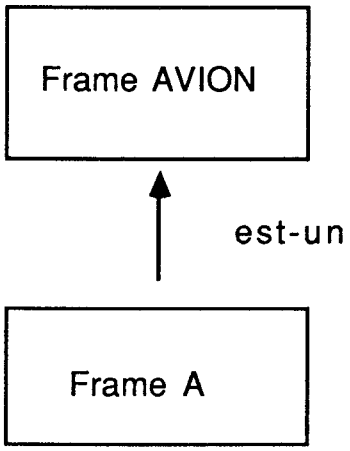
L'attribut fuse doit prendre comme valeur un entier compris entre 0 et 300.
Calculer-décollage est le réflexe associé à la facette $si-besoin.

La création d'un avion A, équivaut à la création d'un sous-frame de AVION, le frame A. Ce sous-frame spécialise le frame AVION en donnant des valeurs précises aux attributs.

```
Frame A
   imat : $valeur tango
   fuse : $valeur 45
```

Deux approches sont possibles pour l'attribut "deco". La première approche consiste à calculer et garder la valeur de cet attribut; cela pose un problème au niveau de la maintenance de la cohérence des frames lors de la modification d'attributs. Par exemple, si l'on modifie la valeur de l'attribut "fuse", il faut aussi modifier la valeur de l'attribut "deco". La deuxième approche consiste à ne pas garder la valeur de l'attribut "deco", mais à calculer celle-ci à chaque fois que l'attribut est accédé.

Graphe d'**héritage** :



Dans un frame nous n'avons pas de méthode, nous avons des procédures rattachées à certains attributs.
Les frames sont organisés sous forme de graphe d'héritage, on distingue donc des super-frames et des sous-frames. Le concept d'héritage est le même que pour les systèmes à base de classes. Comme les frames sont basés sur la théorie des prototypes, nous n'avons pas de notion d'instanciation. Tous les frames sont des prototypes reliés par le seul lien d'héritage "est-un".

Les points communs entre les classes et les frames sont:

- l'intégration des données et des procédures dans les objets,
- l'utilisation du concept d'héritage.

Les différences entre les classes et les frames sont:

- Dans les systèmes à base de classes, les objets sont activés par des messages, c'est la **programmation par objet**; dans les systèmes à base de frames nous accédons directement à la valeur d'un attribut: c'est une **programmation par accès**.

- Comme les classes contiennent des méthodes, on dit que le modèle est **procédural**; les classes contiennent leur propre comportement. Dans les frames, le modèle est dit **déclaratif** : un frame ne peut pas exister par lui-même, une couche de niveau supérieur doit être créée pour les manipuler; cette couche est souvent un mécanisme d'inférence; nous remarquons le côté déductif de ces systèmes.

- La notion d'instanciation n'apparaît que dans les systèmes à base de classes. Pour les systèmes à base de frames, seul le lien d'héritage est défini (sauf pour Shirka qui est un système hybride).

1.4 Etat de l'art

Nous allons étudier quatre systèmes à base de classes : Orion, GemStone, O_2, Loops et un système à base de frames FRL.

1.4.1 Orion

Orion [BAR87] est développé à MCC. C'est un système de base de données orienté objet. L'accent est mis sur la dynamique de l'évolution des schémas et sur les objets composés.

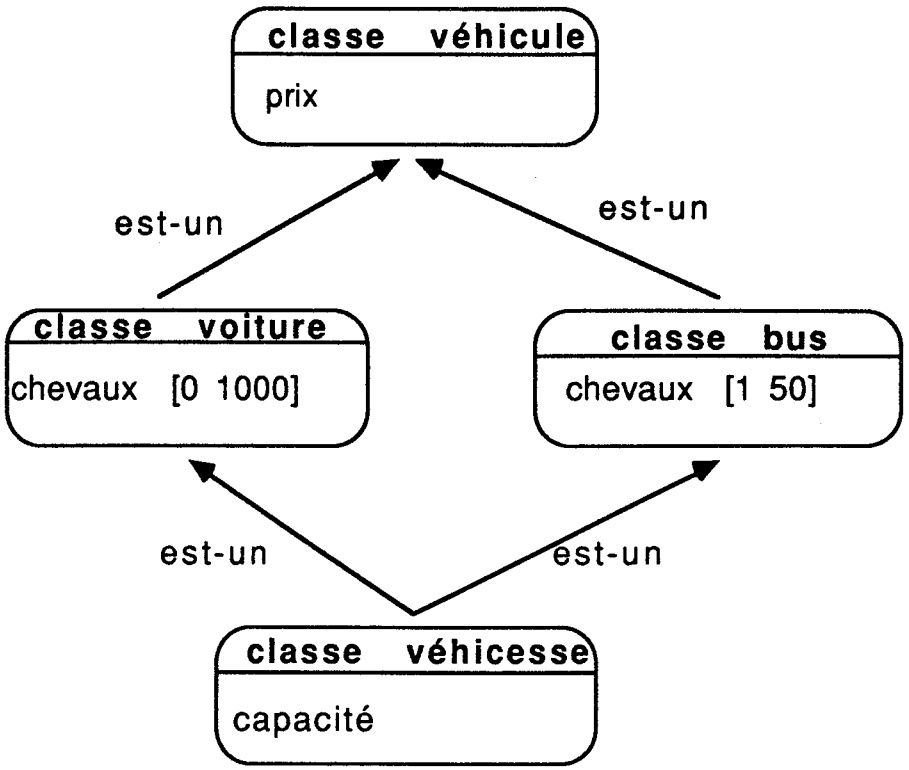
Orion possède un ensemble de caractéristiques appelées **invariants**, et un ensemble de règles qui permettent de vérifier que ces invariants sont respectés lors des modifications de schémas. Cinq invariants sont considérés:

- Le graphe de classes est composé d'une racine qui est directement connectée à un graphe acyclique, comportant des noeuds et des arêtes dûment nommés.
- Toutes les propriétés d'une classe ont des noms différents.
- Si une propriété dans plusieurs super-classes a la même origine (on parle de même identité), la sous-classe associée à ces super-classes n'hérite qu'une seule fois de cette propriété.
- Une classe hérite de toutes les propriétés de ses super-classes sauf en cas de conflit de noms.
- Le domaine d'un attribut hérité est soit le même que le domaine de la super-classe, soit une restriction de ce domaine. Les valeurs partagées ou les valeurs par défaut doivent appartenir à ce domaine.

Douze **règles** sont définies pour préserver les invariants lors de la modification de schémas. Ces règles portent sur quatre points.

Le premier point traite **les conflits de noms ou d'identités**. Deux problèmes de conflit sur les noms des propriétés sont considérés. Le premier conflit se situe entre une classe et ses super-classes. Si une propriété est définie au niveau d'une classe, elle est prioritaire par rapport à la même propriété définie à un niveau supérieur. Le deuxième conflit se pose quand une propriété de même nom est définie de façon différente dans plusieurs super-classes d'une classe. Dans ce cas la propriété héritée est celle de la super-classe considérée comme la première; l'ordre des super-classes est donc important, il peut être modifié à tout instant.

Exemple:



Dans l'exemple, nous avons quatre classes: **véhicule, voiture, bus, véhicesse** (véhicule à essence). La classe **véhicesse** possède une propriété propre: capacité et deux propriétés héritées: prix et chevaux. La propriété prix est héritée une seule fois, alors qu'elle se trouve dans la classe **voiture** et la classe **bus**, car pour les deux classes elle a la même classe origine (**véhicule**). La propriété chevaux est héritée de la première super-classe de **véhicesse**, ici (par exemple) de **bus**. Les propriétés chevaux des classes **bus** et **voiture** ne représentent pourtant pas les mêmes notions (une matérialise les chevaux fiscaux et l'autre les chevaux din). Le domaine de la propriété chevaux de la classe **véhicesse** est celui de la propriété chevaux dans la classe **bus** ([1 50]). Si l'on désire conserver les deux notions de la propriété chevaux (fiscaux et din), il faut renommer la propriété chevaux de la classe **voiture**. En effet dans une classe deux propriétés doivent avoir des noms différents.

Le deuxième point concerne **la propagation des propriétés**: la mise à jour d'une méthode ou d'un attribut est propagée dans toutes les sous-classes concernées, si cela n'introduit pas de conflit.

Le troisième point s'intéresse à **la manipulation du graphe**: il est en effet indispensable de régir l'addition et la suppression de noeuds ou d'arêtes dans le graphe d'héritage. On ne peut, par exemple, tolérer qu'un noeud se trouve isolé suite à la suppression d'une arête.

Le dernier point gouverne **les objets composés**. Un objet composé permet de modéliser le lien "is-part-of" (est-partie-de) entre un objet et son père. Un objet composé est défini par une racine et plusieurs objets fils (fils pouvant à leur tour engendrer d'autres fils). Les objets fils sont des objets dépendants. L'objet père est le propriétaire exclusif des objets fils. La création d'un "lien composé" permet de matérialiser la relation de composition.

exemple:

VOITURE -------> corps

(pris dans la classe: CORP)

Une voiture est composée d'un corps pris dans la classe CORP. La classe VOITURE possède un "lien composé" avec la classe CORP à travers "l'attribut composé" corps.

Un attribut non composé ne pourra jamais devenir composé, par contre un attribut composé peut être changé en attribut non composé. Quand un objet père est supprimé tous les objets fils dépendants sont supprimés. On peut enlever un lien de dépendance entre deux objets, tout en gardant ces deux objets liés; un objet A fait toujours référence à l'objet B, mais la suppression de A n'entraîne plus la suppression de B.

La taxonomie des modifications des schémas dans Orion est la suivante:

- modifications d'attributs
- modifications de méthodes
- modifications d'arêtes
- modifications de noeuds

Les auteurs différencient une vingtaine d'opérations de mise à jour différentes. Nous ne soulignerons ici que quelques opérations.

La modification ou l'addition d'un attribut est propagée dans toutes les sous-classes, excepté dans les cas de conflit de noms. Par exemple, l'addition de l'attribut "capacité" dans la classe **véhicule** est uniquement propagée dans les classes **voiture** et **bus**.

La suppression d'un attribut n'est possible que s'il n'est pas hérité, elle est propagée dans toutes les sous-classes. Par exemple, la suppression de l'attribut prix dans la classe **véhicesse** n'est pas autorisée.

En cas de modification d'une valeur par défaut, les instances ne sont pas modifiées, car la valeur par défaut est conservée au niveau de la classe et non pas au niveau de l'instance.

En cas de modification de noeuds ou d'arêtes, les propriétés sont héritées. Par exemple, si une nouvelle classe est ajoutée comme super-classe de la classe **véhicule**, les classes **voiture, bus** et **véhicesse** bénéficient des propriétés de cette nouvelle classe.

Dans le cas de la suppression de l'arête entre une classe B et sa super-classe A, les super-classes immédiates de A deviennent aussi les super-classes de B. Cela est possible car au sommet du graphe d'héritage on a la classe OBJET (cette classe ne peut être évidemment supprimée). La classe B perd les propriétés définies au niveau de A.

La suppression d'une classe entraîne la suppression des liens avec ses sous-classes, puis avec ses super-classes, et enfin la suppression du noeud dans le graphe. Toutes les instances de la classe sont alors supprimées.

Pour la modification des instances, lors de la modification d'un graphe, Orion utilise la méthode de "**filtrage**": une instance n'est modifiée qu'au moment de son accès.

1.4.2 GemStone

GemStone [PEN87] est développé à Servio Logic. Il allie le concept de programmation orientée objet avec les possibilités d'un système de gestion de base de données. Il utilise un langage orienté objet appelé: OPAL.

Pour stocker les objets, GemStone offre quatre formats de base:

- auto-identifiant (caractères, booléens....): les instances sont considérées comme atomiques,
- multiplet (chaîne...): les instances sont non structurées,
- pointeur: les instances contiennent une référence à d'autres objets,
- collection non séquentielle (ensemble, tas...): les instances sont non ordonnées.

Pour pallier le problème de protection des données, GemStone introduit le concept de **segment** qui est une unité d'autorisation. Chaque objet appartient à un segment, et chaque segment appartient à un propriétaire. Les droits d'accès à un objet d'un segment sont régis par le propriétaire. Les utilisateurs sont modélisés comme les instances d'une classe particulière.

Comme pour Orion, cinq **invariants** sont définis :

- L'implémentation physique d'un objet est déterminée par sa classe.
- Les relations d'héritage entre les classes forment un arbre (une sous-classe hérite d'une seule super-classe). La racine de cet arbre est la classe OBJET.
- Les propriétés d'une classe sont héritées par toutes ses sous-classes.
- Le domaine d'un attribut hérité est compatible avec celui de sa super-classe. Ce domaine est pris dans une classe ou dans une de ses sous-classes.

(Les deux derniers concepts sont communs avec Orion).

- La valeur d'une variable fait toujours référence à un objet existant.

Nous introduisons, ici, quelques opérations de modification de schémas.
Renommer un attribut n'est possible que s'il n'est pas hérité, et s'il n'existe pas déjà un attribut ayant le même nom. Cet attribut est hérité dans toutes les sous-classes.
L'ajout d'un attribut ne peut se faire que s'il n'existe pas un attribut de même nom dans la classe, ou dans les sous-classes, car il y a propagation dans les sous-classes. Cet attribut prend la valeur nil dans les instances existantes.
La suppression d'un attribut est permise si cet attribut n'est pas hérité. Cette suppression n'est pas propagée.
Une classe ne peut être supprimée si elle possède des instances.
Lors de l'ajout d'une classe, on spécifie sa super-classe et ses sous-classes.

La modification des instances se fait suivant la méthode de "**conversion**". C'est-à-dire que cette opération a lieu dès qu'une classe est modifiée. Cette méthode est en opposition avec le filtrage utilisé dans Orion. Dans la première méthode, une perte de temps est enregistrée au moment de la modification d'une classe; dans la deuxième méthode, cette perte de temps compromet la vitesse d'exécution lors de l'accès à une instance.

1.4.3 O2

O2 [LEC88] est un système de base de données orienté objet. Il est développé à Paris dans le cadre du GIP Altaïr.
Les instances d'une sous-classe forment un sous-ensemble d'inclusion de sa super-classe. Les instances d'une classe sont donc aussi instances des classes de niveau supérieur. La substitution d'une propriété héritée n'est donc pas permise, on peut seulement affiner le domaine de cette propriété.

Les données sont non seulement typées, mais aussi fortement structurées. Les constructeurs ensembliste et tuple sont en effet utilisés pour représenter les objets complexes. Les valeurs des objets sont donc non seulement des valeurs atomiques (entier, chaîne...), mais peuvent être des ensembles ou des tuples. Les objets sont caractérisés par leurs valeurs et leur identifiant. Un identifiant permet de référencer de manière unique un objet, et sa valeur est indépendante des valeurs des attributs de l'objet. Deux objets peuvent donc avoir des valeurs d'attributs égales sans être pour autant identiques.

On peut ajouter des attributs à une classe, mais ces nouveaux attributs ont un statut particulier. Ils sont considérés comme "exceptionnels" et ne sont accessibles et modifiables que via les méthodes. L'ajout et la suppression de classes ne peuvent se faire que sur des classes terminales.

1.4.4 Loops

Loops [STE86] est à l'origine un langage orienté objet, mais il a évolué vers une représentation orientée objet.

Le système supporte le multi-héritage. La racine du graphe d'instanciation est CLASS, et la racine du graphe d'héritage est OBJET.

A l'intérieur d'une classe deux sortes attributs sont distingués: les attributs communs à toutes les instances de la classe et les attributs spécifiques à une seule instance.

Pour résoudre le problème de conflit de noms (plusieurs super-classes d'une classe ont des propriétés portant le même nom), une liste des classes précédentes est créée, la propriété héritée est celle de la première classe de la liste. Contrairement à Orion, cette liste ne peut pas être modifiée par l'utilisateur.

Loops offre la possibilité de créer des classes dites "mixin" qui permettent de regrouper des propriétés, mais ne possèdent aucune instance.

La notion d'objet composé dans Loops fait non seulement intervenir la dépendance entre les objets, mais oblige aussi tous les composants de l'objet à être instanciés en même temps. Les objets composés sont instances de classes particulières dont la super-classe est le mixin Objet-Composé.

1.4.5 FRL

FRL a été conçu au MIT par Goldstein et Roberts [ROB77].

C'est un système à base de frames, reposant sur les prototypes et l'héritage. Il répond aux critères décrits dans le paragraphe 1.3.

Ce système supporte un héritage multiple. Mais les conflits d'héritage ne sont pas traités. La racine du graphe est THING.

Les attributs ne sont pas typés. Ils peuvent prendre soit une valeur, soit une liste de valeur.

Pour chaque frame trois attributs sont pré-définis : l'attribut ako (A-Kind-OF) qui prend comme valeur la liste des super-frames; l'attribut instance qui prend comme valeur la liste des sous-frames; et l'attribut classification qui permet de différencier un frame générique d'un frame élémentaire. Dans notre exemple sur les avions, le frame avion est considéré comme un frame générique et le frame A comme un frame élémentaire.

Si l'on utilise la facette $si-besoin, la valeur calculée pour l'attribut est sauvegardée. Si l'on ne désire pas garder une valeur (si elle a un caractère évolutif) nous pouvons utiliser le

symbole "%" suivi d'une fonction de calcul, dans ce cas la valeur est calculée à chaque lecture de l'attribut.

1.5 Présentation du projet Sherpa

Le projet Sherpa [REC88] s'intéresse particulièrement à la gestion de la **dynamique** dans les systèmes de représentation de connaissances : dynamique des instances, des classes et du raisonnement.

La **dynamique des instances** comprend la gestion de leur complétude et de leur cohérence, leur classification automatique dans le graphe, la manipulation d'objets composites.

La **dynamique des classes** permet de gérer la modification des classes et celle du graphe d'héritage.

La **dynamique du raisonnement** tient compte du raisonnement non monotone (un fait peut être vrai à un instant T, mais s'avérer faux à un instant T+1) et du raisonnement non déterministe (la manière dont a été donnée la valeur d'un attribut peut être remise en cause).

Le modèle minimal du projet Sherpa s'appelle SHOOD.
Ce modèle repose sur la théorie des ensembles et l'héritage.
Deux aspects sont distingués dans Sherpa : la couche de **représentation** (c'est le modèle SHOOD) et la couche de **manipulation.**

1.5.1 La couche de représentation

La couche de représentation permet de modéliser les connaissances sur les objets.

Le modèle SHOOD est **méta-circulaire**: tout est objet (classe, instance...), tout objet est instance d'un autre objet. Les classes sont instances d'autres classes appelées méta-classes; les attributs, les contraintes, les inférences sont instances de classes particulières. La méta-circularité permet d'avoir un modèle réflexif et extensible.

La racine du graphe d'instanciation est la méta-classe META.
Plusieurs méta-classes peuvent être créées. Ceci permet de générer des classes ayant des statuts particuliers. Certaines méta-classes sont pré-définies. On peut citer : Méta-à-clé permettant de définir des classes dont les instances sont identifiées par des clés, Méta-méthode qui permet de définir des méthodes ...

SHOOD supporte un **héritage multiple** : une classe peut avoir plusieurs super-classes. En plus de l'héritage multiple, nous offrons le concept de disjonction entre deux classes. Si deux classes sont disjointes alors elles n'ont aucune instance en commun.

Dans le modèle minimal la **spécialisation** est **stricte** : l'utilisateur ne peut redéfinir une propriété définie dans une super-classe, il peut seulement affiner son domaine de

spécialisation. Une instance d'une classe appartient donc aussi aux super-classes de sa classe.
Nous avons le concept de **multi-instanciation.** Si la classification le permet, une instance peut appartenir à plusieurs classes, ces classes n'ayant pas de lien d'héritage entre elles. La classification des instances est gérée automatiquement par le système.

Les classes ont une définition proche des frames. Elles sont définies à l'aide d'attributs. Un attribut est instance d'une classe particulière. Un ou plusieurs **descripteurs** sont associés à un attribut. Ces descripteurs permettent de définir le type de l'attribut, le type de la valeur de l'attribut, des méthodes de calcul pour cette valeur et des contraintes sur cette valeur.
La valeur d'un attribut peut être de type simple (entier, chaîne) ou de type complexe : tas, liste, ensemble, séquence.
Si la valeur d'un attribut d'une classe A est prise dans une classe B, alors des liens particuliers entre la classe A et la classe B peuvent être définis. Ces liens nous permettent, entre autre, de définir des objets composites.

Exemple :

Nous allons définir une classe Avion qui possède quatre attributs :
- imat : immatriculation de l'avion
- fuse : longueur du fuselage
- atte : distance d'atterrissage
- deco : distance de décollage

Avion
nomclasse = avion
propriété =
imat : *un* chaîne
sorte-de attribut-obligatoire
fuse : *un* entier
atte : *un* entier
contrainte
(contr (atte))
deco : *un* entier
inférence
(prop (fuse))
(super (deco atte)) : *sorte-de* contrainte

Les mots en italique sont des descripteurs.

Le descripteur "*un* " représente le type de la valeur de l'attribut. Par exemple, la longueur du fuselage doit être un entier.

Le descripteur "*sorte-de* " permet de connaître le type de l'attribut, il détermine la classe d'appartenance d'un attribut. Si le descripteur "*sorte-de* " n'est pas spécifié, alors l'attribut

est par défaut instance de la classe Attribut. Imat est un attribut appartenant à la classe attribut-obligatoire.

Le descripteur "*contrainte* " permet de définir des contraintes sur un attribut. (contr (atte)) est une contrainte portant sur l'attribut atte.

Le descripteur "*inférence* " associe à un attribut une ou plusieurs inférences qui permettent de calculer la valeur de cet attribut. Ces inférences peuvent utiliser des méthodes. Ces méthodes appellent des programmes définis par l'utilisateur, ces programmes peuvent être écrits dans différents langages. (prop (fuse)) est une inférence qui permet de calculer la distance de décollage (valeur de l'attribut deco) et qui utilise la méthode "prop". Cette méthode a pour argument l'attribut fuse.
Les méthodes sont organisées en graphe, le système choisit la méthode la plus adéquat pour un calcul.
Il est possible de définir des inférences inter-attributs. Ces inférences permettent d'instancier plusieurs attributs à la fois, ou peuvent utiliser des méthodes réversibles (le résultat de l'inférence n'est pas toujours le même argument).

L'utilisateur a la possibilité de définir des contraintes inter-attributs : celles-ci portent sur plusieurs attributs. Ces contraintes sont définies en dehors des attributs, pour éviter de les répéter au niveau de chaque attribut concerné. (super (deco atte)) est une contrainte inter-attributs.

Pour un attribut plusieurs inférences peuvent être définies. Un ordre partiel est instauré entre ces inférences. Par ordre partiel, nous entendons aussi ordre total et non ordonné qui sont des cas particuliers d'ordre partiel.

Exemple :

```
Avion
    nomclasse     = avion
    propriété     =
          imat    : un chaîne
                    sorte-de attribut-obligatoire
          fuse    : un entier
          atte    : un entier
          deco    : un entier
                    inférence
                       (prop (fuse))
                       (fonc (atte))
```

Deux inférences sont définies pour calculer la valeur de l'attribut deco. La valeur peut être calculée soit à l'aide de l'inférence (prop (fuse)) soit à l'aide de l'inférence (fonc (atte)).
Si l'un des deux arguments fuse ou atte des inférences est instancié, aucun problème ne se pose, car la valeur de l'attribut deco est calculée par l'inférence qui a son argument instancié. Par exemple si seul l'argument fuse est instancié le système calcule la valeur de l'attribut à partir de l'inférence (prop (fuse)); si seul l'argument atte est instancié le système calcule la valeur de l'attribut à partir de l'inférence (fonc (atte)).

Si les deux arguments atte et fuse sont instanciés, la valeur de l'attribut deco peut être donnée par la méthode prop ou la méthode fonc. Si un ordre est défini entre ces inférences, alors le système se base sur cet ordre. Si, par exemple, l'inférence (prop (fuse)) est prioritaire sur l'inférence (fonc (atte)), la valeur de l'attribut est alors calculée par la méthode prop. L'ordre peut être complété dans la couche de manipulation.

1.5 2 La couche de manipulation

Cette couche permet de manipuler les objets à partir de la connaissance détenue sur ces objets. Cette couche comprend la partie interface avec l'utilisateur.

C'est à ce niveau que l'on résout le problème de l'ordre partiel des inférences d'un attribut. En effet, si aucun ordre n'est défini entre des inférences au niveau de la représentation, c'est lors de la manipulation que l'utilisateur spécifie quelle est l'inférence à utiliser.
L'utilisateur peut aussi choisir de n'utiliser aucune inférence et de donner lui-même une valeur à l'attribut.
L'utilisateur peut aussi décider de garder toutes les valeurs possibles de l'attribut, c'est-à-dire que l'on calcule la valeur avec chaque inférence. Ceci permet de faire du raisonnement non monotone.

1.5.3 Situation de l'étude dans Sherpa

Cette étude se situe au niveau du modèle minimal SHOOD. Elle porte plus particulièrement sur la dynamique des instances (gestion de l'incomplétude et de l'incohérence) et sur la dynamique des classes. Pour le mémoire et le prototype implanté, nous n'avons pas retenu tous les concepts de SHOOD, mais uniquement ceux qui avaient un impact sur la dynamique des instances et des classes.

Le modèle minimal et l'étude de la dynamique des instances et des classes ont évolué ensemble durant l'année 1989 et sont aujourd'hui entièrement compatibles.

Des réunions de travail avec le laboratoire de biométrie de Lyon nous ont permis de valider les évolutions successives de notre modèle à travers une application portant sur le génome humain.

2 DYNAMIQUE DES INSTANCES

2.1 Les besoins

Les systèmes présentés précédemment s'intéressent peu ou pas à la **dynamique** des instances, c'est-à-dire à la gestion et au contrôle de l'évolution des instances. A un instant T, un objet peut ne pas avoir tous ses attributs instanciés, l'objet est **incomplet**, on parle alors d'**incomplétude**; une ou plusieurs contraintes définies sur cet objet peuvent ne pas être respectées, on parle alors de l'**incohérence** de l'objet.

Dans le cadre d'applications de Conception Assistée par Ordinateur (CAO), par exemple, la gestion de l'incomplétude et de l'incohérence des objets est à considérer. En effet, lors de la conception d'un objet celui-ci n'est complet et cohérent que dans sa phase finale [RIE86].
La cohérence et la complétude d'un objet permettent de déterminer l'état de l'objet au cours de la conception. On distingue quatre cas :

- l'objet est incomplet et cohérent, ce qui veut dire que la conception progresse.
- l'objet est incomplet et incohérent, ce qui veut dire que la conception ne s'améliore pas.
- l'objet est complet et incohérent, ce qui veut dire que la conception de l'objet est erronée.
- l'objet est complet et cohérent ce qui veut dire que la conception est terminée et correcte.

Dans les systèmes (Orion [BAR87], GemStone [PEN87]), où l'incomplétude est prise en compte, l'utilisateur peut créer des objets sans instancier tous les attributs, ceux-ci prennent alors la valeur nil; ces objets particuliers ne sont pas gérés mais sont simplement "estampillés". Quant à l'incohérence, bien souvent elle est interdite : un objet ne peut être créé s'il ne respecte pas toutes les contraintes de sa classe.
Si nous tolèrons des objets incohérents et incomplets, c'est que nous considèrons que cet état n'est que transitoire et que ces objets vont évoluer pour devenir complets et cohérents. Il est alors intéressant d'aider l'utilisateur en constatant l'incomplétude et l'incohérence et surtout en gérant l'évolution de l'objet. Les systèmes existants traitent uniquement le premier aspect. En effet, une estampille ne fait que constater l'état d'un objet.

Nous proposons de nous intéresser à ces deux aspects de la dynamique des instances.

Nous gérons l'incomplétude et l'incohérence des instances à l'aide de **structures** dites "significatives". Ces structures sont construites à partir des classes existantes et reflètent tous les états possibles d'une instance, en tenant compte évidemment des instances incomplètes et/ou incohérentes. Une instance doit toujours appartenir à une structure significative. Ces structures sont créées à partir de **règles d'optimisation** [RIE87].
Pour gérer l'évolution des instances, nous considérons une **relation d'ordre** entre les structures. Si, lors d'une mise à jour, l'évolution d'un objet satisfait la relation d'ordre, nous considérons que celui-ci s'**améliore**. La notion d'amélioration d'un objet peut être affinée selon les applications, l'utilisateur peut en effet spécifier ses propres **règles opératoires**. L'utilisateur peut, par exemple, décider qu'un objet s'améliore si la valeur d'un attribut X augmente ou diminue. Si lors de la mise à jour d'un objet celui-ci régresse, nous considérons ce fait comme anormal et créons alors une boucle de dégradation. Cette boucle est constituée d'un ensemble de versions de l'instance.

2.2 Dynamique des instances dans CADB

2.2.1 Quelques spécificités du modèle CADB

Au début de notre étude, la définition du modèle SHOOD n'était pas arrêtée. L'idée de structure étant apparue à partir du modèle **CADB** [NGU87, RIE87], nous avons débuté l'étude à partir de ce système [RIE85].

CADB est un système de gestion de base de données dédié à la CAO. Les fonctionnalités de CADB sont: la modélisation des objets, l'aide à la conception, l'utilisation et la maintenance de connaissances expertes, et la manipulation de concepts de haut niveau. Ce système est formé de deux composants: une base de données commune à un ensemble de concepteurs, et une base déductive qui est utilisée pour le contrôle automatique de la cohérence et le calcul des effets de bord lors de la mise à jour des informations. Notre étude se place au niveau de la base déductive.

Une classe dans CADB est définie par des règles de spécifications, des liens et des contraintes d'intégrité.

Une **règle de spécification** est rattachée à un attribut. Elle définit le type de l'attribut. Si un attribut est soumis uniquement à une règle de spécification, la valeur de cet attribut est donnée par l'utilisateur.

Un **lien** est une règle de déduction rattachée à un attribut. Ce lien permet de calculer la valeur de l'attribut. La valeur déduite par cette règle est une valeur sûre. Nous avons donc, soit pas de valeur, soit une valeur calculée par cette règle. L'attribut a une valeur quand tous les arguments du lien sont instanciés, sinon il n'a pas de valeur. Quand un lien apparaît au niveau d'un attribut, l'utilisateur ne peut intervenir pour donner une valeur à celui-ci.

Une **contrainte d'intégrité** est une condition que doit respecter l'objet. Cette condition porte sur un ou plusieurs attributs de l'instance. La valeur retournée par la contrainte est booléenne (vrai ou faux). Une contrainte n'est pas rattachée directement à un attribut.

Les liens et les contraintes d'intégrité possèdent des arguments. Un lien ou une contrainte sont dits **décidables** si tous leurs arguments sont instanciés.

Pour manipuler des objets, il faut pouvoir les identifier. Les identifiants d'objets sont de deux sortes : les **identificateurs** et les **clés**. Ces identifiants doivent être uniques et peuvent être donnés soit par le système soit par l'utilisateur. Un identificateur permet d'identifier un objet indépendamment des valeurs de ses attributs. Une clé est déterminée par les valeurs d'un ou plusieurs attributs d'un objet. CADB supporte les deux notions: identificateurs et clés. Nous ne nous intéressons qu'à la notion de clé, celle-ci portant sur un seul attribut.

Exemple:

Avion
- imat : chaîne
- aile : Aile
- fuse : entier
- atte : entier
- deco : entier
- deco := prop (fuse)
- super (deco, atte)

Cet exemple modélise la classe Avion.

Les propriétés de la classe Avion sont:
imat : qui est l'immatriculation d'un avion; le type est une chaîne de caractères. Cet attribut est la clé de la classe Avion. Deux avions différents ne peuvent pas avoir la même valeur pour l'attribut imat.
aile : qui prend ses valeurs dans la classe AILE, l'attribut aile donne les caractéristiques d'une aile.
fuse : qui représente la longueur du fuselage et doit être un entier.
atte : qui est la distance d'atterrissage et doit être un entier.
deco : qui est la distance de décollage et qui est proportionnelle à la longueur du fuselage; la fonction "prop (fuse)" permet de calculer la distance de décollage qui est égale à la longueur du fuselage multipliée par 50.
super : qui est une contrainte d'intégrité et qui spécifie que la distance de décollage doit être supérieure à la distance d'atterrissage. Si la valeur de l'attribut "deco" est supérieure à la valeur de l'attribut "atte" la contrainte super a pour valeur : vrai, sinon sa valeur est : faux.

Nous avons quatre propriétés soumises uniquement à des règles de spécification: imat, aile, fuse, atte; une propriété soumise au lien prop(fuse): deco; et une contrainte: super (deco, atte).

Imat, aile ... représentent les noms des attributs; entier, Aile ... représentent les types des attributs. Ces types peuvent être simples (ex. entier), ou peuvent faire référence à une classe complexe (ex. Aile). Pour donner une valeur à l'attribut aile, il faut instancier toutes les propriétés de la classe Aile.

Le lien "prop" possède un argument: fuse; la contrainte d'intégrité "super" a deux arguments: deco et atte.

2.2.2 Création des structures

Une **structure significative** est construite à partir d'une classe. Elle représente un état possible d'un objet. Par exemple, nous créons une structure pour les avions sans aile, une autre pour les avions dont on ne connaît pas la longueur du fuselage, etc ...

Une instance particulière est rattachée à une structure et une seule. C'est la structure maximale qui contient toutes les propriétés satisfaites par l'objet. Par exemple, un avion dont on ne

connaît que la distance d'atterrissage, appartient à la structure dont seul l'attribut "atte" est mentionné.

A partir d'une classe initiale, nous générons toutes les structures potentielles des sous-objets qu'elle peut contenir. A chaque classe est donc associé un ensemble de structures significatives.
Ces structures sont générées automatiquement à la création d'une classe.

Ces structures sont définies par leurs règles de spécification, leurs liens et leurs contraintes d'intégrité.

Dans un système, nous distinguons les classes de base (classe des entiers, classe des réels ...) et les classes construites (classes créées pour une utilisation donnée). Les structures ne sont élaborées que pour les classes construites.

Nous remarquons que structures et classes sont définies de manière similaire. Nous avons adopté cette solution pour avoir un modèle homogène et pouvoir, si besoin est, transformer une structure en classe (chapitre 3, paragraphe 3.2.2). Ces structures ne font pas partie du graphe d'héritage car nous voulons rendre la gestion des instances transparente à l'utilisateur; de plus cela conduirait à un graphe d'héritage rapidement important et complexe.

Pour construire ces structures, une solution consisterait à déterminer un algorithme de fermeture, c'est-à-dire à retenir toutes les combinaisons possibles entre les attributs. Cette solution est coûteuse en temps et en place car elle nous donne un nombre important de structures. De plus, certaines de ces structures ne représentent aucune réalité. Par exemple, il est inutile de concevoir la structure contenant la contrainte super mais ne contenant pas l'attribut "atte", car la contrainte super ne peut pas être évaluée.

Nous utilisons des **règles d'optimisation** qui permettent de restreindre la fermeture à un ensemble de structures dites **significatives** correspondant à une réalité possible. Seules sont considérées comme légales les structures ainsi obtenues. Ces règles **heuristiques** sont indépendantes des applications, mais dépendent du modèle de données. Nous verrons au paragraphe 2.3.4 qu'elles sont différentes pour SHOOD. Le nombre de structures créées étant réduit, la détermination automatique de la structure d'un objet lors de sa création puis de sa modification est accélérée.

Ces règles d'optimisation sont les suivantes:

- R1: Toute structure doit contenir les contraintes de type associées aux propriétés des objets qu'elle définit.
- R2: Toute structure doit comporter la spécification du nom de ses instances.
- R3: Toute structure comportant des contraintes ou des liens doit aussi comporter la spécification de leurs arguments.
- R4: Toute structure doit contenir toutes ses contraintes décidables.
- R5: Toute structure doit comporter tous ses liens décidables.
- R6: Toute structure doit comporter ses seuls liens définis.
- R7: Toute structure doit comporter pour chaque lien la fonction qui le définit.

R1: spécifie que l'on ne peut avoir une propriété non typée. Ceci permet un contrôle à priori des valeurs des attributs. A chaque modification ou création d'une instance, le type est vérifié systématiquement.

R2: Toute instance d'une structure doit pouvoir être identifiée. Donc la structure significative minimale est celle qui contient uniquement l'attribut clé. En effet, on ne peut pas manipuler une instance si elle ne possède pas d'identifiant.

R3: Une structure ne contient des contraintes ou des liens que si les arguments de ceux-ci sont instanciés. On ne peut en effet vérifier une contrainte ou calculer un lien, si l'on ne connaît pas la valeur de chacun des arguments.

R4: Si tous les arguments d'une contrainte sont instanciés, alors cette contrainte doit être spécifiée dans la structure. Cela permet une **vérification immédiate de la cohérence.**

R5: Cette règle est similaire à la règle R4, mais s'applique au lien. Cela permet une **déduction immédiate de l'information.**

R6: On ne peut spécifier dans une structure les négations des liens d'une classe. Comme un lien définit une valeur sûre, on ne peut qu'interdire sa négation. On ne peut pas avoir une structure contenant l'attribut deco associé à "not prop (fuse)". Mais pour une contrainte, on peut définir une structure comportant sa contrainte inverse, cela permet de tolérer l'incohérence.

R7: Cette règle élimine les structures comportant des propriétés calculées sans qu'apparaissent leurs formules de calcul. Par exemple, la fonction "prop (fuse)" doit toujours apparaître avec l'attribut "deco".

La notion d'objet incomplet est prise en compte puisque toutes les structures significatives possibles sont générées. La notion d'objet incohérent aussi, puisque la négation d'une contrainte détermine une structure au même titre que la contrainte elle-même. Donc, quelle que soit une instance créée par un utilisateur, il existe toujours une structure lui correspondant.

La **structure minimale** d'une classe est la structure contenant uniquement l'attribut clé.

La **structure maximale** d'une classe est la structure possédant les mêmes propriétés que la classe.

Exemple:

Les structures significatives issues de la classe Avion sont les suivantes:

C1 est la structure contenant les avions dont on ne connaît que le nom. C'est la structure minimale.
C1={imat : chaîne}

C2 est la structure contenant les avions dont on ne connaît que le nom, et la valeur de l'attribut aile.
C2={imat : chaîne, aile : Aile}

C3 est la structure contenant les instances dont on connaît le nom, la longueur du fuselage. On doit alors calculer la distance de décollage (règle R5)
C3={imat : chaîne, fuse : entier, deco : entier, deco := prop (fuse)}

C4 est la structure contenant les avions dont on ne connaît que la distance d'atterrissage
C4={imat : chaîne, atte : entier}

C5 est la structure contenant les avions dont on connaît le nom, les ailes, la longueur du fuselage; le lien deco est alors calculé.
C5={imat : chaîne, aile : Aile, fuse : entier, deco : entier, deco := prop (fuse)}

C6 est la structure contenant les avions pour lesquels seuls, le nom, les ailes et la distance d'atterrissage sont connus.
C6={imat : chaîne, aile : Aile, atte : entier}

Dans la structure C7, on connaît le nom, la longueur du fuselage, la distance d'atterrissage. On calcule la distance de décollage, et cette distance est supérieure à la distance d'atterrissage.
C7={imat : chaîne, fuse : entier, atte : entier, deco : entier, deco := prop (fuse),
super (deco, atte)}

La structure C8 est identique à la structure C7, sauf pour la contrainte. Dans cette structure la contrainte "super" est violée. La distance de décollage est inférieure à la distance d'atterrissage.
C8={imat : chaîne, fuse : entier, atte : entier, deco : entier, deco := prop (fuse),
notsuper (deco, atte)}

La structure C9 est la structure maximale, elle contient exactement les propriétés de la classe. Les instances appartenant à cette structure sont complètes et cohérentes.
C9={imat : chaîne, aile : Aile, fuse : entier, atte : entier, deco : entier, deco := prop (fuse),
super (deco, atte)}

Les instances appartenant à la structure C10 sont complètes, mais non cohérentes, car la contrainte "super" est violée
C10={imat : chaîne, aile : Aile, fuse : entier, atte : entier, deco : entier, deco := prop (fuse),
notsuper (deco, atte)}

Nous avons généré, à partir de la classe Avion, dix structures significatives. Si nous avions appliqué un algorithme de fermeture sans règle d'optimisation nous aurions obtenu 2^{12} structures, correspondant à la fermeture des six propriétés de Avion, ainsi que leurs négations.

Si tous les arguments d'une contrainte ne sont pas instanciés, on ne peut pas déterminer si elle est vraie ou fausse. Nous considérons une contrainte non décidable comme respectée. Ceci permet de prendre en compte l'incomplétude des objets dans l'évaluation de leur cohérence sans bloquer leur conception ascendante.

La structure C8 a pour instance des objets incomplets et incohérents. Pour les structures C1 à C7 les objets sont incomplets, mais cohérents car nous ne pouvons pas déterminer si la contrainte "super" est respectée ou non.

Pour chaque objet nous définissons un **degré de complétude** et un **degré de cohérence**.

Soit N le nombre de propriétés d'une classe. Un objet de cette classe est P-complet ou complet de degré P, s'il possède P propriétés instanciées ($0 \leq P \leq N$). Son degré de complétude est P.

Définition: Un objet est complet si toutes les propriétés de sa classe sont instanciées. Il est N-complet.

Par exemple, les instances de la structure C9 sont 6-complet. Les objets appartenant à la structure C9 sont complets. Les instances de la structure C6 sont 3-complet, les objets appartenant à cette structure ne sont pas complets.

Soit M le nombre de contraintes définies sur un objet. Un objet est dit L-cohérent ou cohérent de degré L, s'il possède L contraintes indécidables ou satisfaites. L'objet possède donc (M - L) contraintes non satisfaites. Son degré de cohérence est L.

Définition: Un objet est cohérent s'il ne viole aucune contrainte décidable. Il est M-cohérent.

Par exemple, les instances des structures C8 et C10 sont 0-cohérent, toutes les autres instances sont 1-cohérent. En effet, dans les structures C1 à C7 la contrainte super est indécidable (on ne connaît pas tous les arguments de cette contrainte), les objets appartenant à ces structures sont alors considérés comme cohérents.

D'un point de vue pratique, nous conservons les degrés de cohérence et de complétude au niveau des structures, car tous les objets d'une structure ont les mêmes degrés de cohérence et de complétude. Les degrés d'une instance sont les degrés de la structure à laquelle elle est rattachée. Nous parlerons par abus de langage des degrés de cohérence et de complétude des structures.
Nous adopterons la syntaxe suivante : Ci {p l} où Ci est le nom de la structure, p le degré de complétude des objets de la structure Ci et l le degré de cohérence des objets de la structure Ci.

2.2.3 Création des instances

Une instance doit toujours appartenir à une structure légale et une seule.

L'instance est rattachée directement à la structure contenant toutes les propriétés qu'elle satisfait. Dans les systèmes traditionnels, l'instance est rattachée à sa classe; dans le cas d'incomplétude les attributs non instanciés prennent la valeur "nil". Dans notre système, l'instance instancie toutes les propriétés de la structure. Même dans les cas d'incohérence, l'instance vérifie les contraintes de la structure (même si celles-ci sont contraires aux contraintes de la classe).

Exemple:

Créons un avion A, dont on ne connaît pas la longueur du fuselage.
Avion A
imat : A
aile : 10
atte : 400
L'avion A est rattaché à la structure C6. La valeur de l'attribut deco ne peut pas être calculée, et la contrainte super est non décidable. Cette instance est incomplète, mais cohérente.

Créons un deuxième avion B.
Avion B
imat : B
aile : 30
fuse : 4
atte : 250
Le système calcule la valeur de l'attribut "deco" qui est 200. Puis il vérifie la contrainte "super", celle-ci s'avère non respectée. Cette instance appartient à la structure C10. La contrainte dans la structure C10 étant "notsuper", l'instance B respecte bien cette contrainte.

Il est à noter que les contraintes ne sont pas gardées au niveau de l'instance, mais uniquement au niveau des structures. En effet, pour une même structure toutes les instances ont les mêmes valeurs pour les contraintes. Nous constatons donc, un gain de place au niveau des instances.

2.2.4 Evolution des instances

Le rattachement d'une instance, lors de sa création, à une structure permet de déterminer l'état de complétude et de cohérence de cette instance. Si l'instance est modifiée, son état évolue et sa structure de rattachement peut changer. **Gérer l'évolution** de l'objet nécessite non seulement de prendre en compte son état initial et son état final mais aussi la **transition** entre la structure initiale et la structure finale.

Pour gérer l'évolution, nous nous basons sur les structures. Nous allons, en effet, établir une **relation d'ordre** entre ces structures; cette relation permet de savoir si une instance évolue correctement (c'est-à-dire sans dégradation) ou non.

La relation d'ordre entre les structures est définie comme suit: une structure S1 est strictement supérieure à une structure S2 si l'un des degrés des objets de S1 est supérieur ou égal au degré des objets de S2, et si l'autre degré des objets de S1 est strictement supérieur au degré des objets de S2. Une structure S1 est égale à une structure S2 si les degrés de complétude et de cohérence des objets de S1 sont égaux à ceux des objets de S2.

Soit:
p1 degré de complétude des objets de S1
p2 degré de complétude des objets de S2
l1 degré de cohérence des objets de S1
l2 degré de cohérence des objets de S2

on a:

$S1 > S2 \iff p1 > p2 \text{ et } l1 \geq l2 \text{ ou } p1 \geq p2 \text{ et } l1 > l2$

$S1 = S2 \iff p1 = p2 \text{ et } l1 = l2$

Dans cette relation d'ordre nous considérons que la complétude et la cohérence ont la même influence sur une instance. C'est-à-dire que le concept de complétude n'est pas plus fort que celui de cohérence et inversement. Des structures sont alors non comparables. Les structures S1 et S2 ne sont pas comparables dans les cas suivants :

$l1 > l2$ et $p1 < p2$ ou $l1 < l2$ et $p1 > p2$.

En effet, les degrés de cohérence et de complétude varient en sens inverse.
Pour pallier cet inconvénient nous aurions pu privilégier l'un ou l'autre des degrés. Mais, aucun élément ne nous permet de faire un tel choix, c'est pourquoi complétude et cohérence ont le même impact sur la relation d'ordre.

Reprenons les structures formées à partir de la classe AVION, nous avons :
C1 {1 1), C2 {2 1}, C3 {3 1}, C4 {2 1}, C5 {4 1}, C6 {3 1}, C7 {5 1}, C8 {5 0},
C9 {6 1}, C10 {6 0}
Quelques exemples de relation d'ordre:
La structure C2 est supérieure à la structure C1.
La structure C3 est égale à la structure C6.
Les structures C1 et C10 ne sont pas comparables.

Une instance **s'améliore** si après mise à jour, elle reste dans la même structure ou si elle passe dans une structure de niveau supérieur ou égal. Dans le cas où les deux structures ne sont pas comparables, nous considérons qu'il n'y a pas d'amélioration puisque l'on assiste à une diminution du degré de complétude ou du degré de cohérence. Nous constatons que l'amélioration d'une instance dépend des degrés de cohérence et de complétude des structures.

Nous décomposons les opérations de mise à jour en deux étapes : d'une part la **certification**, d'autre part la **validation.**

Une opération est **certifiée** si l'instance **s'améliore**. En effet, dans ce cas son degré de cohérence et/ou de complétude augmente. Une mise à jour est donc non certifiée dans les cas où l'objet ne respecte plus une ou plusieurs contraintes de la classe, ou/et si un ou plusieurs attributs de la classe ne sont plus instanciés.

La **validation** représente la mise à jour **physique** des modifications.

Si l'opération est certifiée, elle est automatiquement validée; sinon l'utilisateur peut ou non choisir de la valider.

Les données ne sont donc modifiées qu'après validation. Ceci permet une grande souplesse de mise à jour, car l'utilisateur peut faire des essais de modifications d'attributs, sans entraîner une mise à jour physique de l'instance dans le cas où ces modifications ne seraient pas pertinentes. Dans les systèmes classiques ces deux étapes sont confondues.

Exemple:

Modifions l'avion A de l'exemple précédent, en donnant la valeur 3 à l'attribut "fuse". L'instance avion A passe de la structure C6 {3 1} à la structure C10 {6 0}, ces deux structures ne sont pas comparables, et nous considérons alors que l'avion A ne s'améliore pas; donc cette opération n'est pas certifiée. C'est à l'utilisateur de décider de valider ou non cette mise à jour.

Une mise à jour de l'avion B peut être l'instanciation de l'attribut "atte" par la valeur 150. La contrainte "super" est alors vérifiée. L'instance passe de la structure C10 {6 0} à la structure C9 {6 1}. La structure C9 est supérieure à la structure C10, donc l'opération est certifiée et validée automatiquement.

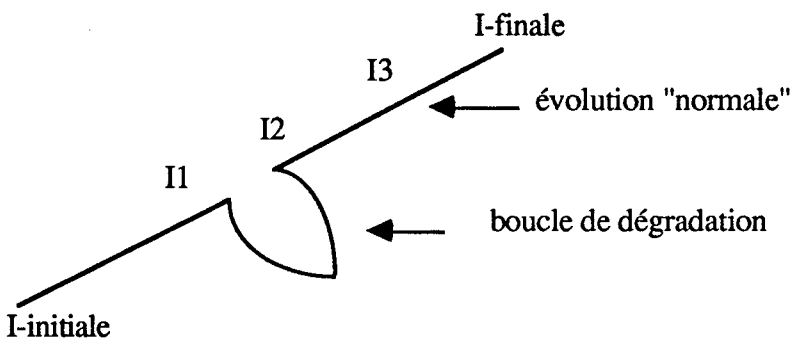
2.2.5 Les boucles de dégradation

Nous autorisons l'utilisateur à valider une opération non certifiée, car nous supposons que cet état de l'instance n'est pas définitif. Nous générons, dans ce cas, une **boucle de dégradation**. La boucle de dégradation d'une instance est constituée d'un ensemble de **versions**. Chacune d'elles modélise l'instance après une modification.
Dès la première dégradation d'une instance, celle-ci est versionnalisée. Tant que la version courante de l'instance ne possède pas des degrés de cohérence et de complétude supérieurs ou égaux à la première version, le système génère à chaque modification validée une nouvelle version. Lorsque l'instance, après une série de modifications, est dans une structure supérieure ou égale à celle de la première version toute la boucle de dégradation est supprimée et l'instance n'est plus versionnalisée.

L'utilisateur peut à tout moment décider de repartir d'une version. La boucle de dégradation se transforme en un arbre de versions. En effet, à partir d'une version plusieurs versions peuvent alors être dérivées. Ceci permet à l'utilisateur de simuler des opérations de mise à jour.

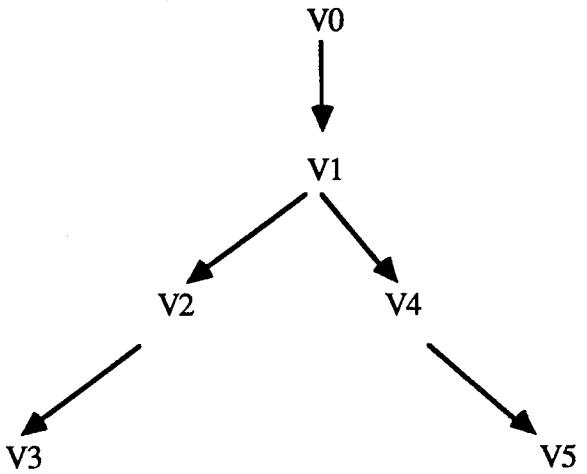
Si l'utilisateur ne désire aucune version pour une instance, il ne valide jamais une opération non certifiée.

Exemple de l'évolution d'une instance :



I-initiale, I1, I2, I3, I-finale correspondent aux différentes étapes de l'évolution d'une instance. De I-initiale à I1 l'instance s'améliore. Après I1, il y a dégradation de l'instance. L'instance revient dans le chemin "normal" (où l'instance s'améliore) en I2, puis s'améliore à chaque mise à jour pour aboutir à l'instance finale (I-finale) qui est complète et cohérente.

La boucle de dégradation de l'instance est un arbre de versions.



v0, v1 ... v5 représentent les différentes versions de l'instance dans la boucle de dégradation. Plusieurs modifications successives ont été faites à partir de la versions v0 : v1, v2, v3. Puis, l'utilisateur a décidé de repartir de la version v1, une autre branche de l'arbre est alors générée, comprenant les versions v4 et v5.

2.2.6 Les classes opératoires

Nous avons vu que la notion d'amélioration d'une instance dépendait des degrés de complétude et de cohérence des objets. Il nous a semblé pertinent d'autoriser l'utilisateur, pour une classe donnée, à **affiner** cette notion d'**amélioration** en spécifiant des critères d'amélioration supplémentaires. Ces critères sont appelés **règles opératoires,** car ils concernent les opérations de mise à jour. Ces critères peuvent soit porter sur des attributs de la classe : une instance de la classe AVION s'améliore si la longueur du fuselage augmente; soit être une fonction à appliquer sur un ou plusieurs attributs de la classe : la différence entre la distance de décollage et la distance d'atterrissage doit diminuer pour qu'un objet s'améliore.

Pour décider de la dégradation ou non d'une instance, on tient non seulement compte des degrés des structures, mais aussi de ces règles. Si l'une de ces règles n'est pas respectée, alors l'objet se dégrade.

L'affinage du concept d'évolution d'un objet permet de gérer l'amélioration d'un objet, même si celui-ci n'a pas changé de structure lors de sa mise à jour. Par exemple, même si après modification de l'attribut "fuse" les degrés de complétude et de cohérence de l'objet ne diminuent pas, l'opération n'est certifiée que si la nouvelle valeur de l'attribut fuse est supérieure ou égale à l'ancienne valeur.

Les règles opératoires, affinant l'amélioration des instances d'une classe C, sont exprimées dans une classe appelée **classe opératoire** de C. Nous exprimons les règles opératoires à l'intérieur d'une classe afin d'avoir un modèle unifié où tout est représenté sous forme d'objet.

Exemple :

La classe Avion.maj est la classe opératoire correspondant à la classe Avion.

Avion.maj
augmfuse : (supeg fuse)
dimdiff : (infeg (diffe deco atte))

Augmfuse (augmentation de la longueur du fuselage) et dimdiff (diminution de la différence) sont des noms d'attributs donnés par l'utilisateur. L'attribut augmfuse est soumis à une fonction "(supeg fuse)". C'est une fonction booléenne s'appliquant à l'attribut fuse et délivrant la valeur vrai si l'opération de mise à jour ne diminue pas la valeur de l'attribut fuse. La fonction "infeg" est appliquée à la fonction (diffe deco atte) qui calcule la différence entre la distance de décollage et la distance d'atterrissage. Si cette différence n'augmente pas, la fonction infeg renvoie le résultat vrai.
Donc pour qu'un avion s'améliore, il faut que ses degrés de complétude et de cohérence ne diminuent pas, que la longueur du fuselage reste stable ou augmente et que la différence entre la distance de décollage et d'atterrissage n'augmente pas.

La classe opératoire d'une classe C est créée par l'utilisateur, uniquement si celui-ci désire affiner le concept d'amélioration.
Les instances d'une classe opératoire vont nous permettre de justifier les dégradations d'une instance de la classe C. Ces instances sont gardées uniquement si l'instance de la classe C est versionnalisée. L'identifiant d'une instance d'une classe opératoire est un identificateur calculé par le système.

Nous aurions pu envisager une gestion du concept de l'amélioration en nous basant uniquement sur les classes opératoires.
Dans ce cas, les classes opératoires contiennent tous les critères pour qu'une instance s'améliore.
A la création d'une classe C, une classe opératoire associée est créée automatiquement par le système. Cette classe contient deux critères spécifiant l'un que le degré de complétude ne doit pas diminuer et l'autre que le degré de cohérence ne doit pas diminuer.
Si l'utilisateur désire ajouter d'autres critères d'amélioration (c'est-à-dire d'autres règles opératoires), il lui suffira d'ajouter de nouveaux attributs à la classe opératoire déjà existante.
Cette solution permet de traiter tous les critères d'amélioration de la même manière. Mais une grande place mémoire est requise pour son implémentation, car la création d'une classe C entraîne automatiquement la création d'une deuxième classe qui est la classe opératoire. A cause de cela, nous n'avons pas retenu cette solution.

2.3 Dynamique des instances dans SHOOD

Avec peu de modifications, les principes développés précédemment peuvent être étendus à SHOOD. Nous allons étudier les caractéristiques qui diffèrent de CADB, et voir l'impact de ces différences sur la création des structures. Nous rappelons que dans SHOOD, plusieurs descripteurs peuvent être associés à un attribut : type, contraintes, inférences.

Exemple :

```
Avion
    nomclasse     = avion
    propriété     =
          imat    : un chaîne
                    sorte-de attribut-obligatoire
          aile    : un entier
          fuse    : un entier
          atte    : un entier
                    contrainte
                        (contr (atte))
          deco    : un entier
                    inférence
                        (prop (fuse))
                        (fonc (aile))
          (super (deco atte)) : sorte-de contrainte
```

A l'attribut atte est associé un type : entier, et une contrainte : contr (atte). A l'attribut deco est associé un type : entier et deux méthodes d'inférence : prop (fuse) et fonc (aile) qui permettent de calculer la valeur de la distance de décollage soit en fonction du fuselage soit en fonction des ailes.

2.3.1 Les attributs obligatoires

Les instances dans SHOOD sont identifiées par un identificateur et éventuellement une ou plusieurs clés.
Une clé dans SHOOD peut être composée de plusieurs attributs. Ces attributs doivent être obligatoirement instanciés. Par extension, nous considérons d'autres **attributs obligatoires** n'appartenant pas à la clé. Ces attributs représentent la connaissance minimale que doit posséder l'utilisateur pour créer ou modifier une instance.
Par exemple, si la création de la classe Avion est faite surtout dans le but de gérer des statistiques sur la longueur du fuselage, il est intéressant de rendre obligatoire l'attribut "fuse", bien que celui-ci n'ait aucune raison de faire partie de la clé de la classe Avion.

Les structures doivent toutes comporter l'ensemble des attributs obligatoires. La structure minimale est donc la structure contenant l'ensemble des attributs obligatoires. Si une inférence ne contient comme arguments que des attributs obligatoires, alors l'attribut calculé à l'aide de cette inférence fait aussi partie de la structure minimale.

Exemple :

Avion
nomclasse = avion
propriété =
imat : *un* chaîne
sorte-de attribut-obligatoire
fuse : *un* entier
atte : *un* entier
sorte-de attribut-obligatoire
deco : *un* entier
inférence
(prop (fuse))
(super (deco atte)) : *sorte-de* contrainte

Les deux attributs obligatoires de cette classe sont : imat et atte
La structure minimale est donc :
C1 : {imat : chaîne, atte : entier}

Avion
nomclasse = avion
propriété =
imat : *un* chaîne
sorte-de attribut-obligatoire
fuse : *un* entier
sorte-de attribut-obligatoire
atte : *un* entier
deco : *un* entier
inférence
(prop (fuse))
(super (deco atte)) : *sorte-de* contrainte

Dans ce deuxième exemple, les attributs obligatoires sont imat et fuse. L'attribut deco est alors toujours calculable.
La structure minimale est :
C1 : {imat : chaîne, fuse : entier, deco : entier}

Nous remarquons que si la classe ne possède aucun attribut obligatoire, et aucun attribut clé (les instances sont manipulées à l'aide d'identificateurs), aucune structure minimale n'est créée. Dans CADB, nous avions toujours une structure minimale contenant l'attribut clé.

2.3.2 Les contraintes

Dans SHOOD nous distinguons deux sortes de contraintes :

- les contraintes **intra-attribut** qui sont définies au niveau d'un attribut, et portent sur ce seul attribut. Par exemple, nous pouvons contraindre l'attribut atte, en spécifiant que sa valeur doit être supérieure à 100.

- les contraintes **inter-attributs** qui sont définies en dehors des attributs et portent sur plusieurs attributs. La contrainte "super (deco atte)" est une contrainte inter-attributs qui porte sur les attributs deco et atte.

Le système traite ces deux sortes de contraintes de façon identique. L'intérêt de les séparer n'étant que d'éviter une répétition des contraintes inter-attributs au niveau de chaque attribut impliqué.

Dans CADB, nous avons introduit la notion de contraintes violables. Dans SHOOD, nous laissons l'utilisateur choisir si une contrainte doit toujours être respectée ou non. En effet, celui-ci peut déterminer des **contraintes fortes**, ces contraintes doivent toujours être respectées; ou des **contraintes faibles**, lors de la conception d'un objet, elles peuvent être non respectées.

Les descripteurs de type se comportent en contrainte forte. Lors de l'instanciation ou la mise à jour d'un attribut, la valeur de l'attribut doit toujours répondre au type spécifié, un contrôle systématique est effectué.

Dans CADB, pour chaque contrainte appartenant à une structure, nous élaborions une structure avec la contrainte inverse. Dans SHOOD, nous élaborons cette structure uniquement si la contrainte est faible. Dans le cas d'une contrainte forte, une seule structure est créée.

Exemple :

```
Avion
    nomclasse     = avion
    propriété     =
          imat    : un chaîne
                    sorte-de attribut-obligatoire
          fuse    : un entier
          atte    : un entier
                    contrainte
                      (contr (atte))
          deco    : un entier
                    inférence
                      (prop (fuse))
          (super (deco atte)) : sorte-de contrainte
```

Par défaut une contrainte est une contrainte forte.

Nous avons deux contraintes : contr (atte) qui est une contrainte faible intra-attribut, et super (deco atte) qui est une contrainte forte inter-attributs.

La contrainte contr(atte) peut être violée, nous pouvons donc avoir les structures:
Ci : {imat : chaîne, atte : entier, contr (atte)}
Cj : {imat : chaîne, atte : entier, notcontr (atte)} où notcontr est la négation de la contrainte contr.

La contrainte super est une contrainte forte.
Ck : {imat : chaîne, atte : entier, contr (atte), fuse : entier, deco : entier, super (deco atte)} est une structure possible.
Cl : {imat : chaîne, atte : entier, contr (atte), fuse : entier, deco : entier, notsuper (deco atte)} n'est pas une structure car la contrainte super ne peut être violée.

2.3.3 Les inférences

Nous avons vu que plusieurs inférences peuvent intervenir au niveau d'un même attribut, ceci permet de calculer la valeur de l'attribut à l'aide de différentes méthodes ayant des arguments différents.

L'ordre défini sur ces inférences est un ordre partiel.

Exemple :

```
Avion
    nomclasse     = avion
    propriété     =
          imat    : un chaîne
                    sorte-de attribut-obligatoire
          fuse    : un entier
          aile    : un entier
          deco    : un entier
                    inférence
                        (prop (fuse))
                        (fonc (aile))
```

Entre les inférences prop (fuse) et fonc (aile) nous pouvons définir un ordre total, en déterminant par exemple, que l'inférence prop (fuse) est prioritaire sur l'inférence fonc (aile); si les attributs fuse et aile sont instanciés la valeur de l'attribut deco est toujours calculée par l'inférence prop (fuse).
Si aucun ordre n'est défini entre ces inférences, l'utilisateur peut spécifier lors de la création d'une instance, quelle est l'inférence prioritaire. Nous pouvons donc avoir une instance A, où le calcul de l'attribut deco a été fait à l'aide de l'inférence prop (fuse), et une instance B où le calcul a été effectué à l'aide de fonc (aile).

Deux solutions peuvent être envisagées pour créer les structures. La solution 1 tient compte de la façon dont a été calculée la valeur de l'attribut, alors que la solution 2 n'en tient pas compte.

Solution 1 :
Ci : {imat : chaîne, aile : entier, fuse : entier, deco : entier, prop(fuse)}
Cj : {imat : chaîne, aile : entier, fuse : entier, deco : entier, fonc(aile)}
Deux structures sont créées, l'une où la valeur de l'attribut deco est calculée à l'aide de la fonction prop(fuse) et une où la valeur de l'attribut deco est calculée à l'aide de l'inférence fonc(aile).

Solution 2 :
Ck : {imat : chaîne, aile : entier, fuse : entier, deco : entier}
Dans cette solution nous ne précisons pas l'inférence qui a permis de calculer l'attribut deco, nous nous contentons de spécifier que l'attribut deco est instancié.

Dans la solution 1, nous remarquons que le nombre de structures peut être très important. En effet, si un attribut contient dix inférences, dix structures devront être créées. Dans le même cas de figure, si l'on adopte la solution 2, une seule structure est créée.

La solution 2 a été retenue. D'une part, pour une question d'optimisation, car le nombre de structures étant moindre, la recherche de la structure d'une instance, lors de sa création ou de sa modification, est plus rapide. D'autre part, la façon dont a été calculé un attribut n'intervient pas dans le calcul des degrés de complétude et de cohérence. La manière dont est calculé un attribut est gardée au niveau de l'instance et non pas au niveau de la structure.

L'utilisateur peut, à la création ou à la mise à jour d'un attribut, décider de ne pas utiliser les inférences spécifiées pour un attribut, mais donner une valeur autre à cet attribut. Il faut donc créer une structure contenant cet attribut même si les arguments de ses inférences ne sont pas instanciés.

Exemple :

```
Avion
    nomclasse     = avion
    propriété     =
          imat    : un chaîne
                    sorte-de attribut-obligatoire
          fuse    : un entier
          aile    : un entier
          deco    : un entier
                    inférence
                      (prop (fuse))
                      (fonc (aile))
```

Les structures suivantes sont des structures possibles de la classe Avion.
Ci : {imat : chaîne, deco : entier}
Cj : {imat : chaîne, fuse : entier, deco : entier}
Ck : {imat : chaîne, aile : entier, deco : entier}

2.3.4 Les règles d'optimisation

Nous avons vu que les règles d'optimisation dépendaient du modèle de données, pour SHOOD de nouvelles règles d'optimisation sont alors à définir. Ces règles s'appuient sur le même principe que les règles de CADB et tiennent compte des spécificités de SHOOD.

R1 : Toute structure doit contenir les contraintes de type associées aux propriétés des objets qu'elle définit.

R2 : Toute structure doit comporter l'ensemble des propriétés obligatoires.

R3 : Toute structure comportant des contraintes (faibles ou fortes) doit comporter la spécification de leurs arguments.

R4 : Toute structure doit contenir toutes ses contraintes décidables.

R5 : Toute structure doit comporter les attributs possédant des inférences décidables.

R6 : Toute structure doit comporter ses seules inférences et contraintes fortes.

R1 est identique à la règle R1 de CADB. Les attributs sont toujours fortement typés.

R2 : Cette règle est plus générale que la règle R2 de CADB, car elle tient compte de tous les attributs obligatoires (clé ou autres).

R3 : Quand une contrainte apparaît dans une structure, ses arguments doivent être instanciés. Dans CADB, cette règle s'appliquait également aux inférences. Ici, seules les contraintes sont concernées, car l'utilisateur peut donner une valeur à un attribut indépendamment des inférences définies sur cet attribut.

R4 : Cette règle est identique à la règle R4 de CADB. Si une contrainte peut être évaluée, elle est évaluée.

R5 : Si un attribut peut être calculé par une inférence dont les arguments apparaissent dans la structure, alors cet attribut est cité dans la structure.

R6 : La règle R6 s'applique non seulement aux inférences, mais aussi aux contraintes fortes. En effet, celles-ci doivent toujours être respectées.

Nous pouvons remarquer l'absence de la règle R7 de CADB, car la formule de calcul d'un attribut n'est pas mentionnée dans les structures.

Exemple :

Nous allons appliquer ces nouvelles règles pour la construction des structures d'une classe Avion.

Avion
nomclasse = avion
propriété =
imat : *un* chaîne
sorte-de attribut-obligatoire
fuse : *un* entier
atte : *un* entier
contraintv
(contr (atte))
deco : *un* entier
inférence
(prop (fuse))
(fonc (aile))
(super (deco atte)) : *sorte-de* contrainte

C1 : {imat : chaîne}

C2 : {imat : chaîne, aile : entier, deco : entier}

C3 : {imat : chaîne, fuse : entier, deco : entier}

C4 : {imat : chaîne, deco : entier}

C5 : {imat : chaîne, atte : entier, contr (atte)}

C6 : {imat : chaîne, atte : entier, notcontr (atte)}

C7 : {imat : chaîne, aile : entier, fuse : entier, deco : entier}

C8 : {imat : chaîne, aile : entier, atte : entier, contr (atte), deco : entier, super (deco atte)}

C9 : {imat : chaîne, aile : entier, atte : entier, notcontr (atte), deco : entier, super (deco atte)}

C10 : {imat : chaîne, fuse : entier, deco : entier, atte : entier, contr (atte), super (deco atte)}

C11 : {imat : chaîne, fuse : entier, deco : entier, atte : entier, notcontr (atte),
super (deco atte)}

C12 : {imat : chaîne, deco : entier, atte : entier, contr (atte), super (deco atte)}

C13 : {imat : chaîne, deco : entier, atte : entier, notcontr (atte), super (deco atte)}

C14 : {imat : chaîne, aile : entier, fuse : entier, deco : entier, atte : entier, contr (atte), super (deco atte)}

C15 : {imat : chaîne, aile : entier, fuse : entier, deco : entier, atte : entier, notcontr (atte), super (deco atte)}

Nous pouvons remarquer la structure C4, où l'attribut deco apparaît seul, ceci permet à l'utilisateur de donner une valeur à l'attribut deco sans que les attributs aile et fuse ne soient instanciés.
Dans la structure C7, les attributs aile et fuse sont instanciés, l'attribut deco est cité sans que sa formule de calcul ne soit donnée. C'est au niveau de l'instance que nous savons si cette valeur est donnée par l'utilisateur ou par l'inférence (prop (fuse)) ou par l'inférence (fonc (aile)).
Comme la contrainte (contr (atte)) peut être non respectée, nous constatons que la contrainte inverse (notcontr (atte)) apparaît dans des structures C15, C13, C11, C9, C6.
La contrainte (notsuper (deco atte)) n'apparaît jamais, car (super (deco atte)) a été déclarée contrainte forte dans la classe Avion.

Le degré de **complétude** des instances d'une structure est calculé en fonction du nombre d'**attributs** instanciés par l'objet, et du nombre de **contraintes inter-attributs** pouvant être évaluées. Les contraintes intra-attribut ne sont pas comptées car elles correspondent à un attribut, donc si cet attribut apparaît dans une structure, la contrainte associée apparaît aussi. Il serait, en effet, redondant de compter, par exemple, l'attribut "atte" et la contrainte "contr"; dans ce cas seul l'attribut "atte" intervient pour le calcul du degré de complétude. Par exemple, la structure C14, qui est la structure maximale, est de degré de complétude 6.

Le degré de **cohérence** tient compte des **contraintes** intra et inter-attributs qui sont **faibles**. Les contraintes fortes n'entrent pas dans le calcul du degré de cohérence, car le nombre de ces contraintes est toujours inchangé quelque soit la structure. Par exemple, si une classe a deux contraintes fortes et une contrainte faible, le degré de cohérence des structures ne peut varier que de "2" à "3". Nous aurons toujours au minimum "2" comme degré de cohérence. L'information intéressante est la différence entre le degré minimal et le degré maximal, et ce nombre correspond au nombre de contraintes faibles.
Le degré de cohérence des objets de la structure C14 est: 1.

Le contrôle de l'instanciation d'un attribut obligatoire et des contraintes fortes est fait au moment de la mise à jour d'un attribut. Une instance correspond donc toujours à une structure.

La gestion de l'évolution d'une instance est identique à la gestion instaurée dans CADB.

2.4 Conclusion

L'utilisation de structures a permis de répondre aux besoins de la CAO en gérant l'incomplétude et l'incohérence des instances. Nous avons optimisé la création des structures en définissant des

règles d'optimisation. Les structures permettent aussi un gain de place au niveau des instances : les contraintes ne sont pas gardées au niveau des instances et un attribut non instancié n'apparaît pas dans les instances.

La solution proposée est souple, puisque l'utilisateur peut définir ses propres critères d'amélioration et qu'il peut revenir sur la conception d'un objet en repartant d'une ancienne version. La décomposition de la mise à jour des instances en deux étapes : certification et validation, concourt également à aider l'utilisateur dans sa conception.

De plus, nous voyons que le changement de modèle de données n'affecte que la création des structures. La gestion de l'incomplétude et l'incohérence définie dans CADB est donc **facilement adaptable** à un autre modèle.

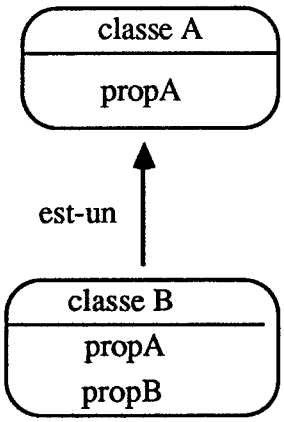
3 DYNAMIQUE DES CLASSES

3.1 Les besoins

Dans le chapitre précédent, nous nous sommes intéressés à la dynamique des instances, nous allons étudier maintenant la dynamique des classes. Pour que l'utilisateur ait la possibilité de faire évoluer une base, nous devons gérer ces deux aspects de la dynamique (classes et instances). Par dynamique des classes, nous entendons aussi bien la création d'une classe que sa mise à jour.

La création d'une classe engendre une modification du graphe d'héritage et éventuellement une modification de classes existantes. En effet, les propriétés d'une classe sont d'une part les propriétés héritées de ses super-classes et d'autre part les propriétés propres à la classe créée. Si nous créons une classe en milieu de graphe, nous devons faire hériter les nouvelles propriétés dans les sous-classes de la nouvelle classe.

Exemple :
Nous partons du graphe d'héritage suivant :

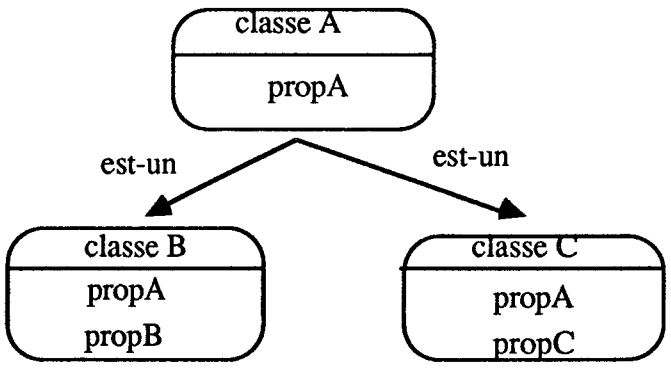


propA : propriétés propres à A

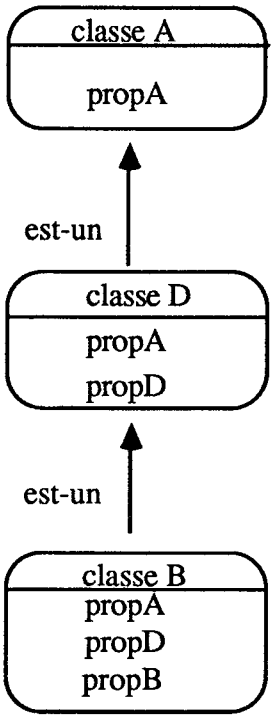
propB : propriétés propres à B

Nous avons un lien d'héritage entre les classes A et B. La classe B est une sous-classe de la classe A. Elle possède donc les propriétés de A qui sont héritées et ses propres propriétés : propB.

Si nous ajoutons une classe C sous la classe A, avec les propriétés propC, seul le graphe d'héritage est modifié, aucune classe n'est touchée par l'ajout de la classe C.



Si nous ajoutons une classe D, qui est sous-classe de la classe A, et super-classe de la classe B, nous constatons que la classe B est modifiée, car nous devons ajouter à ses propriétés les nouvelles propriétés créées dans la classe D.



Nous introduisons deux nouveaux moyens pour créer des classes, en offrant à l'utilisateur la possibilité de créer des classes à partir d'une classe existante, ou à partir d'une structure.

La mise à jour (ajout, suppression, modification de propriétés) d'une classe a un impact sur le graphe d'héritage et sur les structures associées à cette classe ainsi que sur les instances rattachées à ces structures. Un point important de notre étude est l'optimisation de la mise à jour des instances, lors de la mise à jour d'une classe, en nous appuyant sur les structures. Nous n'insistons pas sur l'impact des mises à jour d'une classe sur le graphe d'héritage, car une étude sur le graphe d'héritage dans SHOOD a été faite dans le cadre d'un DEA [ESC89].

Nous allons étudier les problèmes liés à la création et à la mise à jour des classes ainsi que les conditions que doit respecter l'utilisateur pour conserver la cohérence du schéma. L'étude se fera uniquement au niveau du modèle SHOOD.

3.2 Création d'une classe

Les propriétés d'une classe sont l'union des propriétés de ses super-classes ainsi que les propriétés définies au niveau de la classe même.

Pour l'instant nous avons considéré la classe déconnectée du graphe d'héritage. Nous avons traité au même niveau les propriétés héritées et les propriétés propres à la classe, car dans la création des structures et des instances, ces deux types de propriétés ont le même impact. Nous créons les structures à partir de toutes les propriétés d'une classe, et une instance doit

instancier toutes les propriétés d'une classe. Pour la création ou la mise à jour d'une classe, ces deux sortes de propriétés doivent être gérées différemment.

Pour créer une classe trois méthodes sont envisagées.
L'utilisateur peut créer une classe à partir de "rien", c'est-à-dire qu'il ne s'appuie pas sur les classes déjà existantes. C'est la création de classe qui se trouve dans tous les systèmes.
Une deuxième façon de créer des classes est de s'appuyer sur une structure d'une classe existante. Nous allons transformer la structure en une classe du graphe d'héritage.
La dernière façon est de créer une classe à partir d'une autre classe en modifiant éventuellement une ou plusieurs propriétés.
Dans les deux dernières méthodes, nous pouvons choisir de garder un lien entre la classe initiale et la classe finale ou de rendre ces deux classes indépendantes. Nous nous intéressons aussi à la transition des instances d'une classe à l'autre.

3.2.1 Création d'une classe à partir de "rien"

Nous mettons le mot "rien" entre guillemets car une classe est toujours créée dans le graphe d'héritage, donc toujours au moins dépendante de la classe Objet (qui est la racine du graphe d'héritage dans SHOOD).

Une classe peut être insérée au milieu du graphe d'héritage. Elle possède alors une ou plusieurs super-classes et une ou plusieurs sous-classes.

A la création d'une classe en milieu de graphe, nous distinguons quatre étapes :

- héritage des propriétés des super-classes
- spécialisation des propriétés héritées
- création des propriétés propres de la classe
- mise à jour des sous-classes de la classe créée

Exemple :

Nous avons une classe personne et une classe Elève-sportif qui est une sous-classe de la classe Personne.

Classe Personne :

nomclass = personne
propriété =
nom : *un* chaîne
sorte-de attribut-obligatoire
prénom : *un* chaîne
âge : *un* entier

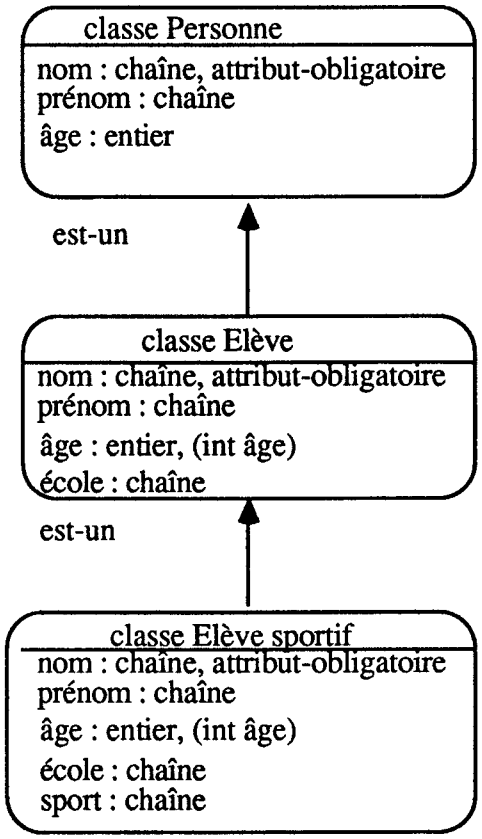
Classe Elève-sportif
nomclasse = élève-sportif
propriété =
nom : *un* chaîne
sorte-de attribut-obligatoire
prénom : *un* chaîne
âge : *un* entier
sport : *un* chaîne

Dans la classe Elève-sportif les attributs nom, prénom et âge sont hérités de la classe Personne et l'attribut sport est un attribut défini dans la classe.
Nous allons créer une classe Elève sous-classe de la classe Personne et super-classe de la classe Elève-sportif.

Classe Elève
nomclasse = élève
propriété =
nom : *un* chaîne
sorte-de attribut-obligatoire
prénom : *un* chaîne
âge : *un* entier
contrainte
(int âge)
école : *un* chaîne

Lors de l'ajout de la classe Elève :
La classe Elève hérite des attributs nom, prénom et âge de la classe Personne.
L'attribut âge a été spécialisé : une contrainte forte a été ajoutée sur cet attribut : (int âge) qui stipule que l'âge doit être compris entre 2 et 18.
Un attribut propre à la classe Elève a été ajouté : école.
La classe Elève-sportif est modifiée, et devient :
Classe Elève-sportif
nomclasse = élève-sportif
propriété =
nom : *un* chaîne
sorte-de attribut-obligatoire
prénom : *un* chaîne
âge : *un* entier
contrainte
(int âge)
école : *un* chaîne
sport : *un* chaîne

Le nouveau graphe d'héritage est :



3.2.1.1 Héritage des propriétés

L'héritage des propriétés pose de nombreux problèmes (conflit de nom...) que nous ne développons pas ici [ESC89]. Nous considérons qu'une classe hérite de l'union des propriétés de ses super-classes.

3.2.1.2 Spécialisation des propriétés héritées

Dans SHOOD, nous pouvons uniquement spécialiser les propriétés héritées, mais nous ne pouvons pas les redéfinir, car le modèle supporte la spécialisation stricte. Dans ce cas, une instance créée dans une classe est aussi instance des super-classes de sa classe d'origine. Si des propriétés héritées sont redéfinies nous ne pouvons plus garantir ce fait.
Nous allons regarder plus en détail les mises à jour permises (ou non) sur les propriétés héritées.
La suppression d'un attribut hérité est interdite, car comme une instance est aussi instance de la super-classe de sa classe d'origine elle doit avoir une valeur pour cet attribut. Par exemple, nous ne pouvons pas supprimer l'attribut prénom dans la classe Elève, car sinon les élèves ne pourraient pas être instances de la classe Personne : ils n'auraient pas de prénom.

La modification du type d'un attribut hérité est autorisée uniquement si le nouveau type est un sous-ensemble du premier. En effet, si le nouveau type est incompatible avec le type de la super-classe, une instance créée au niveau de cette classe ne peut pas appartenir aussi à sa

super-classe. Considérons que la classe entier est une sous-classe de la classe nombre. Si un attribut hérité est de type nombre, nous pouvons alors le spécialiser en type entier, mais ne pouvons pas le changer en type chaîne.

Un attribut non obligatoire peut devenir obligatoire, mais l'inverse n'est pas possible. Si un attribut est obligatoire dans une classe, les instances de ses sous-classes doivent aussi posséder une valeur pour cet attribut. Par exemple, nous pouvons, dans la classe Elève, rendre obligatoire l'attribut prénom, mais l'attribut nom doit rester obligatoire.

Une contrainte faible peut devenir forte, mais pas l'inverse. On peut ajouter des contraintes intra ou inter-attributs. Si nous voulons qu'une instance appartienne à la super-classe de sa classe d'origine, nous devons autoriser uniquement la création d'une nouvelle classe autant ou plus contrainte que sa super-classe. La classe Elève est plus contrainte que la classe Personne : elle possède une contrainte de plus. La classe Elève-sportif est autant contrainte que la classe Elève.

Toutes les mises à jour d'inférences sont autorisées : ajout, modification, suppression, car au niveau d'un même attribut plusieurs moyens d'inférence sont autorisés. Par exemple, à la création de la classe Elève, nous pouvons ajouter une inférence pour calculer l'âge.

3.2.1.3 Création des propriétés propres à la classe

La création des propriétés propres d'une classe permet d'ajouter de nouveaux attributs et de nouvelles contraintes inter-attributs. Dans la classe Elève, nous avons ajouté l'attribut école.

3.2.1.4 Mise à jour des sous-classes

La mise à jour des sous-classes (immédiates ou non) d'une classe créée A, consiste en l'ajout des nouvelles propriétés de la classe A, et en la modification des propriétés héritées, si elles ont été spécialisées dans la classe A. Par exemple, dans la classe Elève-sportif, nous ajoutons l'attribut école, et modifions l'attribut âge en ajoutant la contrainte (int âge) (car cet attribut a été modifié dans la classe Elève).

3.2.2 Création d'une classe à partir d'une structure

Les structures peuvent être utilisées pour détecter une mauvaise définition de classe. Dans le domaine de la CAO, nous avons vu que les structures permettent de juger de l'amélioration d'un objet en cours de conception. Le but de l'utilisateur est de modifier l'objet de façon à obtenir un objet en complet accord avec la définition de la classe. Dans d'autres domaines, biométrie par exemple, les structures vont aussi servir à juger du manque d'informations sur les objets, et à détecter les incohérences sur ces objets. Les propriétés créées dans la classe initiale sont les propriétés que sont sensées respecter les instances; mais ce ne sont pas des propriétés "exactes", des incertitudes peuvent encore exister sur certaines d'entre-elles. La connaissance de la proportion d'instances appartenant à chaque structure peut aider l'utilisateur à appréhender ces incertitudes. Par exemple, si l'utilisateur s'aperçoit que plus de 60% des instances se trouvent dans une structure qui n'est pas la structure maximale, il peut être amené à créer une classe répondant à tous les critères de cette structure.

Exemple :

Nous avons une classe Personnesala.

Classe Personnesala
  nomclasse = personnesala
  propriété =
    nom : *un* chaîne
    prénom : *un* chaîne
    salaire : *un* réel
      contraintv
        (min salaire)

La classe Personnesala comprend les attributs nom, prénom et salaire. Ce dernier est soumis à une contrainte faible (min salaire) qui stipule que le salaire doit être supérieur au smic.

Beaucoup d'instances de la classe Personnesala, appartiennent à la structure Ck de forme Ck : {nom : chaîne, prénom : chaîne}, l'utilisateur décide alors de créer une classe Personne ne possédant que ces deux attributs. Une personne sans salaire est alors considérée comme une instance complète et cohérente de la classe Personne et non plus comme une instance incomplète de la classe Personnesala.

Comme la structure avec laquelle nous allons créer la nouvelle classe n'est pas la structure maximale (la classe correspondant étant la classe initiale), elle est moins complète et/ou moins cohérente. Donc la nouvelle classe qui va être créée est une classe moins complète et/ou moins cohérente que la classe initiale.

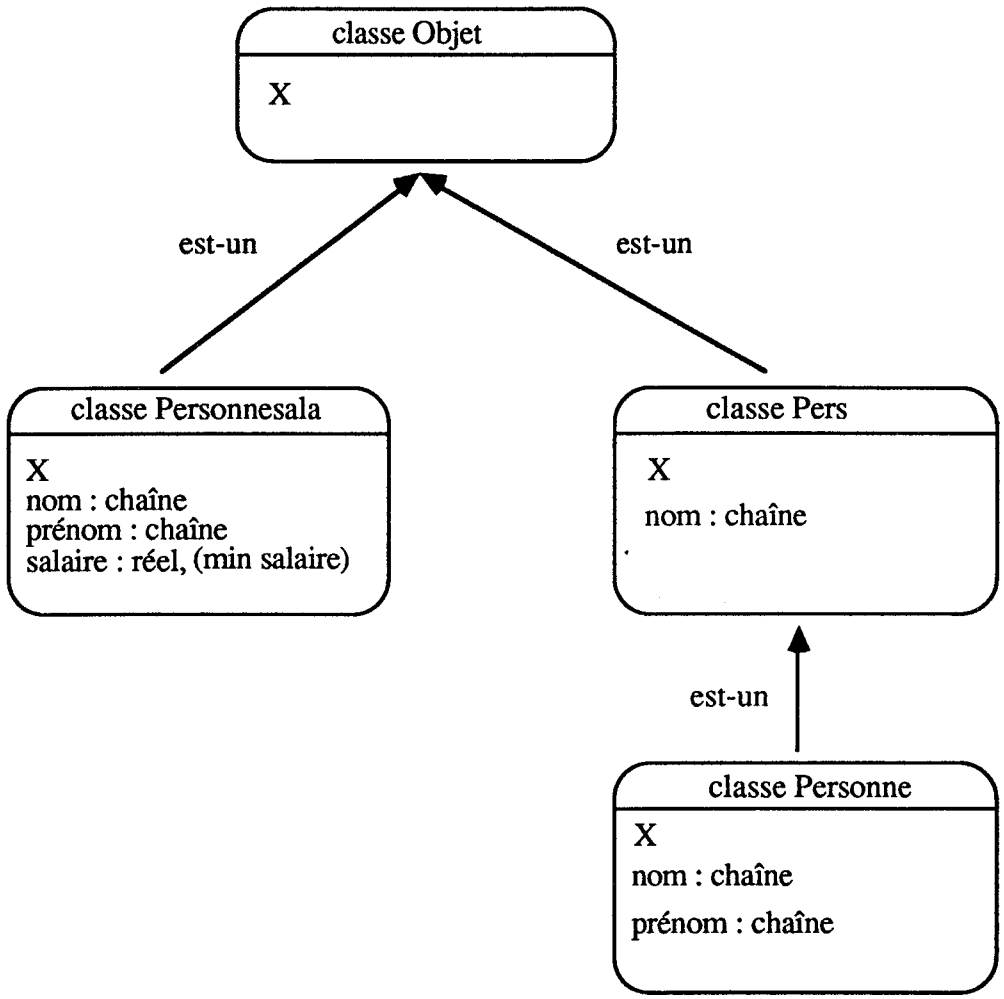
L'utilisateur peut choisir de garder un lien entre la classe de la structure et la nouvelle classe, ou ne pas garder de lien.

3.2.2.1 Aucun lien entre la classe initiale et la classe finale

Le système crée automatiquement la nouvelle classe à partir de la structure choisie dans la classe initiale. L'utilisateur spécifie explicitement la place de la classe finale dans le graphe d'héritage. Cette place est indépendante de la place de la classe initiale, mais doit évidemment répondre aux impératifs et aux règles de l'héritage (spécialisation stricte ...).

Exemple :

Le nouveau graphe d'héritage peut être le suivant :



X représente les propriétés de la classe Objet. Ces propriétés ne sont pas détaillées, car elles ne présentent pas d'intérêt pour l'exemple.

L'utilisateur a décidé de placer la classe Personne sous la classe Pers, cela est possible car la spécialisation stricte est respectée. Les classes Personnesala et Personne sont indépendantes.

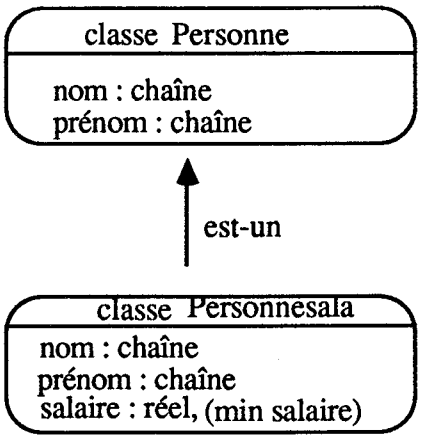
3.2.2.2 Lien conservé entre la classe initiale et la classe finale

Si un lien est conservé, la nouvelle classe est, soit une généralisation de la classe initiale, soit une classe ayant une super-classe directe et commune avec la classe initiale. Les classes initiale et finale sont alors dépendantes.

Si la classe créée est moins complète, cela signifie qu'elle contient moins d'attributs : nous rétrocédons les attributs communs aux deux classes (la classe initiale et la classe finale). Dans la classe initiale des attributs qui ont été jusqu'à présent considérés comme propres à la classe sont maintenant considérés comme hérités.

Le système modifie automatiquement le graphe d'héritage.

Dans notre exemple, le nouveau graphe est :



La classe Personne a été créée en tant que super-classe de la classe Personnesala.

Les attributs nom, prénom sont considérés comme des attributs propres dans la classe Personne et comme des attributs hérités dans la classe Personnesala.

Si la classe créée est moins cohérente, cela signifie qu'elle possède moins de contraintes ou bien que certaines des contraintes de la classe initiale sont niées.

Si elle possède moins de contraintes (intra ou inter-attributs), la nouvelle classe est une généralisation de la classe initiale. Par exemple, la classe Personne possède moins de contraintes que la classe Personnesala, c'est donc une généralisation de la classe Personnesala. Le système la crée en tant que super-classe de la classe Personnesala.

Si des contraintes sont niées, nous ne pouvons pas faire de la nouvelle classe une généralisation de la classe initiale, car nous n'aurions plus la spécialisation stricte. Nous allons alors créer une classe système avec les propriétés communes aux deux classes, puis spécialiser cette classe en créant la nouvelle classe où les contraintes (intra ou inter-attributs) sont niées.

Exemple :

L'utilisateur décide de créer la classe Personneexpl à partir de la structure Cj de la classe Personnesala.

Cj :{nom : chaîne, prénom : chaîne, salaire : réel, notmin (salaire)}

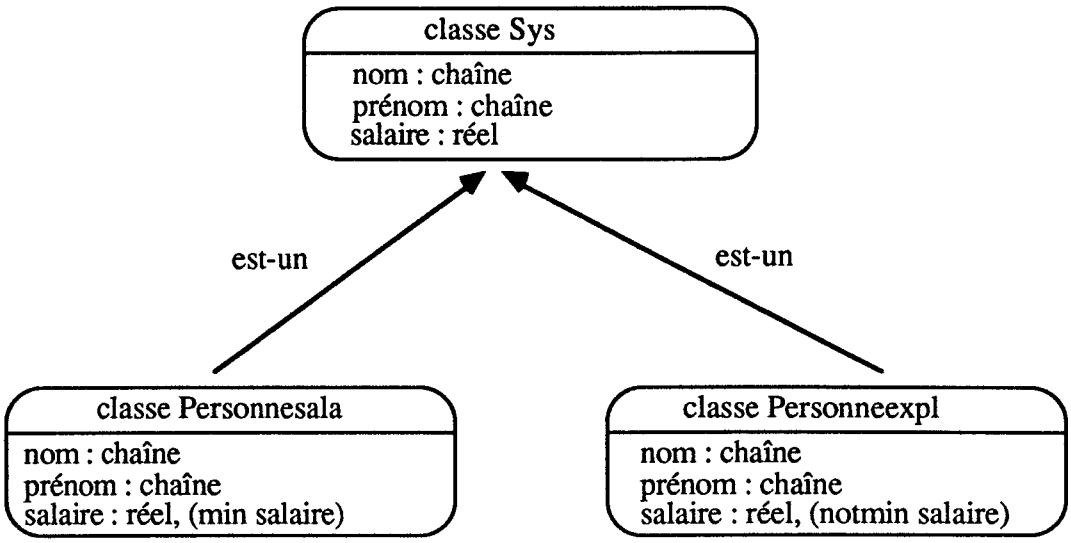
Dans cette structure la contrainte (min salaire) est violée.

La classe Personneexpl ne peut pas être une super-classe de la classe Personnesala, car les contraintes (min salaire) et (notmin salaire) sont incompatibles.

Le système crée une classe système : Sys, qui est une super-classe de la classe Personnesala et de la classe Personneexpl. Cette classe possède les propriétés communes aux deux classes, c'est-à-dire : nom, prénom et salaire (cet attribut est sans contrainte).

Dans la classe Personneexpl, on spécialise l'attribut salaire en lui ajoutant la contrainte (notmin salaire) qui spécifie que le salaire doit être inférieur au smic.

Nouveau graphe d'héritage :



Donc, dans le cas où la classe finale possède moins d'attributs et/ou moins de contraintes, une seule classe est créée : la classe finale qui est super-classe de la classe initiale. Dans le cas où la classe finale possède des contraintes niées par rapport à la classe initiale, alors deux classes sont créées : la classe finale et une classe système qui est une super-classe de la classe finale et de la classe initiale. C'est le système qui crée automatiquement les classes, et les place dans le graphe.

3.2.2.3 Transition des instances

Les instances de la structure de la classe initiale vont transiter dans la classe finale. Elles sont toutes rattachées à la structure maximale de la classe finale, puisque la classe a été créée selon le modèle de ces instances. Par la suite, de nouvelles instances peuvent être créées dans cette classe, ces instances sont rattachées aux différentes structures de la classe.
Une classe système (Sys dans notre exemple) n'a pas d'instance propre, car elle sert uniquement à faire le lien entre la classe initiale et la classe finale. Elle ne possède que les instances rétrocédées (d'une de ses sous-classes) ou héritées (d'une de ses super-classes) par la classification automatique.

3.2.3 Création d'une classe à partir d'une classe existante

Nous offrons à l'utilisateur la possibilité de créer une classe à partir des propriétés d'une classe existante. Ceci, en vue d'une plus grande convivialité pour l'utilisateur, il ne doit pas redéfinir à deux endroits différents des propriétés de classes presque identiques; de plus, nous lui permettons de faire transiter des instances de la classe initiale à la classe finale.

Après que l'utilisateur ait stipulé les mises à jour (ajout, suppression, modification de propriétés) à apporter à la classe initiale, le système place la nouvelle classe dans le graphe. Nous considérons, dans ce paragraphe, que l'utilisateur ne désire pas faire une simple spécialisation de la classe initiale. Toutes les mises à jours effectuées sur la classe initiale

sont autorisées car la classe finale n'est pas placée comme sous-classe de la classe initiale, mais peut être placée n'importe où dans le graphe. Cette place doit évidemment répondre aux impératifs de la spécialisation stricte, mais pas forcément par rapport à la classe initiale.

Si l'utilisateur ne désire pas garder de lien entre la classe initiale et la classe finale, il doit préciser la place de la nouvelle classe.

Si l'utilisateur désire garder un lien entre les deux classes, nous distinguons deux cas :

- la classe finale possède moins d'attributs et/ou moins de contraintes que la classe initiale : la nouvelle classe est alors créée comme super-classe de la classe initiale
- la classe finale possède plus d'attributs et/ou de contraintes, ou des contraintes sont niées par rapport à la classe initiale : le système crée alors deux classes : une classe système possédant les propriétés communes à la classe initiale et à la classe finale, et la nouvelle classe.

L'utilisateur a la possibilité de faire transiter des instances de la classe initiale à la classe finale. Cette transition se fait structure par structure. C'est-à-dire que l'utilisateur fait transiter toutes les instances d'une structure dans la nouvelle classe.
Dans deux cas, nous n'autorisons pas la transition d'instances :

- Si un attribut est supprimé et si une instance possède une valeur pour cet attribut, alors l'instance ne peut pas transiter dans la nouvelle classe, car il y aurait une perte d'information au niveau de l'instance.
- Si la nouvelle classe créée possède des contraintes inverses par rapport aux contraintes de la classe initiale, les instances ne niant pas les contraintes de la classe initiale ne peuvent pas transiter. Nous n'autorisons pas cette transition, car nous ne voulons pas perdre de cohérence sur les instances.

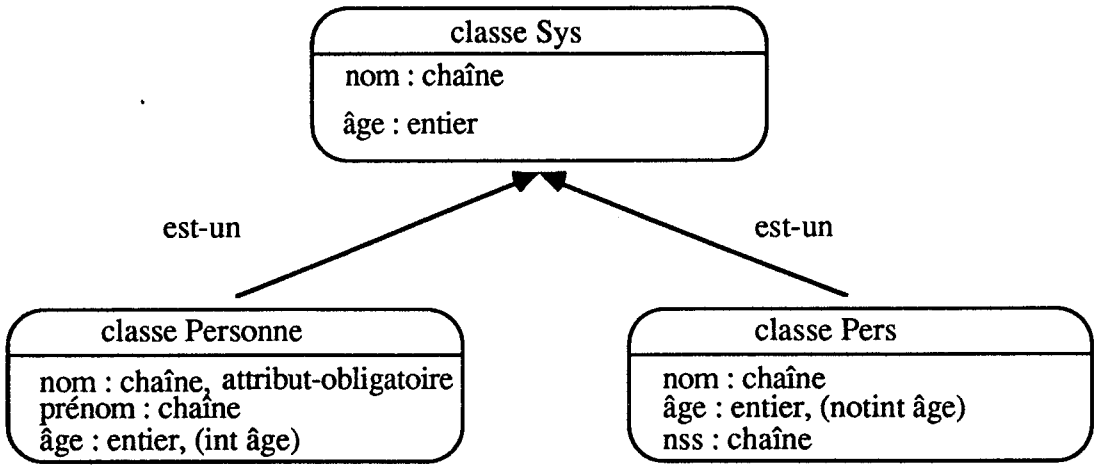
Exemple :

Classe Personne
nomclasse = personne
propriété =
nom : *un* chaîne
sorte-de attribut-obligatoire
prénom : *un* chaîne
âge : *un* entier
contrainv
(int âge)

L'utilisateur désire créer une classe Pers à partir de la classe Personne. Il stipule que l'attribut nom devient non obligatoire, il ajoute l'attribut nss (numéro de sécurité sociale), il supprime l'attribut prénom et change la contrainte (int âge) en (notint âge) qui est la contrainte inverse.

Classe Pers
  nomclasse = pers
  propriété =
    nom : *un* chaîne
    âge : *un* entier
      contraintv
        (notint âge)
    nss : *un* chaîne

Si l'utilisateur désire garder un lien entre la classe Personne et la classe Pers alors le système génère le graphe suivant :



Les instances de la structure Cj : {nom : chaîne, prénom : chaîne} ne peuvent pas transiter dans la nouvelle classe, car les instances perdraient la valeur de l'attribut prénom.
Les instances de la structure Ck : {nom : chaîne, âge : entier, (int âge)} ne peuvent pas non plus transiter, car elles perdraient de la cohérence.

3.3 Mise à jour d'une classe

Nous allons étudier dans ce paragraphe les impacts des mises à jour d'une classe.

Les principales mises à jour possibles dans une classe sont :

- l'ajout d'une propriété : attribut ou contrainte inter-attributs
- la suppression d'une propriété : attribut ou contrainte inter-attributs
- la modification d'une propriété :
    - modification du type d'un attribut
    - possibilité de transformer un attribut obligatoire en un attribut non obligatoire ou vice versa
    - possibilité de transformer une contrainte faible en une contrainte forte ou vice versa

- mise à jour (ajout, modification, suppression) des contraintes portant sur un ou plusieurs attributs
- mise à jour (ajout, modification, suppression) des inférences portant sur les attributs

Les impacts de la mise à jour d'une classe portent sur :

- la classe
- les structures
- les instances

Si la classe modifiée possède des sous-classes alors ses sous-classes sont modifiées aussi.

3.3.1 Impact sur la classe

Les possibilités de mises à jour des propriétés héritées sont les mêmes que celles définies dans la création d'une classe. C'est-à-dire qu'un attribut hérité obligatoire ne peut pas devenir non obligatoire, qu'une contrainte héritée forte ne peut pas devenir faible et que la modification du type d'un attribut doit être une spécialisation du type de l'attribut de la super-classe.

L'ajout d'une propriété est toujours permis, que ce soit un attribut ou une contrainte inter-attributs, car cet ajout ne met pas en cause la spécialisation stricte.

Si la propriété à modifier est propre à la classe alors toutes les mises à jour (suppression, modification) sont permises sur cette propriété.

Nous remarquons que la suppression d'un attribut peut avoir des effets de bord. En effet, si cet attribut est argument d'inférences ou de contraintes alors ces inférences et ces contraintes doivent être aussi supprimées, car elles ne sont plus calculables. Les autres mises à jour n'ont pas d'effets de bord.

Après avoir contrôlé que les mises à jour sont correctes, le système effectue la **mise à jour physique** des propriétés de la classe.

Dans la suite de notre paragraphe, nous allons prendre pour exemple la classe Avion suivante:

```
Avion
  nomclasse = avion
  propriété =
    imat  : un chaîne
            sorte-de attribut-obligatoire
    aile  : un entier
    atte  : un entier
            contraintv
              (contr (atte))
    deco  : un entier
            inférence
              (fonc (aile))
```

3.3.2 Impact sur les structures

Deux alternatives sont possibles pour créer les structures de la classe modifiée.
La première alternative consiste en la modification des structures existantes. Dans ce cas, pour chaque structure les impacts des mises à jour sont à étudier. Certaines structures sont alors à supprimer, d'autres à créer, d'autres à modifier. Après ces mises à jour, des structures peuvent être en double et sont alors à épurer; on voit que ce mécanisme est peu rigoureux, car il n'y a pas de règle stricte.
C'est pourquoi nous avons choisi la deuxième alternative qui consiste à recréer toutes les structures à partir de la classe modifiée. Dans ce cas, nous n'introduisons pas de nouveaux algorithmes, mais utilisons l'algorithme de création de structures lors de la création d'une classe.

3.3.3 Impact sur les instances

Nous allons voir que la mise à jour d'une classe entraîne aussi la mise à jour des instances existantes.
Le système doit non seulement modifier les instances mais aussi classifier ces instances dans les nouvelles structures.

3.3.3.1 Modification des instances

Lors de l'**ajout d'un attribut non obligatoire**, si une inférence calculable (c'est-à-dire une inférence ayant ses arguments instanciés) est associée à l'attribut ajouté, nous pouvons donner une valeur à cet attribut; nous ajoutons alors cette valeur dans l'instance. Si aucune inférence n'est calculable sur cet attribut, alors cette mise à jour n'a aucun impact sur les instances. Dans les systèmes où tous les attributs (instanciés ou non) apparaissent dans les instances, toutes les instances doivent être mises à jour lors de l'ajout d'un attribut (on ajoute à une instance l'attribut ajouté associé à la valeur nil).

Par exemple, nous ajoutons l'attribut fuse soumis à l'inférence (prop aile) qui permet de calculer la longueur du fuselage à partir de la longueur des ailes. Les instances possédant l'attribut aile instancié sont modifiées, le système rajoute l'attribut fuse, calculé à partir des ailes; les autres instances ne sont pas modifiées.

Si l'**attribut ajouté** est un **attribut obligatoire**, l'attribut doit toujours avoir une valeur. Si aucune inférence calculable n'est rattachée à l'attribut, l'utilisateur doit donner pour chaque instance une valeur à cet attribut.
Par exemple, si l'utilisateur ajoute l'attribut obligatoire fuse dans la classe Avion, toutes les instances de la classe Avion doivent être modifiées. L'utilisateur donne alors une valeur à l'attribut fuse pour chaque instance.

L'**ajout d'une contrainte faible** intra ou inter-attributs n'entraîne pas de modifications d'instances. En effet, la valeur d'une contrainte n'est pas gardée au niveau de l'instance mais au niveau de la structure de l'instance.

L'**ajout d'une contrainte forte** intra ou inter-attributs, peut conduire le système à rejeter des instances. En effet, une contrainte forte ne peut être violée et il se peut que pour certaines instances cette règle ne soit pas respectée. L'utilisateur doit alors soit modifier l'instance soit supprimer cette instance.
Si nous ajoutons, dans la classe Avion, la contrainte forte (super (deco atte)), il faut vérifier pour toutes les instances où la contrainte est calculable, si cette contrainte est violée ou non. Si cette contrainte est violée dans une instance, alors l'utilisateur doit modifier cette instance.

La **suppression d'un attribut** dans une classe entraîne sa suppression dans les instances où il apparaît.
Si cet attribut est utilisé comme argument d'une inférence, nous ne supprimons pas la valeur de l'attribut calculé à partir de cette inférence, car nous voulons perdre le moins d'informations possibles sur les instances au moment de la mise à jour d'une classe.
Si ,lors de la modification de la classe Avion, l'attribut aile est supprimé, l'inférence (fonc (aile)) dans l'attribut deco est alors supprimée automatiquement par le système. Si dans une instance l'attribut deco possède une valeur, nous laissons cette valeur.

La **suppression de contraintes** intra ou inter-attributs n'affecte pas les instances, car les valeurs des contraintes ne sont pas gardées au niveau des instances.

Si un **attribut obligatoire** est rendu **non obligatoire**, les instances ne sont pas modifiées.

Si un **attribut non obligatoire** devient **obligatoire**, toutes les instances doivent alors posséder une valeur pour cet attribut. Si dans une instance la valeur de l'attribut n'est pas calculable par inférence alors le système propose à l'utilisateur cette instance, pour que celui-ci puisse donner une valeur à l'attribut.
Si, dans notre exemple, l'attribut deco est rendu obligatoire, alors toutes les instances qui ne possèdent pas de valeur pour cet attribut, sont proposées à l'utilisateur.

La **modification du descripteur contrainte** (ou **contraintv**) est décomposée en contraintes ajoutées et en contraintes supprimées.

L'**ajout d'inférences** entraîne la mise à jour d'instances dans le cas où certains attributs, non présents dans l'instance, peuvent être calculés par cette nouvelle inférence.
Si l'utilisateur ajoute l'inférence (prop aile) à l'attribut atte, cette inférence est calculée dans les instances où l'attribut atte ne possède pas de valeur et où l'attribut aile est instancié.

La **suppression d'inférences** ne touche pas les instances, car nous voulons perdre le moins d'informations possibles lors de la mise à jour d'une classe. Donc, si la valeur d'un attribut a été calculée par l'inférence supprimée nous gardons cette valeur.

La **modification du descripteur inférence** est décomposée en suppression d'inférences et en ajout d'inférences.

Dans les systèmes existants, deux méthodes sont employées pour modifier les instances.
La première méthode est le **filtrage** : les instances sont mises à jour uniquement lorsqu'elles sont accédées, c'est l'optique choisie par Orion. Dans cette méthode, la mise à jour d'une classe est rapide, mais le temps d'accès à une instance est long car le système doit analyser les mises à jour à faire; de plus, nous ne pouvons pas savoir quand la répercution de la mise à jour d'une classe est réellement terminée, car nous ne savons pas à quel moment toutes les instances de la classe modifiée ont été accédées; des problèmes de cohérence de la base à un instant T sont aussi posés.

La deuxième méthode est la **conversion** (utilisée dans GemStone) : les instances sont mises à jour dès la modification d'une classe. Avec cette méthode, la mise à jour de la classe prend plus de temps, car elle comprend aussi la mise à jour des instances, mais la base est toujours cohérente.

Pour notre étude nous avons choisi l'approche de la conversion, en nous aidant des structures pour optimiser la mise à jour des instances.

Nous pourrions considérer les instances une à une et analyser pour chacune d'elle l'impact des mises à jour. Si la base de connaissances contient beaucoup d'instances, le temps de mise à jour risque d'être long. Les structures vont nous permettre d'optimiser ce temps.

Les instances sont mises à jour structure par structure. Nous analysons pour chaque structure les modifications par rapport à la classe initiale, puis nous appliquons ces modifications aux instances. Une fois les instances modifiées, les structures nous permettent de reclassifier celles-ci par rapport aux structures de la nouvelle classe.

Afin de modifier les instances en nous aidant des structures, nous allons, à partir des mises à jour des propriétés de la classe, créer cinq "**paquets**". Ces paquets contiennent uniquement les mises à jour qui ont un impact sur les instances. Dans chaque paquet, nous regroupons des mises à jour qui ont le même impact sur les instances. Les paquets sont analysés, dans un ordre pré-déterminé, structure par structure. Après l'analyse de ces paquets, les instances sont modifiées le cas échéant.
L'analyse d'un paquet se fait une seule fois par structure (et non pas une fois par instance), car les structures permettent de filtrer les opérations à faire. De plus si pour une structure

donnée, aucune mise à jour sur les instances n'est nécessaire, ce fait est détecté au niveau de la structure et non pas au niveau des instances, celles-ci ne sont, alors, pas accédées.

Le premier paquet contient tous les attributs supprimés lors de la mise à jour de la classe.
Le deuxième paquet contient les nouvelles inférences ajoutées aux attributs.
Le troisième paquet contient toutes les contraintes ajoutées, que ce soit des contraintes intra ou inter-attributs.
Le quatrième paquet contient les nouveaux attributs obligatoires. Par nouveaux attributs obligatoires, nous entendons les ajouts d'attributs obligatoires et aussi les anciens attributs non obligatoires qui sont devenus obligatoires après la mise à jour.
Le cinquième paquet contient les contraintes faibles qui ont été supprimées lors de la mise à jour de la classe.

Nous remarquons que lors de l'ajout d'un attribut non obligatoire, celui-ci n'apparaît pas dans les paquets. En effet, si aucune inférence n'est associée à cet attribut, cet ajout n'a pas d'impact sur les instances; si une ou plusieurs inférences sont associées à cet attribut, elles apparaissent dans le deuxième paquet, et la valeur de l'attribut est calculée, le cas échéant, à partir de ces inférences.

Dans une structure nous analysons en premier le premier paquet. Si un attribut supprimé est un attribut de la structure, alors nous savons que cet attribut est à supprimer dans toutes les instances de la structure.

Le deuxième paquet est ensuite analysé. Si tous les arguments de l'inférence sont des attributs de la structure, et que l'attribut à calculer n'est pas attribut de la structure, alors nous déduisons que l'inférence est calculable. Pour toutes les instances, nous allons calculer cette inférence.

Si tous les arguments, d'une contrainte du troisième paquet, sont attributs de la structure alors la contrainte est calculable. Cette contrainte est calculée pour toutes les instances de la structure. Si la contrainte est forte alors le système propose à l'utilisateur les instances qui ne respectent pas la contrainte.

Ces trois premiers paquets peuvent entraîner des mises à jour directement sur les instances.

Le quatrième paquet (nouveaux attributs obligatoires) intervient uniquement au niveau de la structure et non pas au niveau des instances. Nous connaissons pour chaque structure les attributs instanciés. Le système vérifie que les attributs contenus dans le quatrième paquet font bien partie des attributs de la structure. Si ce n'est pas le cas, alors toutes les instances de la structure sont rejetées. Le système propose à l'utilisateur toutes les instances de la structure pour que celui-ci donne une valeur aux attributs obligatoires.

Le cinquième paquet n'intervient pas dans la mise à jour même des instances, car nous avons vu que la suppression de contraintes ne modifie pas les instances. Ce paquet intervient dans la reclassification des instances. Ce paquet nous permet de déterminer pour chaque structure la liste des contraintes violées dans cette structure : ce sont les anciennes contraintes violées de la structure moins les contraintes faibles supprimées du cinquième paquet et contenues dans la structure.

Exemple : Le cinquième paquet contient les contraintes faibles A et D qui doivent être supprimées. Dans une structure Cj les contraintes B, D et F sont violées, et la contrainte faible A est non violée. Nous déduisons que les nouvelles contraintes violées de la structure sont B et F.

Nous devons aussi, lors de la mise à jour des instances, nous intéresser aux boucles de dégradation.
Dans une même structure une instance peut être versionalisée ou non.
Deux solutions peuvent être retenues.
La première solution consiste à garder les boucles de dégradation. Pour ce faire nous devons garder la cohérence de la boucle. En effet, pour sortir de la boucle de dégradation nous comparons la version 0 à la version courante; si ces deux versions ne sont pas issues de la même classe, nous ne pouvons plus les comparer, et donc ne pouvons plus sortir de la boucle de dégradation. Pour garder la cohérence de la boucle, il faut mettre à jour toutes les versions de la boucle. Si des contraintes faibles sont devenues fortes, nous constatons que certaines versions ne peuvent être gardées. De plus, si la mise à jour d'une classe consiste en l'ajout d'un attribut obligatoire non calculable, nous devrions demander à l'utilisateur de donner une valeur à cet attribut pour toutes les versions. Nous voyons donc que la mise à jour des boucles de dégradation n'est pas simple, car la boucle peut être complètement remise en cause.

Nous avons choisi une seconde solution qui consiste en la suppression de toutes les boucles de dégradation lors de la modification d'une classe. En effet, nous considérons que l'ancienne et la nouvelle classe sont deux concepts différents, et que la cohérence et la complétude de l'une n'ont pas la même signification que la cohérence et la complétude de l'autre.
Donc toutes les instances versionalisées sont rendues non versionalisées.

3.3.3.2 Classification des instances

Pour chaque ancienne structure, nous gardons la liste des attributs ajoutés et la liste des attributs supprimés.
Une ancienne structure est comparée à toutes les nouvelles structures. Si les attributs de l'ancienne structure, plus ou moins les ajouts et les suppressions d'attributs associés à cette structure, sont identiques aux attributs d'une nouvelle structure, alors nous déduisons que cette nouvelle structure est une structure de reclassification possible pour les instances de l'ancienne structure.
Si aucune contrainte faible n'est calculable dans une ancienne structure Cj, alors toutes les instances de cette structure Cj sont reclassifiées dans une seule nouvelle structure Kj.
Si des contraintes faibles sont calculables pour une ancienne structure Cj alors toutes les instances de cette structure Cj ne sont pas reclassifiées dans une même structure. En effet, certaines instances de la structure Cj vont violer les contraintes et d'autres non. A l'aide de la liste des contraintes violées de la structure Cj, et de la liste des contraintes violées par l'instance, nous allons déterminer quelle est la nouvelle structure de l'instance parmi les structures de reclassification possible de la structure Cj.

La recherche de la nouvelle structure d'une instance est optimisée car cette recherche se fait sur un nombre restreint de structures. En effet, cette recherche ne se fait que dans les structures de reclassification possible de l'ancienne structure.

3.3.3.3 Exemple

Nous avons la classe Avion suivante :

Classe Avion :
  nomclasse = avion
  propriété =
    imat : *un* chaîne
           sorte-de attribut-obligatoire
    aile : *un* entier
           contraintv
              (cont1 aile)
    atte : *un* entier
    deco : *un* entier
           inférence
              (fonc aile)

Les mises à jour effectuées sur cette classe sont :
- suppression de l'attribut aile. Cette suppression entraîne la suppression de la contrainte faible associée (cont1 aile) et de l'inférence (fonc aile) associée à l'attribut deco.
- ajout d'une contrainte faible sur l'attribut atte (cont2 atte)
- ajout d'un nouvel attribut : fuse
- modification de l'attribut deco : ajout d'une nouvelle inférence (prop fuse) et transformation de cet attribut en attribut obligatoire.

Après application des mises à jour la classe devient :

Classe Avion :
  nomclasse = avion
  propriété =
    imat : *un* chaîne
           sorte-de attribut-obligatoire
    atte : *un* entier
           contraintv
              (cont2 atte)
    fuse : *un* entier
    deco : *un* entier
           sorte-de attribut-obligatoire
           nférence
              (prop fuse)

Les structures de la classe non modifiée sont :
C1 : {imat : chaîne}
C2 : {imat : chaîne, aile : entier, deco : entier, (cont1 aile)}

C3 : {imat : chaîne, aile : entier, deco : entier, (notcont1 aile)}
C4 : {imat : chaîne, aile : entier, atte : entier, deco : entier, (cont1 aile), }
C5 : {imat : chaîne, aile : entier, atte : entier, deco : entier, (notcont1 aile)}
C6 : {imat : chaîne, atte : entier}
C7 : {imat : chaîne, deco : entier}
C8 : {imat : chaîne, deco : entier, atte : entier}

Les structures de la classe modifiée sont :
K1 : {imat : chaîne, deco : entier}
K2 : {imat : chaîne, deco : entier, atte : entier, (cont2 atte)}
K3 : {imat : chaîne, deco : entier, atte : entier, (notcont2 atte)}
K4 : {imat : chaîne, deco : entier, fuse : entier}
K5 : {imat : chaîne, deco : entier, atte : entier, fuse : entier, (cont2 atte)}
K6 : {imat : chaîne, deco : entier, atte : entier, fuse : entier, (notcont2 atte)}

Dans le premier paquet, nous avons l'attribut aile.
Dans le deuxième paquet, nous avons l'inférence (prop fuse).
Dans le troisième paquet, nous avons la contrainte (cont2 atte).
Dans le quatrième paquet, nous avons l'attribut deco.
Dans le cinquième paquet, nous avons la contrainte (cont1 aile).

Analyse des paquets pour quelques structures :
Structure C2 : L'attribut aile appartient à cette structure, donc cet attribut est supprimé dans toutes les instances de la structure. L'inférence (prop fuse) n'est pas calculable car l'attribut fuse n'appartient pas à C2. La contrainte (cont2 atte) n'est pas calculable, car l'attribut atte n'appartient pas à C2. L'attribut deco fait bien partie de la structure C2. Comme dans la structure C2 aucune contrainte faible n'est calculable, alors cette structure a une seule structure de reclassification possible : K1. Toutes les instances de C2 sont reclassifiées dans K1.

Structure C6 : L'attribut aile n'appartient pas à C6, donc la suppression de cet attribut ne modifie pas les instances de cette structure. L'inférence (prop fuse) n'est pas calculable. La structure C6 possède l'attribut atte, donc la contrainte (cont2 atte) est calculable. Cette contrainte est calculée pour toutes les instances de la structure. Certaines instances violeront cette contrainte et d'autres non. L'attribut deco ne fait pas partie de la structure, alors le système rejette toutes les instances de C6. L'utilisateur doit donner une valeur à l'attribut deco pour toutes ces instances. Notons, qu'il n'existe aucune structure de reclassification possible pour la structure C6.

Structure C8 : La suppression de l'attribut aile n'a pas d'impact sur les instances de C8. L'inférence (prop fuse) n'est pas calculable. La contrainte (cont2 atte) est calculée pour chaque instance. L'attribut deco est bien attribut de la structure C8. Les structures de reclassification possible de la structure C8 sont : K2 et K3. La liste des contraintes violées de la structure C8 est vide puisque la contrainte faible (cont1 aile) a été supprimée. Pour chaque instance, selon la valeur de la contrainte (cont2 atte), nous allons déterminer, parmi K2 et K3, quelle est sa nouvelle structure.

3.4 Conclusion

Afin d'aider l'utilisateur dans la conception d'une base, nous lui offrons deux nouvelles possibilités pour créer des classes : à partir d'une structure ou à partir d'une autre classe. Nous nous sommes peu intéressés à la répercussion de la création d'une classe sur le graphe d'héritage, ceci étant fait dans [ESC89], mais nous envisageons de coupler ces deux études.

Dans un deuxième temps, nous nous sommes surtout intéressés à l'impact de la mise à jour d'une classe sur ses instances. Nos choix ont été guidés par deux principes :

- optimiser le temps de mises à jour des instances; cela est fait en nous aidant des structures et des paquets,
- perdre le moins d'informations possibles sur les instances; à part dans le cas d'une suppression d'attribut, nous conservons toute information détenue sur une instance.

Notre solution est également interactive, car dans le cas où une instance ne répond plus aux critères de la nouvelle classe, l'utilisateur est seul décideur du devenir de cette instance (il peut la modifier ou la supprimer).

4 REALISATION

4.1 Introduction

Un premier prototype, élaborant les structures, a été développé à l'aide du générateur de systèmes experts : L'Experkit. Ce système est basé sur les règles de production. Il est composé d'une base de faits, d'une base de connaissances (les règles) et d'un moteur d'inférence qui permet l'interprétation des connaissances. Mais il s'est avéré que le moteur d'inférence, peu utilisé par le programme, amenait plus d'inconvénients que d'avantages. De plus, les temps de réponse n'étaient pas satisfaisants. Notre choix s'est alors porté sur Le_Lisp 15.2 [CHA86]. Le_Lisp a été choisi pour son côté interactif, sa souplesse pour une programmation incrémentale, et la facilité avec laquelle nous pouvons implanter des objets (d'ailleurs des primitives objets sont intégrées, mais n'ont pu être utilisées dans le cadre de notre étude, car elles ne correspondent pas aux primitives de notre modèle minimal). Ce choix est aussi motivé par le fait que notre étude est intégrée dans un projet où les différents modules sont développés en Le_Lisp.

L'implémentation a été faite sur un Macintosh II, en raison de son côté convivial et de la disponibilité de Le_Lisp sur cette machine.

Notre étude a abouti à la réalisation de deux prototypes : un pour CADB et un pour SHOOD. Aucune implémentation de SHOOD n'étant faite et ne disposant pas du modèle CADB, nous avons dû, pour les deux prototypes, implanter les fonctionnalités de base : création et modification des classes et des instances, avant de réaliser la programmation propre à l'étude. Dans les paragraphes suivants, nous insisterons surtout sur les aspects traitant de la dynamique.

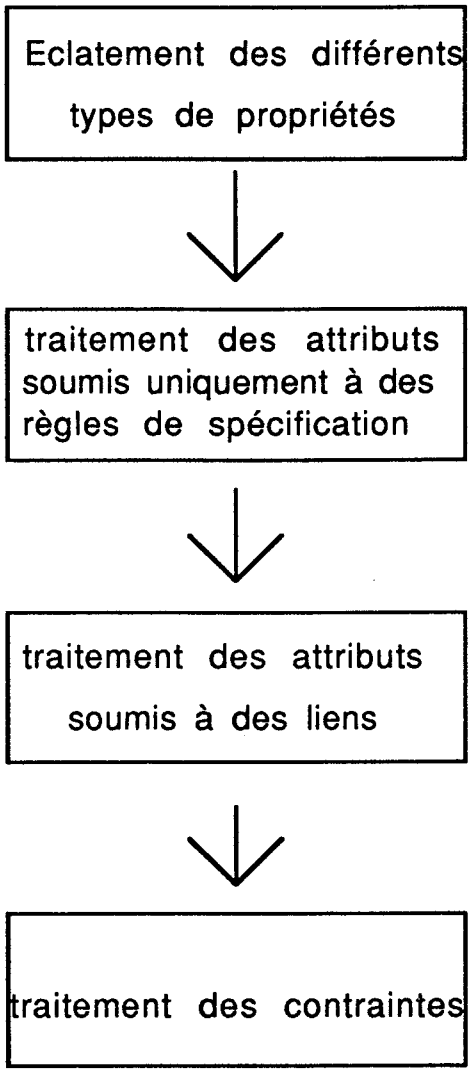
4.2 Dynamique des instances dans le modèle CADB

4.2.1 Création des structures

La création des structures se fait dès la création d'une classe. Nous rappelons que les structures sont créées uniquement pour les classes construites.
L'algorithme de création des structures est indépendant de l'algorithme de création d'une classe. Nous avons fait ce choix pour deux raisons :

- une classe créée n'est pas forcément génératrice de structures,
- nous pouvons ajouter facilement l'algorithme de création des structures à un modèle déjà implanté.

Pour générer les structures, nous employons un algorithme dont l'organigramme est le suivant:



Dans un premier temps, nous éclatons les différents types de propriétés. Nous séparons les attributs soumis uniquement à des règles de spécification, de ceux soumis à des liens et des contraintes.

Nous traitons en premier les attributs soumis uniquement à une règle de spécification. Nous générons un algorithme de fermeture ne prenant en compte que ces attributs et ajoutons à ces structures l'attribut obligatoire (la clé de l'instance).

Exemple de quelques structures créées, à cette étape, pour la classe Avion :

C1={imat : chaîne}

C2={imat : chaîne, aile : Aile}

C3={imat : chaîne, fuse : entier}

C4={imat : chaîne, fuse : entier, atte : entier} .....

En deuxième, nous traitons un par un tous les attributs de la classe soumis à un lien. Pour chaque structure générée par l'algorithme de fermeture, nous regardons si les arguments du

lien considéré sont présents ou non. Si tous les arguments du lien sont présents, nous ajoutons à la structure l'attribut concerné avec son type et son mode de calcul.

A cette étape nous complétons, par exemple, les structures C3 et C4 créées précédemment.

C3={imat : chaîne, fuse : entier, deco : entier, deco := prop (fuse)}

C4={imat : chaîne, fuse : entier, deco : entier, deco := prop (fuse), atte : entier}

Nous traitons en dernier lieu les contraintes, car les arguments de celles-ci peuvent être des attributs soumis soit uniquement à des règles de spécification, soit à des liens. Pour chaque structure comportant tous les arguments spécifiés dans une contrainte nous ajoutons cette contrainte et créons une structure similaire comportant la négation de la contrainte.

A cette étape, nous complétons, par exemple, la structure C4 et créons une structure C5.

C4={imat : chaîne, fuse : entier, deco : entier, deco := prop (fuse), atte : entier, super (deco atte)}

C5={imat : chaîne, fuse : entier, deco : entier, deco := prop (fuse), atte : entier, notsuper (deco atte)}

Le calcul des degrés de complétude et de cohérence se fait pendant le déroulement des trois dernières étapes.

Les structures sont représentées sous forme de P-liste. Le nom de cette P-liste est le nom de la classe, associé à un indice (numéro incrémenté). Cette P-liste contient les propriétés de la structure, ainsi que les degrés de complétude et de cohérence des objets de la structure.

Par exemple, la structure C3 est représentée sous la forme d'une p-liste de nom "avion3" et de forme :

(*propriété* ((imat chaîne) (fuse entier) (deco entier (prop (fuse))))
degré (3 1))

Le premier nombre associé à "degré" représente la cohérence et le second la complétude.

4.2.2 Création des instances

Pour la création d'une instance, l'utilisateur spécifie la classe de l'instance, puis le système propose tous les attributs de la classe (sauf les liens et les contraintes). L'utilisateur a la possibilité d'instancier ou non une propriété; les attributs non instanciés ne figurent pas dans l'instance. La clé est le seul attribut que l'utilisateur doit obligatoirement instancier. Le système calcule ensuite les liens et les contraintes quand tous les arguments de ceux-ci sont instanciés. Puis à partir du degré de cohérence, de la liste des attributs instanciés et de la liste des contraintes violées dans l'instance, il recherche à quelle structure appartient l'instance.

Rappelons, que les contraintes ne sont pas gardées au niveau des instances.

Prenons l'exemple d'un avion A

avion A

imat : A

fuse : 5

deco : 250

Cette instance est rattachée à la structure C3 précédente. Son degré de complétude est 3 et son degré de cohérence est 1.

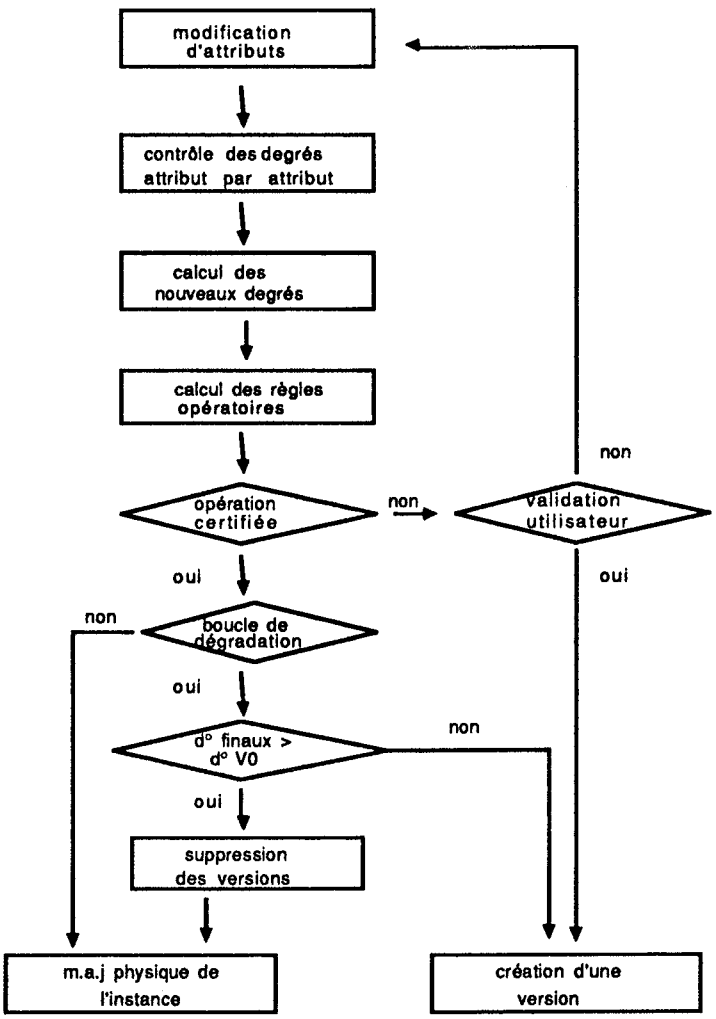
Une instance est représentée par une P-liste dont le nom est la clé de l'instance. La P-liste contient l'ensemble des propriétés instanciées (liens compris), le nom de la classe et le numéro de structure à laquelle appartient l'instance.

L'instance A est la P-liste de nom "A" et de forme :

(*propriété* ((imat A) (fuse 5) (deco 250)) *classe* avion *structure* 3)

4.2.3 Evolution des instances

Organigramme de la mise à jour d'une instance :



L'opération de mise à jour se fait sur une instance d'une classe donnée (annexe 1, p.90). Cette opération de mise à jour consiste soit en la modification de la valeur d'un attribut, soit en l'instanciation d'un attribut, soit en la désinstanciation d'un attribut.
L'utilisateur choisit un à un les attributs à mettre à jour en spécifiant leurs nouvelles valeurs. L'attribut clé est le seul attribut sur lequel aucune mise à jour n'est autorisée, du fait de son statut d'identifiant. L'utilisateur n'a accès ni aux liens, ni aux contraintes, ceux-ci étant calculés automatiquement par le système.

Afin de guider l'utilisateur dans sa mise à jour, le système calcule pour chaque attribut modifié les nouveaux degrés de complétude et de cohérence de l'instance. Si pour un attribut donné l'un et/ou l'autre des degrés diminuent, un message signale ce fait à l'utilisateur. Celui-ci peut donc suivre attribut par attribut l'évolution d'une instance. Même si l'opération est certifiée, l'utilisateur est averti qu'un attribut donné a pu dégrader l'instance momentanément. Par exemple, si l'utilisateur désinstancie une propriété, mais en instancie deux autres, le degré de complétude de l'instance finale a augmenté, mais la désinstanciation de l'attribut a quand même diminué le degré de complétude momentanément.

Le système calcule ensuite les nouveaux degrés de l'instance, en tenant compte de la modification de tous les attributs.

Par exemple si l'on modifie l'avion A en désinstanciant l'attribut fuse et en instanciant les attributs aile et atte, le système signale que la désinstanciation de l'attribut fuse diminue le degré de complétude de 2 (car l'attribut deco est alors aussi désinstancié puisque sa valeur dépend de l'attribut fuse), mais le système certifie quand même l'opération car les degrés de complétude et de cohérence finaux de l'instance sont égaux aux degrés initiaux de l'instance.
L'avion A devient , par exemple :
imat : A
aile : 10
atte : 200
Son degré de complétude est 3 et son degré de cohérence est 1.

Si une classe opératoire correspond à la classe de l'instance modifiée, alors une instance de la classe opératoire est créée.
Une classe opératoire est formée d'attributs (annexe 1, p.89). A chaque attribut est associée une fonction booléenne. Ces fonctions permettent de représenter les règles opératoires.
Un attribut d'une classe opératoire est de la forme :
nom attribut := (fonction booléenne (argument))
L'argument est soit un attribut, soit une fonction portant sur un ou plusieurs attributs. La fonction booléenne est en fait appliquée sur deux arguments. Le premier argument est calculé à partir des anciennes valeurs des attributs de l'instance et le deuxième argument est calculé à partir des nouvelles valeurs des attributs de l'instance.
Les fonctions booléennes ne sont pas toujours calculables à cause de la non instanciation de certains attributs de l'instance. Trois cas peuvent se présenter :

- Aucun des deux arguments n'est connu : dans ce cas nous considérons que la réponse de la fonction est "vrai". Nous ne voulons pas pénaliser l'utilisateur alors que nous n'avons aucun moyen de juger de la dégradation de l'instance.

- Le premier argument est connu, mais pas le deuxième argument : dans ce cas la réponse de la fonction est "faux". En effet, ce cas ne peut se produire que si, lors de la mise à jour de l'instance, des attributs ont été désinstanciés, donc l'instance ne s'améliore pas.

- Le premier argument n'est pas connu, mais le deuxième est connu: la réponse de la fonction est "vrai". Ce cas se produit quand de nouveaux attributs de l'instance ont été instanciés. Nous considérons donc qu'il y a eu amélioration.

Une instance de la classe opératoire possède donc des attributs dont les valeurs sont uniquement "vrai" ou faux".
Si l'instance d'une classe opératoire créée pour une instance modifiée possède un attribut ayant la valeur "faux" alors la mise à jour de l'instance modifiée n'est pas certifiée.
Si l'instance modifiée est versionnalisée alors l'instance de la classe opératoire est conservée sinon cette instance est supprimée.

Exemple :

La classe Avion.maj est la classe opératoire associée à la classe Avion.

```
Avion.maj
  augmfuse := (supeg fuse)
  dimdiff  := (infeg (diffe deco atte))
```

Augmfuse (augmentation de la longueur du fuselage) et dimdiff (diminution de la différence) sont des noms d'attributs donnés par l'utilisateur.
A l'attribut augmfuse est associée une fonction "(supeg fuse)". C'est une fonction booléenne qui délivre la valeur "vrai" si l'opération de mise à jour ne diminue pas la valeur de l'attribut fuse. Elle effectue une comparaison entre l'ancienne valeur de l'attribut fuse et la nouvelle valeur de cet attribut.
La fonction "infeg" de l'attribut dimdiff est appliquée à la fonction (diffe deco atte) qui calcule la différence entre la distance de décollage et la distance d'atterrissage. Si cette différence n'augmente pas, la fonction infeg renvoie le résultat "vrai". Il y a une comparaison entre l'ancienne différence entre la distance de décollage et celle d'atterrissage et la nouvelle différence.
Donc, pour qu'un avion s'améliore, il faut que ses degrés de complétude et de cohérence ne diminuent pas, que la longueur du fuselage reste stable ou augmente et que la différence entre la distance de décollage et d'atterrissage n'augmente pas.

Reprenons l'exemple de l'avion A :

```
avion A
  imat : A
  fuse : 5
  deco : 250
```

Si une mise à jour de cette instance consiste en la modification de l'attribut fuse par la valeur 3, le système ne certifie pas la mise à jour, bien que les degrés de complétude et de cohérence de l'instance n'aient pas changé, car cette modification ne satisfait pas la règle opératoire qui spécifie que la longueur du fuselage doit augmenter. La différence entre la distance de décollage et la distance d'atterrissage n'étant pas calculable pour l'instance A,

ni avant ni après sa modification, nous considérons que l'attribut dimdiff prend la valeur "vrai".

Un message signale à l'utilisateur la différence entre les degrés de l'instance finale et les degrés de l'instance initiale et affiche l'instance de la classe opératoire.

Si l'opération est certifiée le système valide automatiquement l'opération, sinon l'utilisateur a la possibilité de valider ou non l'opération (annexe 1, p.90). S'il ne valide pas l'opération, il peut modifier à nouveau les attributs de l'instance.

Si l'instance n'est pas versionnalisée (elle ne possède pas de boucle de dégradation) et si l'opération est certifiée, alors la mise à jour physique de l'instance est réalisée. Cette mise à jour consiste en la modification des valeurs des propriétés de l'instance, et en la modification du numéro de structure le cas échéant.

Si une instance n'est pas versionnalisée et qu'elle se dégrade, alors une boucle de dégradation est créée. Le système crée une version V0 qui sauvegarde les anciennes valeurs de l'instance; puis il crée une version V1 qui contient les nouvelles valeurs de l'instance et qui représente la version courante. Les valeurs de l'instance sont alors les valeurs de la version courante. La P-liste associée à l'instance, ne contient plus l'ensemble des propriétés de l'instance, mais contient le numéro de la version courante ainsi que le dernier numéro de version utilisé. Nous devons garder ces deux numéros de version car si l'utilisateur revient à une ancienne version, le numéro de la version courante ne correspond plus au dernier numéro de version.
En plus des versions, le système garde l'explication du passage d'une version à une autre. Cette explication est formée de l'instance de la classe opératoire (si elle existe), et de la différence entre les anciens et les nouveaux degrés de complétude et de cohérence de l'instance.
Une version est représentée sous forme d'une P-liste. Elle contient les propriétés et leurs valeurs, le numéro de la version précédente, le ou les numéros des versions suivantes et l'explication du passage à cette version.

Exemple :

Si l'utilisateur valide la mise à jour de l'instance A qui consiste en la modification de l'attribut fuse, son ancienne valeur étant 5 et sa nouvelle valeur 3, alors deux versions sont créées car l'opération n'est pas certifiée.

Version 0	Version 1
imat : A	imat : A
fuse : 5	fuse : 3
deco : 250	deco : 150

Représentation sous-forme de P-liste :
P-liste de nom A0 (pour la version 0)
(*propriété* ((imat A) (fuse 5) (deco 250)) *structure* 3 *versionsuivante* 1)

P-liste de nom A1 (pour la version 1)
(*propriété* ((imat A) (fuse 3) (deco 150)) *structure* 3 *versionprécédente* 0
explication (I1 (complétude 0) (cohérence 0)))

I1 est l'identificateur d'une instance de la classe opératoire. Cette instance nous permet de connaître la valeur des différents attributs de la classe opératoire, lors de la modification concernée. Dans notre cas l'instance I1 a pour valeur (augmfuse = faux , dimdiff = vrai). Les informations (complétude 0) et (cohérence 0) indiquent que les degrés de complétude et de cohérence de l'instance n'ont pas évolué.

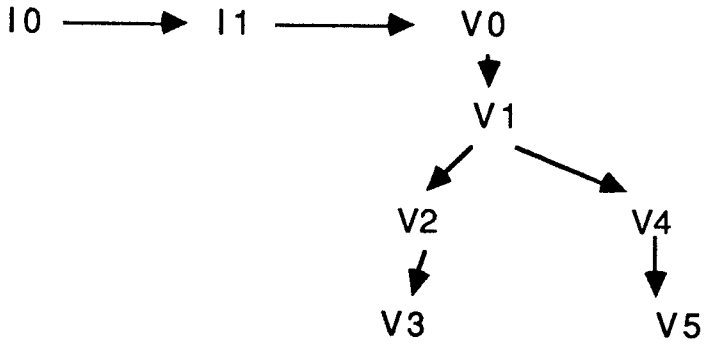
P-liste de l'instance A

(*classe* avion *dernièreversion* 1 *versioncourante* 1)

Une instance versionnalisée devient non versionnalisée uniquement si les nouveaux degrés de complétude et de cohérence de la version courante sont supérieurs ou égaux aux degrés de la version 0. Dans ce cas, toutes les versions de la boucle de dégradation sont supprimées. Dans le cas contraire, le système crée une nouvelle version.

Quand l'utilisateur choisit de revenir à une version (annexe 1, p.91), cette version devient la version courante. Le système génère alors un arbre qui est conservé tant que l'instance est versionnalisée. Cet arbre est visible à tout moment par l'utilisateur.

Exemple d'évolution d'une instance :



La création de l'instance est modélisée par le point I0. Le point I1 est atteint après modification de l'instance et certification de la mise à jour. Une autre opération certifiée est effectuée sur l'instance, elle devrait donner le point I3. Mais la modification du point I3, entraîne une dégradation de l'instance. Le point I3 devient alors la version V0, et la modification de ce point la version V1. Deux nouvelles opérations sont effectuées à partir de la version V1 : V2 et V3. L'utilisateur a ensuite décidé de reprendre à partir de la version V1. C'est pourquoi une nouvelle branche est créée à partir de V1, branche contenant V4 et V5. La version courante est, par exemple, la version V5.

4.3 Dynamique des instances dans le modèle SHOOD

4.3.1 Création des classes

Dans SHOOD, un attribut peut être soumis à plusieurs descripteurs. Cinq types de descripteurs sont utilisés :

- un : qui permet de déterminer le type de la valeur de l'attribut
- inférence : qui permet de définir une liste d'inférences
- contraintv : qui permet de définir une liste de contraintes faibles
- contrainte : qui permet de définir une liste de contraintes fortes
- sorte-de : permet de connaître le type de l'attribut. Ce type sert à spécifier, si la propriété est un attribut obligatoire ou une contrainte inter-attribut forte ou faible.

Exemple :

Avion
  nomclasse = avion
  propriété =
    imat : *un* chaîne
           sorte-de attribut-obligatoire
    aile : *un* entier
    fuse : *un* entier
    atte : *un* entier
           contraintv
              (contr atte)
    deco : *un* entier
           inférence
              (prop fuse)
              (fonc aile)
    (super deco atte) : *sorte-de* contrainte

4.3.2 Création des structures

Comme pour CADB, la création des structures se fait dès la création d'une classe. L'organigramme général, employé dans le cadre de la création des structures dans CADB, reste inchangé, mais les différents modules qui le composent ont été modifiés.

Dans l'éclatement des différents types de propriétés, nous séparons : les attributs obligatoires, les attributs soumis à des inférences, les attributs non soumis à des inférences et non obligatoires, et les contraintes qu'elles soient intra ou inter-attributs.

Dans un premier temps, nous élaborons la structure minimale, c'est-à-dire celle qui comporte tous les attributs obligatoires. Il est à noter que cette structure peut ne pas exister. Puis, nous générons un algorithme de fermeture, similaire à celui de CADB, sur tous les attributs.

Dans un deuxième temps, nous traitons une par une toutes les inférences de la classe. Si tous les arguments d'une inférence sont inclus dans la liste des attributs d'une structure, alors le

système ajoute l'attribut concerné par l'inférence dans la structure. Sinon, le système crée une structure ayant les mêmes attributs que la structure de départ et ajoute l'attribut concerné par l'inférence. Ceci permet de créer des structures où la valeur d'un attribut peut être donnée par l'utilisateur et non calculée par inférences.

Le module de traitement des contraintes se comporte comme le celui de CADB, mais ne crée des structures avec des contraintes inverses, que dans le cas de contraintes faibles.

Le calcul des degrés de cohérence et de complétude se fait pendant le déroulement des trois dernières étapes.

4.3.3 Création des instances

La création d'une instance dans SHOOD est plus ardue que dans CADB car un attribut possède plusieurs descripteurs.
Le système propose à l'utilisateur tous les attributs (même ceux qui sont soumis à des inférences), mais ne propose pas les contraintes.

Si l'utilisateur donne une valeur à un attribut soumis à des inférences, la valeur donnée par l'utilisateur est toujours prioritaire.
Si un attribut n'est pas instancié par l'utilisateur et s'il possède un descripteur "inférence", le système calcule alors la valeur de l'attribut à l'aide de ces inférences (annexe 1, p.89). Les inférences sont évaluées dans l'ordre de la liste d'inférences.

Exemple :

```
Avion
   nomclasse = avion
   propriété =
     imat  : un chaîne
             sorte-de attribut-obligatoire
     aile  : un entier
     fuse  : un entier
     deco  : un entier
             inférence
                (prop fuse)
                (fonc aile)
```

La méthode prop permet de calculer la distance de décollage en multipliant la longueur du fuselage par 50.
La méthode fonc permet de calculer la distance de décollage en multipliant la valeur de l'attribut aile par 100.

création d'une instance A1 :
imat : tango
aile : 5
fuse : 6
deco : 300
L' attribut deco a été calculé avec l'inférence prop (fuse).

création d'une instance A2 :
imat : toto
aile : 5
deco : 500
La longueur du fuselage n'étant pas connue, la distance de décollage est calculée à partir de l'attribut aile.

création d'une instance A3 :
imat : titi
aile : 5
fuse : 6
deco : 200
L'utilisateur a donné la valeur 200 à l'attribut deco, aucun calcul n'est effectué à partir des inférences.

Une instance est identifiée par un identificateur qui est indépendant de la clé (si elle existe). Une instance est sous-forme de P-liste dont le nom est l'identificateur. La P-liste est identique à celle de CADB.

Par exemple, la P-liste associée à l'instance A3 est de nom A3 et de forme :
(*propriété* ((imat titi) (aile 5) (fuse 6) (deco 200)) *structure* 3
classe avion)

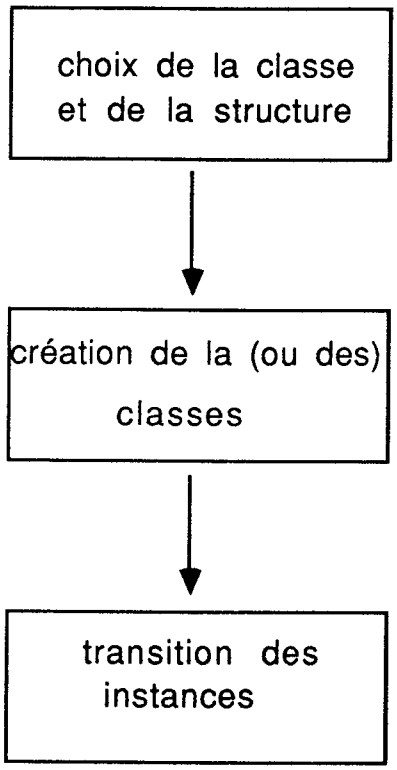
4.3.4 Evolution des instances

La gestion de l'évolution d'une instance est identique à CADB.

4.4 Dynamique des classes dans le modèle SHOOD

4.4.1 Création d'une classe à partir d'une structure

Organigramme de la création d'une classe à partir d'une structure :



L'utilisateur a la possibilité de visualiser les structures d'une classe, il peut aussi pour chaque structure connaître le pourcentage d'instances attachées à cette structure.

Si l'utilisateur spécifie les super-classes et les sous-classes de la nouvelle classe, alors le système crée la nouvelle classe sans analyser ses propriétés, il modifie le graphe d'héritage, puis il fait transiter les instances de la structure dans la nouvelle classe.

Si l'utilisateur désire conserver un lien entre la classe initiale et la classe finale, le système doit alors analyser les propriétés de la nouvelle classe, pour décider de la création d'une ou deux classes. Si la structure de départ contient au moins une contrainte violée alors le système crée deux classes : la classe finale et une classe système (annexe 2, p.92).
Le système modifie ensuite le graphe d'héritage en mettant à jour les super-classes et les sous-classes des classes concernées.

Exemple :

Reprenons l'exemple de la classe Personnesala qui est sous-classe de la classe Objet, et qui ne possède pas de sous-classe.

Classe Personnesala

nomclasse = personnesala

propriété =

nom : *un* chaîne

prénom : *un* chaîne

salaire : *un* réel

contraintv

(min salaire)

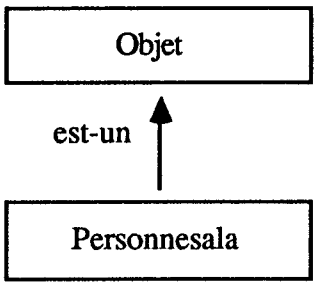
Si nous créons une classe Personne, contenant uniquement le nom et le prénom, à partir d'une structure Ci de la classe Personnesala. La classe Personne est une super-classe de la classe Personnesala.

La liste des sous-classes de la classe Objet est à modifier : nous enlevons la classe Personnesala et ajoutons la classe Personne.

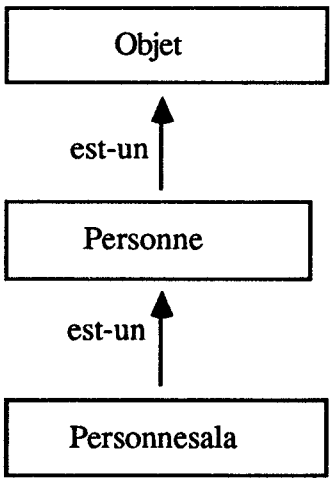
La super-classe de la classe Personnesala est maintenant Personne.

La classe Personne a pour super-classe la classe Objet.

ancien graphe d'héritage



nouveau graphe d'héritage



L'utilisateur désire créer la classe Personneexpl, à partir de la structure Cj de la classe Personnesala.

Cj :{nom : chaîne, prénom : chaîne, salaire : réel, notmin (salaire)}

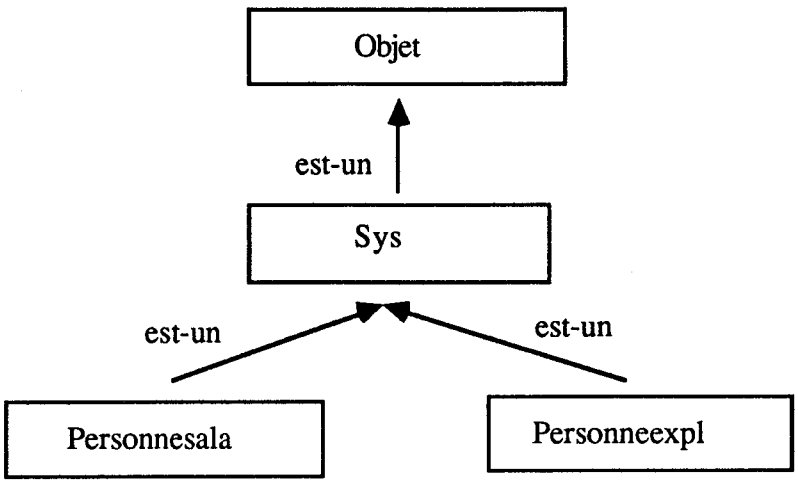
La liste des sous-classes de la classe Objet doit être modifiée : le système supprime la classe Personnesala de la liste et ajoute la classe SYS (qui est la classe créée par le système).

La super-classe de la classe Sys est Objet, ses sous-classes sont : Personnesala et Personneexpl.

La super-classe de la classe Personnesala devient Sys.

La super-classe de la classe Personneexpl est Sys.

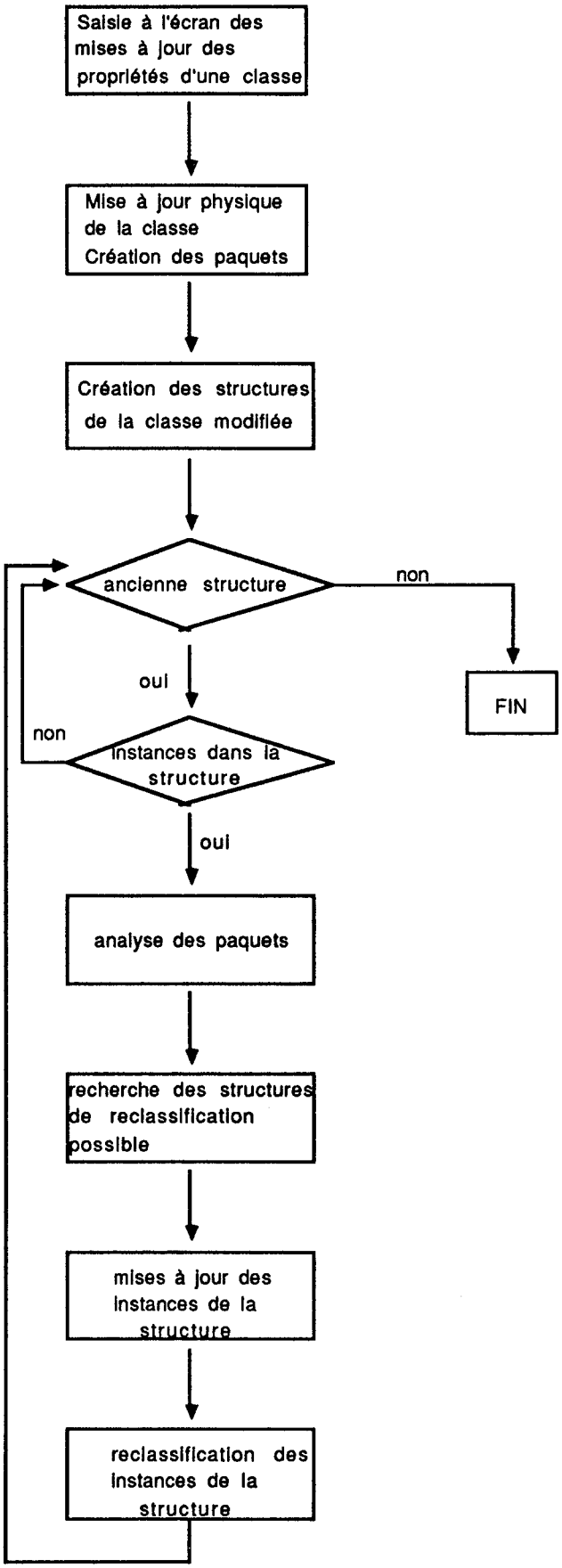
nouveau graphe d'héritage



En dernier lieu, le système transfert toutes les instances de la structure de départ dans la structure maximale de la nouvelle classe.

4.4.2 Mise à jour d'une classe

Organigramme de la mise à jour d'une classe :


Saisie à l'écran des mises à jour des propriétés d'une classe
Mise à jour physique de la classe
Création des paquets
Création des structures de la classe modifiée
ancienne structure
non
FIN
oui
non
instances dans la structure
oui
analyse des paquets
recherche des structures de reclassification possible
mises à jour des instances de la structure
reclassification des instances de la structure

Nous allons détailler la création des paquets, en fonction des différentes mises à jour. Nous parlerons uniquement des mises à jour qui ont une influence sur les paquets.

Nous rapellons ici le contenu des différents paquets :
Le premier paquet contient tous les attributs supprimés lors de la mise à jour de la classe.
Le deuxième paquet contient les nouvelles inférences ajoutées aux attributs.
Le troisième paquet contient toutes les contraintes ajoutées, que ce soit des contraintes intra ou inter-attributs.
Le quatrième paquet contient les nouveaux attributs obligatoires.
Le cinquième paquet contient les contraintes faibles qui ont été supprimées lors de la mise à jour de la classe.

La suppression d'un attribut entraîne son ajout dans le premier paquet. Si cet attribut est argument d'une contrainte faible, alors cette contrainte est ajoutée dans le cinquième paquet, car cette contrainte doit être supprimée.
La suppression d'une contrainte inter-attributs faible entraîne son ajout dans le cinquième paquet.

L'ajout d'une contrainte inter-attributs entraîne l'ajout de cette contrainte dans le troisième paquet.

L'ajout d'un attribut obligatoire entraîne son ajout dans le quatrième paquet.
Si l'attribut ajouté possède une inférence, alors cette inférence est ajoutée au deuxième paquet.
Si l'attribut ajouté possède une contrainte, alors cette contrainte est ajoutée au troisième paquet.

Modification d'un attribut :

- l'ajout d'inférences dans un attribut entraîne l'ajout de ces inférences dans le deuxième paquet.
- l'ajout de contraintes dans un attribut entraîne l'ajout de ces contraintes dans le troisième paquet
- la suppression de contraintes faibles dans un attribut entraîne l'ajout de ces contraintes dans le cinquième paquet
- si l'attribut est rendu obligatoire, alors l'attribut est ajouté dans le quatrième paquet.

L'analyse des paquets se fait structure par structure. Si une structure ne possède pas d'instances, alors aucune analyse des paquets n'est faite. Pour chaque structure, nous définissons les attributs à supprimer dans les instances, les inférences et les contraintes à calculer. Si un attribut obligatoire n'est pas présent dans les attributs de la structure, alors toutes les instances de la structure sont ajoutées à la liste des instances rejetées (annexe 2, p.93). Le système génère ensuite la liste des contraintes violées dans la structure.

Le système recherche ensuite toutes les structures possibles de reclassification.

Pour chaque instance, nous supprimons les attributs à supprimer, nous calculons les inférences et les contraintes. Si une contrainte forte est violée dans une instance, l'instance est ajoutée à la liste des instances rejetées.

Une instance non rejetée est ensuite automatiquement classifiée dans une des structures possibles de reclassification. Les instances rejetées sont uniquement modifiées, mais pas reclassifiées. Elles le seront après modification de l'utilisateur.

Après l'analyse de toutes les structures, le système propose à l'utilisateur toutes les instances rejetées. Ces instances sont gérées comme s'il s'agissait d'une modification d'instance, sans le calcul d'amélioration évidemment.

4.5 Conclusion

La réalisation d'un prototype pour CADB a permis de valider l'utilisation des structures. Celui de Sherpa valide également certains concepts minimaux de SHOOD. Nous envisageons par la suite une extension de ce prototype en tenant compte de tous les concepts de SHOOD et en le couplant avec d'autres prototypes élaborés dans le cadre de Sherpa. Cette extension nous permettra, par exemple, de tenir compte de la méta-circularité du système, d'intégrer les objets composites, de mettre à jour tout le graphe d'héritage lors de la création ou la modification d'une classe ...

Pour représenter les objets nous avons utilisé des P_listes, nous pensons que l'utilisation de vecteurs rendrait plus performant le prototype.
Nous envisageons également une création de classes et d'instances à partir d'un fichier, afin que l'utilisateur puisse rapidement créer un nombre de classes et d'instances important.
Une utilisation d'AIDA (environnement graphique en Le-Lisp), permettra d'offrir à l'utilisateur une interface plus conviviale (écran pleine page ...).

CONCLUSION

Les bases de connaissances, dans les domaines de la CAO et de l'intelligence artificielle, ne sont pas statiques, à tout instant une classe ou une instance peut être modifiée. En considérant ce besoin, nous avons envisagé, dans notre étude, une nouvelle approche de la gestion de la dynamique des instances et des classes dans les systèmes de représentation de connaissances et en particulier dans le système Sherpa. Cette étude a pour base la notion de structure. Une structure représente un état possible d'une instance. C'est un concept original qui permet de gérer systématiquement la cohérence et la complétude des objets [FAV89].

Les structures sont créées à l'aide de règles heuristiques. A partir d'une relation d'ordre établie entre ces structures, nous définissons les concepts d'amélioration et de dégradation d'une instance. Des boucles de dégradation permettent de gérer l'évolution des instances et offrent à l'utilisateur la possibilité de revenir sur la conception d'un objet. Ces boucles de dégradation sont notamment intéressantes dans le cadre d'applications de biométrie ou de CAO.

Nous avons montré, en étudiant deux modèles différents CADB et SHOOD, que l'utilisation des structures est souple et facilement adaptable : les règles de génération des structures changent d'un modèle à l'autre, mais la gestion de l'évolution des instances est identique pour les deux modèles.

Les structures sont également utilisées pour la gestion de la dynamique des classes. Nous pouvons notamment créer des classes à partir de structures, ceci nous permet d'introduire de nouveaux concepts dans la base et donc d'affiner le graphe d'héritage. Nous nous aidons des structures pour optimiser la mise à jour des instances lors de la modification d'une classe. En effet, les instances ne sont pas traitées une par une, mais structure par structure. L'utilisation de "paquets" lors de la mise à jour d'une classe concourt également à optimiser la méthode de conversion.

La réalisation d'un prototype en Le-Lisp, sur Macintosh II, a permis de valider notre étude ainsi qu'une partie des concepts du modèle SHOOD. Par la suite, nous envisageons une extension de ce prototype en tenant compte de tous les concepts minimaux de SHOOD.

L'utilisation des structures peut être optimisée en définissant un graphe formé des structures d'une ou plusieurs classes. Ce graphe nous permet d'optimiser la classification d'une instance. En effet, pour classifier une instance dans sa structure maximale, seul un nombre restreint de branches sont à explorer, c'est-à-dire que la recherche se fait sur un nombre restreint de structures.

Nous avons étudié la modification d'une instance à l'intérieur d'une classe. Une extension possible est la gestion des instances qui changent de classe après leur mise à jour.

L'étude peut être approfondie au niveau de la notion de complétude. Dans le mémoire, nous avons toujours considéré la valeur des attributs comme des types simples : entier, chaîne .... Il faut traiter le cas où la valeur d'un attribut est une instance d'une classe complexe, par exemple la valeur de l'attribut enfant dans la classe Personne peut être de type personne. Dans ce cas, un attribut peut n'être instancié que partiellement, il est donc difficile de déterminer le degré de complétude global d'une instance.

Une utilisation future des structures est envisagée dans le cadre de la gestion des objets composites dans Sherpa [JEA89].

BIBLIOGRAPHIE

[BAR87] BANERJEE J., KIM W.
Semantics and implementation of schema evolution in object-oriented databases
Proc. ACM SIGMOD Conference, San Fransisco (CA), Mai 1987

[BOB77] BOBROW D.G., WINOGRAD T.
An overview of KRL, a Knowledge Representation Language
Cognitive Science, vol. 1, **n° 1**, 1977

[CHA86] CHAILLOUX J.
Le_Lisp 15.2 : manuel de référence
Rapport technique INRIA, 1986

[DAN88] DANFORTH S., TOMLINSON C.
Type theories and object-oriented programming
ACM Computing Surveys, Vol. 20, **n° 1**, Mars 1988

[ESC89] ESCAMILLA J.
Mécanismes d'héritage dans les bases de connaissances orientées objet
Rapport de DEA, INPG, 1989

[FAV89] FAVIER V.
Manipulation d'objets dynamiques dans les bases de connaissances
Article soumis à MICAD 90

[GOL83] GOLDBERG A., ROBSON D.
Smalltalk-80, the language and its implementation
Addison-Wesley, Reading, Massachusetts, 1983

[HEW75] HEWITT C.E, SMITH B.
A Plasma primer
MIT Artificial Intelligence Laboratory, Septembre 1975

[JEA89] JEAN P.
Manipulation d'objets composites dans les bases de connaissances
Rapport de DEA, INPG,1989

[LEC88] LECLUSE C., RICHARD P., VELEZ F.
O2 un modèle de données orienté-objet
4e journées INRIA Bases de Données Avancées, Benodet, Mai 1988

[MIN75] MINSKY M.
A Framework for Representing Knowledge
The Psychology of Computer Vision
P. Winston (ED.), McGraw-Hill, New York, 1975

[NGU87] NGUYEN G.T., RIEU D.
Manipulation d'objets dynamiques dans les bases de données
Actes Journées AFCET, "Des bases de données aux bases de connaissances",
Sophia-Antipolis, Septembre 1987

[NGU89a] NGUYEN G.T., RIEU D.
Schema evolution in object-oriented database systems
Data & Knowledge Engineering, Elsevier Science Publ. 1989

[NGU89b] NGUYEN G.T., RIEU D.
Schema change propagation in object-oriented databases
Proc. XIth World Computer Congress, IFIP Congress '89,
San Francisco (Ca.), Août 1989

[PEN87] PENNEY D.J, STEIN J
Class modification in the GemStone object-oriented DBM
Proc. OOPSLA '87 Conference, Orlando (Florida), Octobre 1987

[REC87] RECHENMANN F.
SHIRKA : système de gestion de bases de connaissances centrées objets, manuel de référence.
INRIA/ARTEMIS, Grenoble, 1987

[REC88] RECHENMANN F., NGUYEN G.T.
Sherpa : dynamique des bases de connaissances
Proposition de projet INRIA - IMAG, juin 1988

[RIE85] RIEU D.
Modèle et fonctionnalités d'un SGBD pour les applications CAO
Thèse de Doctorat INPG, Juillet 1985

[RIE86] RIEU D., NGUYEN G.T.
Semantics of CAD objects for generalized databases
Proc. 23rd Design Automation Conference, Las Vegas (Nevada), juin 1986

[RIE87] RIEU D., NGUYEN G.T.
Contrôle automatique de cohérence sur des objets dynamiques
3e Journées INRIA Base de Données Avancées
Port-Camargue, Mai 1987

[ROB77] ROBERTS R.B., GOLDSTEIN I.P.
The FRL manual
Technical Report Memo 409, MIT, AI Lab., Cambridge, MA, 1977

[SKA86] SKARRA A.H, ZDONIK S.B
The management of changing types in an object-oriented database
Proc. OOPSLA '86 Conf, Portland (Oregon), Septembre 1986

[STE86] STEFIK M., BOBROW D.G
Object oriented programming: themes and variations
The AI magazine, Janvier 1986

ANNEXE 1

DYNAMIQUE DES INSTANCES DANS SHOOD

Les commentaires sont en italique, les réponses fournies par l'utilisateur sont en gras, le texte affiché par le système est en caractère normal.

;; nous avons créé une classe avion

visualisation
1- d'une classe
2- des structures d'une classe
3- des pourcentages d'instances par structure
9- fin
1

donner le nom de la classe à visualiser
avion
nombre d'instances = 0
propriete =
imat: sorte-de attribut-obligatoire
un chaîne
aile: un entier
fuse: un entier
atte: contraitv ((contr atte))
un entier
deco: inférence ((prop fuse) (fonc aile))
un entier
(super deco atte): sorte-de contrainte
sous = ()
super = (objet)

;; exemple d'une structure de la classe avion

degré = (4 1)
propriete =
(super deco atte) : contrainte
(contr atte) : contraintv
deco : entier
atte : entier
imat : chaîne

;; nous allons créer une classe opératoire, de nom Avion.maj, associée à la classe Avion

creation d'une classe
1- simple
2- à partir d'une structure
3- à partir d'une classe
1
classe avec structure o/n : **n**
nom de la classe : **avion.maj**
super : **(objet)**
sous : **()**
propriete :
attribut : **augmfuse**
descripteur : **inférence**
valeur descripteur : **(supeq fuse)**
descripteur : **fin**
attribut : **fin**

donner le nom de la classe à visualiser
avion.maj
propriete =
augmfuse: inférence (supeq fuse)
sous = ()
super = (objet)

;; Nous allons créer une instance A

classe de l'instance : **avion**
identificateur de l'instance : **A**
imat : **tango**
aile : **-**
fuse : **5**
atte : **-**
deco : **-**
;; nous n'avons pas donné de valeurs aux attributs : aile, atte, deco

identificateur de l'instance : a
imat : tango
fuse : 5
deco : 250

degré de complétude de l'instance : 3
degré de cohérence de l'instance : 1

*;; La valeur de l'attribut deco a été calculée à partir de l'attribut fuse (5*50)*

;; nous allons modifier cette instance

modification
1- d'une classe
2- d'une instance
3- retour a une ancienne version
4- visualisation d'une boucle de dégradation
9- fin
2

donner le nom de la classe : **avion**
nom de l'instance : **a**
attribut à modifier : **atte**
nouvelle valeur : **60**
attribut à modifier : **fin**
la m.à.j de l'attribut atte diminue la cohérence

conséquence de la mise à jour
((augmfuse . vrai) (complétude . 2) (cohérence . -1))

l'objet s'est dégradé
validation de cet objet o/n : **o**
version créee : 1
identificateur de l'instance : a
imat : tango
fuse : 5
atte : 60
deco : 250
(contr atte) : faux
(super deco atte) : vrai

degré de complétude de l'instance : 5
degré de cohérence de l'instance : 0

;; nous modifions à nouveau l'avion A

donner le nom de la classe : **avion**
nom de l'instance : **a**
attribut à modifier : **fuse**
nouvelle valeur : **4**
attribut à modifier : **aile**
nouvelle valeur : **6**
attribut à modifier : **fin**
dégradation de l'attribut augmfuse

conséquence de la mise à jour
((augmfuse . faux) (complétude . 1) (cohérence . 0))

l'objet s'est dégradé
validation de cet objet o/n : **o**
version créee : 2
identificateur de l'instance : a
imat : tango
aile : 6
fuse : 4
atte : 60
deco : 250
(contr atte) : faux
(super deco atte) : vrai

degré de complétude de l'instance : 6
degré de cohérence de l'instance : 0

;; nous décidons de repartir de la version 1

nom de la classe : **avion**
nom de l'instance : **a**
nombre de versions : 2
numero version ? : **1**

identificateur de l'instance : a1
imat : tango
fuse : 5
atte : 60
deco : 250
(contr atte) : faux
(super deco atte) : vrai

degré de complétude de l'instance : 5
degré de cohérence de l'instance : 0

; nous modifions l'avion A à partir de la version 1

donner le nom de la classe : **avion**
nom de l'instance : **a**
attribut à modifier : **fuse**
nouvelle valeur : **1**
attribut à modifier : **fin**
dégradation de l'attribut augmfuse par rapport à a1

conséquence de la mise à jour
((augmfuse . faux) (complétude . 0) (cohérence . 0))

l'objet s'est dégradé
validation de cet objet o/n : **o**
version créee : 3
identificateur de l'instance : a
imat : tango
fuse : 1
atte : 60
deco : 250
(contr atte) : faux
(super deco atte) : vrai

degré de complétude de l'instance : 5
degré de cohérence de l'instance : 0

;; visualisation de la boucle de dégradation de A

boucle de dégradation de l'instance A
v0

.....v1 ((augmfuse . vrai) (complétude . 2) (cohérence . -1))

.....v2 ((augmfuse . faux) (complétude . 1) (cohérence . 0))

.....v3 ((augmfuse . faux) (complétude . 0) (cohérence . 0))

version courante : 3

ANNEXE 2

DYNAMIQUE DES CLASSES DANS SHOOD

;; Création d'une classe à partir d'une structure de la classe Avion, par exemple la structure où se trouve l'avion A

nom de la classe initiale : **avion**
numero de la structure : **23**
nom de la classe a creer : **nouvel-avion**

;; visualisation de la classe nouvel-avion

```
nombre d'instances = 1
sous = ()
super = (sysavion)
propriete =
   (super deco atte): sorte-de   contrainte
   deco: inférence   ((prop fuse) (fonc aile))
        un   entier
   atte: contraintv   ((negat (contr atte)))        ;;; c'est la négation de la contrainte "contr atte"
        un   entier
   fuse: un   entier
   imat: sorte-de   attribut-obligatoire
        un   chaîne
```

;; visualisation de la classe système : sysavion

```
nombre d'instances = 0
sous = (nouvel-avion avion)
super = (objet)
propriete =
   (super deco atte): sorte-de   contrainte
   deco: inférence   ((prop fuse) (fonc aile))
        un   entier
   atte: un   entier
   fuse: un   entier
   imat: sorte-de   attribut-obligatoire
        un   chaîne
```

;; modification de la classe avion : transformation de l'attribut aile en attriubt-obligatoire et
;; suppression de l'attribut fuse

nom de la classe à modifier : **avion**
attribut à modifier
nom de l'attribut : **aile**
nouveau nom de l'attribut : **aile**
descripteur à modifier
nom du descripteur : **fin**
descripteur à ajouter
nom du descripteur : **sorte-de** *;; nous transformons l'attribut aile,en attribut obligatoire*
valeur : **attribut-obligatoire**
nom du descripteur : **fin**
nom de l'attribut : **fuse**
nouveau nom de l'attribut : - *;;; nous supprimons l'attribut fuse*
nom de l'attribut : **fin**
attribut à ajouter : **fin**

veuillez modifier l'instance a, qui ne répond plus aux attributs de la classe avion
;;; le système rejette l'instance a, qui n'a pas de valeur pour l'attribut aile

attribut à modifier : **aile**
nouvelle valeur : **90**
attribut à modifier : **fin**

;;; La nouvelle instance est :

identificateur de l'instance : a
aile : 90
atte : 60
imat : tango
deco : 250
(super deco atte) : vrai
(contr atte) : faux

degré de complétude de l'instance : 5
degré de cohérence de l'instance : 0

;; la nouvelle classe Avion est :

nombre d'instances = 1
propriete =
imat: sorte-de attribut-obligatoire
un chaîne
aile: sorte-de attribut-obligatoire
un entier
atte: contraintv ((contr atte))
un entier
deco: inférence ((fonc aile))
un entier
(super deco atte): sorte-de contrainte
sous = ()
super = (objet)

RESUME

Cette étude s'inscrit dans le cadre du projet Sherpa. Ce projet a pour but de concevoir un système de représentation de connaissances basé sur le concept d'objet. L'axe privilégié du projet concerne la gestion de la dynamique des données et des connaissances.
L'étude présentée s'intéresse plus particulièrement à la **dynamique des instances** et la **dynamique des classes**.

La dynamique d'une instance est gérée grâce à des structures. Celles-ci reflètent les différents états possibles d'incomplétude et d'incohérence des instances. Ces structures sont créées à l'aide de règles heuristiques. Si une instance se détériore, une boucle de dégradation permet de gérer cette détérioration. Une instance se détériore si elle devient moins complète et/ou moins cohérente. Cependant, l'utilisateur a la possibilité d'affiner pour une application donnée le concept d'amélioration et de détérioration d'une instance.

La dynamique des classes comprend leur création et leur mise à jour. Plusieurs façons de créer une classe sont envisagées, notamment la création d'une classe à partir d'une structure. Nous étudions, plus particulièrement, l'optimisation de la mise à jour et de la reclassification des instances d'une classe modifiée.

Mots clés : objets - connaissances - dynamique - instances - classes - complétude - cohérence