Gestion de l'évolution dans les bases de connaissances : une approche par les règles

Fethi Bounaas

THESE

présentée par

BOUNAAS Fethi

pour obtenir le titre de

Docteur de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 Mars 1992)

Spécialité : **INFORMATIQUE**

Gestion de l'évolution dans les bases de connaissances : une approche par les règles

---

Date de soutenance : *11 Octobre 1995*

Composition du jury :

Président :	*Jean-Pierre Verjus*
Rapporteurs :	*Claude Chrisment*
	Mourad C. Oussalah
Examinateur :	*Jacques Mossière*
Directeur de thèse :	*Gia Toan Nguyen*

Thèse préparée au sein du Laboratoire de Génie Informatique
et de l'INRIA Rhône-Alpes

A mes parents

Je tiens à remercier :

Monsieur *Jean-Pierre Verjus*, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de L'INRIA Rhône-Alpes, de m'avoir fait l'honneur de présider le jury de cette thèse.

Monsieur *Claude Chrisment*, Professeur à l'Université Paul Sabatier de Toulouse, d'avoir accepté de juger ce travail et pour la lecture détaillée du mémoire.

Monsieur *Mourad Oussalah*, Professeur à l'Ecole des Mines d'Alès, d'avoir accepté d'être rapporteur de ma thèse. Je le remercie également pour l'intérêt qu'il a porté à mon travail.

Monsieur *Jacques Mossière*, Professeur à l'Institut National Polytechnique de Grenoble, de participer au jury de cette thèse.

Monsieur *Gia Toan Nguyen*, Directeur de Recherche INRIA, d'avoir accepté de m'accueillir dans son équipe et d'avoir dirigé ce travail. Je le remercie également pour l'expérience acquise durant cette thèse.

Je tiens à exprimer toute ma gratitude aux membres de l'équipe *SHOOD* : *Yasmina*, *Annie*, *Karine*, *Jérôme*, *Pascal*, *Stéphane* et *Emmanuel* pour leur amitié, leur bonne humeur et leur aide à la rédaction de cette thèse. Ainsi qu'aux "ex-Shoodiens" : *Dominique*, *Hélène*, *Marie-Pierre*, *François*, *Pierre*, *Pépé* et *Jeff*. Sans oublier *Leïla*, *Martine*, *Gilles* et *Jean-Luc* qui ont contribué de près ou de loin à l'aboutissement de ce travail.

Des remerciements tous particuliers à toute ma famille et mes amis pour m'avoir encouragé et soutenu pendant toutes ces années, spécialement à ma sœur *Faïza* et à mon frère *Sofiane*.

Résumé

Le travail présenté dans cette thèse aborde les problèmes de gestion de l'évolution des schémas et des objets dans les systèmes à base de connaissances. Ce travail a été réalisé dans le cadre du système de représentation de connaissances à objets SHOOD.

Cette étude tente d'apporter des solutions aux problèmes d'extensibilité et de réutilisation des mécanismes d'évolution. Nous proposons un système d'évolution permettant la définition et la mise en œuvre de la dynamique. Cette mise en œuvre est réalisée par un ensemble de mécanismes tels que la classification d'instances et les règles actives ou règles ECA (Evénement, Condition, Action), ainsi qu'un ensemble d'opérations de manipulation : le support d'évolution. Nous proposons de même un mécanisme de règles et de stratégies d'évolution pour permettre une expression déclarative des contraintes d'évolution de structures ou de données. Le concepteur peut définir des stratégies pour regrouper ces contraintes. Suivant ses besoins, il peut faire cohabiter plusieurs stratégies, mais une seule sera active à un moment donné. Ce mécanisme de règles et de stratégies d'évolution est principalement développé en établissant une correspondance entre une règle d'évolution et plusieurs règles ECA, ainsi qu'une correspondance entre une stratégie et une base de règles ECA.

Mots clefs : Dynamique des objets, evolution de schémas, règles actives, stratégie d'évolution, classification d'instances, représentation de connaissances.

Abstract

The work presented in this thesis concerns schema and object evolution in knowledge based systems. It is realized in the framework of the knowledge representation system Shood.

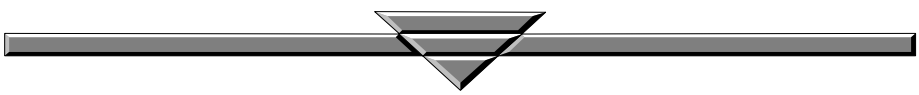
We try to solve extensibility and reuse problems for evolution mechanisms. We propose an evolution system which allows the definition and the management of the dynamics. This management is realized by a set of mechanisms, such as instance classification, active rules, and a set of modification operations called the evolution support. We propose also a mechanism based on evolution rules and strategies to allow the declarative expression of constraints for structures and data evolution. The designer can define strategies to organize these constraints. According to his needs, he can define multiple strategies, but only one will be active at a given moment. The mechanism implementing evolution rules and strategies is developped mainly by mapping between an evolution rule and several active rules, and between a strategy and a rule base.

Keywords: Dynamics of objects, schema evolution, active rules, evolution strategies, instance classification, knowledge representation.

CHAPITRE 1

INTRODUCTION

PROBLEMATIQUE 1
CONTRIBUTION 2
PLAN DE LA THESE 4



CHAPITRE 2

UN ETAT DE L'ART SUR L'EVOLUTION

2.1 COMMENT SUPPORTER L'EVOLUTION 7

2.2 EVOLUTION DE SCHEMAS 8

2.2.1 Gestion de l'évolution de schémas 8

2.2.1.1 Correction 8

2.2.1.2 Versionnement 14

2.2.1.3 Réorganisation de la hiérarchie des classes 20

2.2.2 Adaptation des instances 24

2.2.2.1 Conversion 24

2.2.2.2 Emulation 25

2.2.2.3 Versionnement 25

2.2.3 Bilan 26

2.3 EVOLUTION D'INSTANCES 27

2.3.1 Contraintes d'intégrité 27

2.3.1.1 Typologie des contraintes 28

2.3.1.2 Contrôle de l'intégrité 28

2.3.1.3 Les travaux sur les contraintes d'intégrité 29

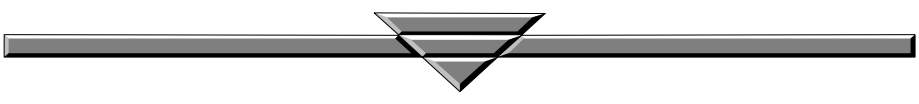
2.3.1.4 Conclusion sur les contraintes d'intégrité 29

2.3.2 Les versions 29

2.3.3 La classification d'instances 31

2.3.4 Bilan 33

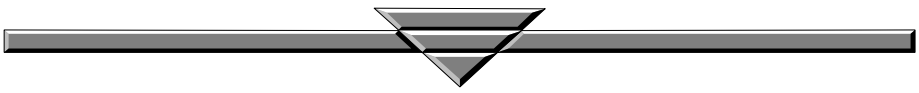
2.4 CONCLUSION 34



CHAPITRE 3

LE MODELE SHOOD

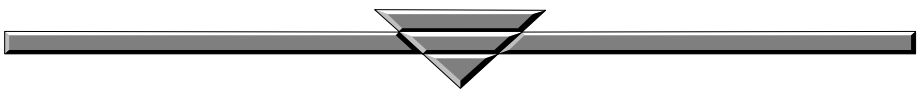
3.1 LES CONCEPTS DE BASE 35
3.2 LES ATTRIBUTS 37
3.2.1 Les noms des attributs 37
3.2.2 Les descripteurs 38
3.2.2.1 Descripteur de type 39
3.2.2.2 Descripteur d'inférence 40
3.2.2.3 Descripteur de contraintes 41
3.2.3 Les attributs obligatoires 41
3.2.4 Les attributs pré-déclarés 41
3.2.5 Les relations sémantiques 42
3.2.5.1 Lien de composition 43
3.2.5.2 Liens de dépendance 43
3.3 LES METHODES 45
3.3.1 Shood et les fonctions génériques 45
3.3.2 Les méthodes sont des classes 45
3.3.3 La sélection des méthodes 47
3.3.4 La coopération des méthodes 48
3.4 LES LIENS INTER-CLASSES 48
3.4.1 La spécialisation 48
3.4.2 La disjonction 49
3.5 LE LIEN D'INSTANCIATION 50
3.5.1 L'instanciation 50
3.5.2 La multi-instanciation 51
3.6 LA META-CIRCULARITE 52
3.7 SYNTHESE DU MODELE 56



CHAPITRE 4

LE SYSTEME D'EVOLUTION DE SHOOD

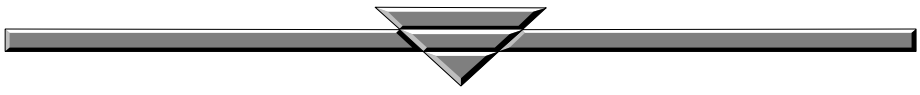
4.1 MOTIVATIONS 59
4.2 LE SUPPORT D'EVOLUTION 63
4.2.1 Les invariants du modèle 64
4.2.2 Les principes de propagation 64
4.2.3 Taxonomie des opérations 66
4.2.4 Sémantique des opérations 67
4.2.4.1 Opérations sur les classes 67
4.2.4.2 Opérations sur les instances 71
4.2.5 Réalisation 73
4.3 CONCLUSION 75



CHAPITRE 5

LE MECANISME DE CLASSIFICATION D'OBJETS EVOLUTIFS

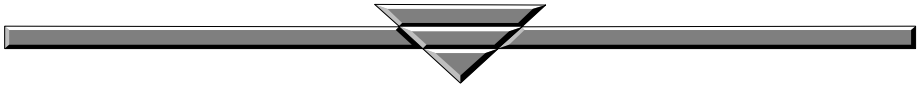
5.1 ELEMENTS FONDAMENTAUX 78
5.1.1 Multiplicité des inférences 78
5.1.2 Multi-instanciation 80
5.2 LIMITES DE LA REPRESENTATION PAR UN OBJET INFORMATIQUE 82
5.3 DESCRIPTION INFORMELLE DE LA CLASSIFICATION 83
5.4 GENERATION D'OBJETS INFORMATIQUES 85
5.4.1 La multiplicité des inférences 85
5.4.2 La gestion d'un attribut sans valeur 87
5.4.3 La disjonction 88
5.5 CONTEXTE DE VIE 89
5.6 CONCLUSION 91



CHAPITRE 6

LES MODELES A BASE DE REGLES

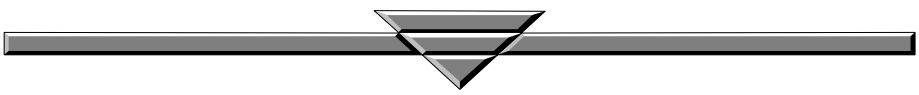
6.1 CONCEPTS FONDAMENTAUX 93
6.2 HIPAC 96
6.2.1 Le modèle de règles 96
6.2.2 Bilan 98
6.3 URDOS 99
6.3.1 Le triplet <E,C,A> 99
6.3.2 Le Schéma Actif 100
6.3.3 Bilan 101
6.4 D'AUTRES MODELES A BASE DE REGLES ECA 102
6.4.1 Le modèle Exact 102
6.4.2 Les travaux de Medeiros 102
6.4.3 Sentinel 103
6.4.4 Samos 104
6.5 LE SYSTEME NEOPUS 104
6.6 BILAN 105



CHAPITRE 7

LE MODELE ACTIF DE SHOOD

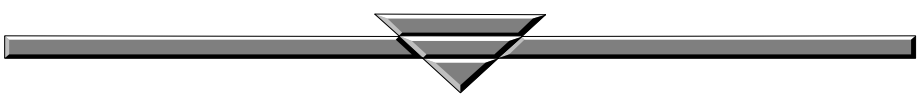
7.1 LES CONCEPTS DE BASE 107
7.1.1 L'événement 107
7.1.2 Les composants d'une règle ECA 109
7.1.2.1 L'expression événementielle 109
7.1.2.2 L'action et la condition 110
7.1.3 Deux familles de règles ECA 110
7.2 BASES DE REGLES POUR ORGANISER L'ACTIVITE 111
7.2.1 Non encapsulation des règles 111
7.2.2 Regroupement des règles 112
7.2.3 Héritage des règles 112
7.2.4 L'inhibition des règles 113
7.3 LA MODELISATION DU SCHEMA ACTIF 115
7.3.1 Modélisation 115
7.3.1.1 Les événements 115
7.3.1.2 Les règles 116
7.3.1.3 Les bases 117
7.3.2 Le noyau du système actif 118
7.4 LE MOTEUR D'EXECUTION 120
7.4.1 Déclenchement des événements 120
7.4.2 Sélection des règles 122
7.4.3 Exécution des règles 122
7.4.3.1 Les règles de vérification 122
7.4.3.2 Les règles de propagation 123
7.5 SYNTHESE 125



CHAPITRE 8

STRATEGIES ET REGLES D'EVOLUTION

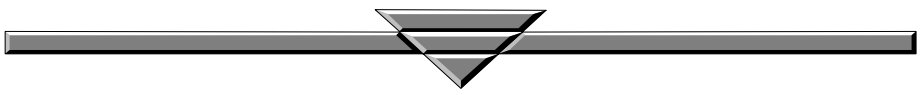
8.1 DEFINITIONS DES CONCEPTS 127
8.1.1 Définition d'une opération 127
8.1.2 Définition d'une pré-opération 128
8.1.3 Définition d'une post-opération 129
8.1.4 Définition d'une règle d'évolution 129
8.1.5 Définition d'une stratégie 130
8.2 REGLES D'EVOLUTION 131
8.3 STRATEGIES D'EVOLUTION 135
8.4 EXEMPLES 137
8.4.1 Exemple : suppression d'une classe 137
8.4.2 Exemple : représentation de contraintes 138
8.5 REALISATION 140
8.5.1 Règles d'évolution 141
8.5.2 Stratégies d'évolution 146
8.6 CONCLUSION 149



CHAPITRE 9

EXPERIMENTATION

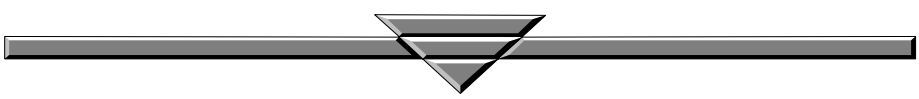
9.1 LE CADRE DE RECHERCHE 151
9.1.1 Le Dossier Unique Actualisé 152
9.2 LA DEMARCHE 152
9.3 LA MODELISATION 154
9.3.1 Description structurelle 154
9.3.2 Les variantes 158
9.4 LA MAQUETTE 162
9.5 CONCLUSION 163



CHAPITRE 10

CONCLUSION

BILAN 166
PERSPECTIVES 169



Les logiciels SHOOD et SHOOD/DUA sont déposés
à l'Association pour la Protection des Programmes (APP)

1. INTRODUCTION

Cette thèse s'inscrit dans le cadre des systèmes de représentation de connaissances à objets. Ces systèmes proposent un modèle permettant de représenter les connaissances du monde réel en utilisant un langage de classes et d'instances [Marino93] [Rechenmann88] [Ducournau88] [Bobrow77]. La description d'une application, dans ces systèmes à objets, passe par une phase d'acquisition des connaissances [Dieng90], de spécification de ces connaissances dans un modèle, et de validation. La phase de spécification se traduit par la construction du schéma de l'application, qui sera représenté par un ensemble de classes. La phase de validation, ou prototypage, passe par la création d'un ensemble d'instances, pour tenter de valider le schéma de l'application. Dans la plupart des situations, un retour vers la description du schéma est nécessaire pour modifier les classes : c'est la conception incrémentale. Une fois le schéma validé, l'utilisateur peut commencer à exploiter son application.

PROBLEMATIQUE

Les applications, en particulier celles dédiées à la CAO (Conception Assistée par Ordinateur), ont mis en évidence le caractère très évolutif des objets manipulés et des structures qui leur sont associées [Katz87]. Durant les phases de conception et d'exploitation de ces applications, le concepteur puis l'utilisateur peuvent être amenés à modifier soit le schéma de l'application, donc un ensemble de classes, soit les données de l'application, donc un ensemble d'instances.

Dans le cas où les classes sont modifiées, les classes qui en dépendent, par exemple leurs sous-classes, doivent rester valides. Cette validité est assurée en respectant des invariants, par exemple, en respectant l'héritage des propriétés [Banerjee87]. De plus, les instances doivent s'adapter aux modifications de leurs classes d'appartenance. Dans le cas où les instances sont modifiées, il faut s'assurer que les instances qui en dépendent restent cohérentes, en respectant des règles d'intégrité vis-à-vis de l'application et en étant conformes aux classes associées.

Afin de satisfaire ces besoins, les systèmes à objets, en particulier SHOOD [Nguyen92b] dans le cadre de cette thèse, devront autoriser l'évolution cohérente des classes, et celles des instances. La gestion de l'évolution dans un système à objets apparaît donc à deux niveaux :

- Au niveau des schémas : il s'agit de maintenir la cohérence de l'ensemble des classes en respectant les invariants du modèle de données. Un invariant est une condition que l'ensemble des schémas de classes doit vérifier. Ce respect des invariants est exprimé par des règles d'évolution de schémas, en l'occurrence en décrivant quand et comment évolue un schéma : par exemple "Si on supprime une classe, ses attributs doivent être supprimés". Dans la majorité des systèmes étudiés (ORION [Banerjee87], GEMSTONE [Penney87], ENCORE [Skarra86], OTGEN [Lerner90] etc.), les règles d'évolution ne sont pas exprimées de manière déclarative, sauf dans certains systèmes comme ADELE3 [Ahmed-Nacer94], où elles sont exprimées par des règles actives. En général, cette évolution est décrite plutôt par des opérations câblées, noyées dans le code de l'application ou du système, offrant une faible extensibilité au prix d'un lourd effort de programmation.

- Au niveau des objets de la base, il s'agit de maintenir la cohérence des données. Une base d'objets est dite cohérente si les objets de cette base respectent des contraintes. Ces contraintes peuvent être liées au modèle (les objets doivent être conformes à leur structure) ; dans ce cas, elles sont gérées par le système. Elles peuvent également être liées à l'application ; dans ce cas, elles sont généralement réalisées par des mécanismes d'expression et de mise en œuvre de contraintes d'intégrité et/ou de calculs inférentiels. Par exemple, la contrainte : "Si le rayon de la roue avant est modifié, alors vérifier qu'il est égal à celui de la roue arrière" est une contrainte liée à l'application. Par contre, la contrainte "toute valeur d'attribut doit être conforme au type de l'attribut défini" est une contrainte liée au modèle de données.

En plus du peu d'extensibilité des mécanismes d'évolution actuels [Banerjee87] [Penney87], ces derniers présentent un autre inconvénient : celui d'offrir des stratégies d'évolution souvent uniques et imposées à tous les types de classes. Cet inconvénient empêche le programmeur d'exprimer plusieurs stratégies et d'activer la plus adéquate pour l'application. En effet, l'utilisateur ne peut pas, par exemple, exprimer deux stratégies de suppression de classes différentes, chacune adaptée à une application spécifique.

Exemple : Dans SHOOD, la suppression d'une classe entraîne le rattachement de ses instances aux super-classes. Cette évolution est une sémantique par défaut. Le programmeur peut vouloir exprimer d'autres stratégies pour certaines applications, par exemple, que la suppression d'une classe entraîne celle de ses instances.

CONTRIBUTION

Afin de satisfaire les besoins d'extensibilité et d'expression de stratégies d'évolution multiples, nous proposons, dans le cadre de cette thèse, un système d'évolution supportant un ensemble de mécanismes : classification d'instances, gestion des règles actives, des stratégies et des règles d'évolution (Figure 1-1).

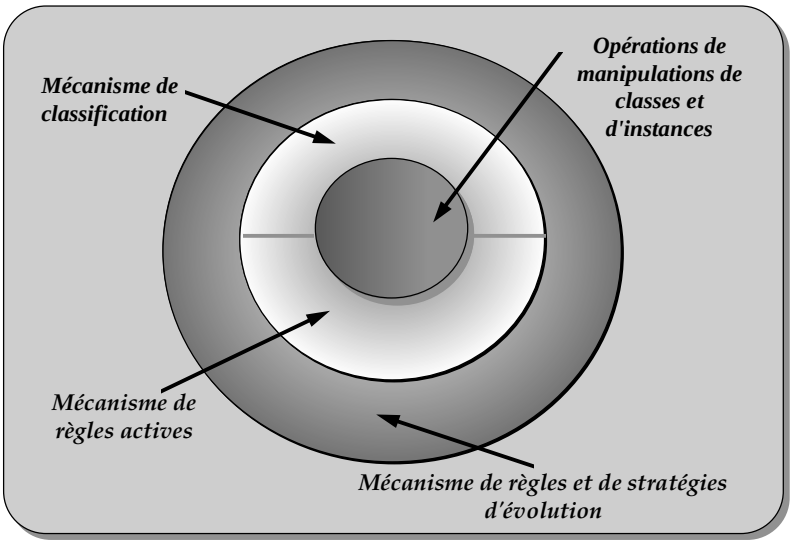


Figure 1-1 : Composition du système d'évolution

L'évolution des classes et des instances est mise en œuvre par un ensemble d'opérations de manipulation de classes et d'instances. Ces opérations respectent un ensemble d'invariants du modèle de connaissances SHOOD. Au-dessus de ces opérations, nous avons construit un mécanisme de classification d'objets évolutifs (Figure 1-1) qui utilise les opérations d'évolution des instances. La classification d'instances a pour but de trouver la représentation la plus adéquate pour un objet incomplet et temporairement incohérent. Le mécanisme de classification et les opérations de manipulation ne sont malheureusement, ni extensibles, ni modifiables facilement. D'où la nécessité d'introduire une nouvelle dimension au niveau du système d'évolution.

Nous proposons pour cela, d'utiliser le concept de règle qui va permettre une expression déclarative de l'évolution des schémas et des instances : Un mécanisme à base de règles actives permet d'exprimer de manière déclarative les vérifications et les propagations nécessaires lors de l'évolution des classes et des instances. Une règle active est définie en général par trois composants : un événement, une condition et une action [Dayal88][Diaz95]. Elle est aussi appelée règle ECA (Evénement, Condition, Action). Ces règles actives sont utilisées pour rendre un système actif (ou réactif), donc capable de détecter des situations et de réagir sans intervention de l'utilisateur, en exécutant des actions ou des programmes : la détection d'un événement entraînant l'exécution d'une action lorsqu'une condition est satisfaite. Cette expression déclarative de l'évolution permettra une maintenance facile des règles d'évolution, fonctionnalité souvent absente dans un mécanisme non extensible.

Par ce modèle de règles, nous étendons l'utilisation habituelle des règles actives à l'évolution des schémas. En effet, leur utilisation pour assurer le contrôle de l'intégrité a déjà fait l'objet de travaux antérieurs [Bouaziz95] [Medeiros91]. Mais, nous pensons que l'expression de l'évolution par les règles actives nécessite un langage de spécification plus puissant et plus

concis. En effet, il faut souvent écrire plusieurs règles ECA pour exprimer une seule règle d'évolution. De plus, le modèle de règles actives proposé ne permet pas la définition des stratégies d'évolution.

Pour pallier ces inconvénients, nous proposons un mécanisme permettant d'exprimer des règles d'évolution et de définir et gérer des stratégies d'évolution. Les règles d'évolution proposées ici permettent d'exprimer les vérifications et les propagations de mise à jour à exécuter avant et après une opération faisant évoluer un schéma ou une instance. Une règle d'évolution désigne alors une opération précédée d'un traitement appelé pré-opération et suivie d'un traitement appelé post-opération, chacun assurant la vérification puis la propagation des mises à jour. Une pré et une post opération sont composées d'une condition et d'une action assurant respectivement la vérification et la propagation dues à l'exécution de l'opération.

Pour assurer l'expression et la gestion de plusieurs sémantiques d'évolution différentes, nous proposons la notion de stratégie, définie par un ensemble de règles d'évolution. Les stratégies sont organisées dans un graphe de spécialisation avec redéfinition et héritage des règles d'évolution. Le mécanisme de règles et de stratégies est implémenté sur le système SHOOD. Cette implémentation est réalisée, entre autre, en établissant une correspondance entre une règle d'évolution de schémas ou d'instances et plusieurs règles actives.

PLAN DE LA THESE

Cette thèse est organisée en sept chapitres : - les chapitres 2 et 6 présentent un état de l'art dans les domaines de l'évolution et des modèles à base de règles ; - les chapitres 4, 5, 7 et 8 décrivent l'ensemble des mécanismes permettant la gestion de l'évolution dans le modèle SHOOD qui est décrit au chapitre 3 ; - et une expérimentation est présentée dans le chapitre 9.

Le **chapitre 2** est consacré à un état de l'art de l'évolution des schémas et des instances. Nous présentons principalement les différentes approches pour gérer l'évolution dans un système à objets. Dans ce chapitre, nous insistons plus sur l'évolution de schémas que sur l'évolution des instances, car notre travail est plus axé sur l'évolution des classes. En effet, comme le modèle SHOOD est **tout objet**, la gestion de l'évolution des instances était plus un pré-requis à celle des classes (une classe étant une instance d'une métaclasse) qu'une étude distincte.

Dans **le chapitre 3,** nous décrivons brièvement le modèle de données de SHOOD. Les différents concepts y sont présentés avec les invariants correspondants. Les invariants doivent être respectés par toute opération ou mécanisme d'évolution.

Une description générale du système d'évolution est faite dans le **chapitre 4** ; nous présentons ses propriétés et son architecture. Le chapitre concerne principalement la présentation du

noyau du système d'évolution, appelé ici support d'évolution. Le reste du document est consacré à la présentation des autres mécanismes de ce système.

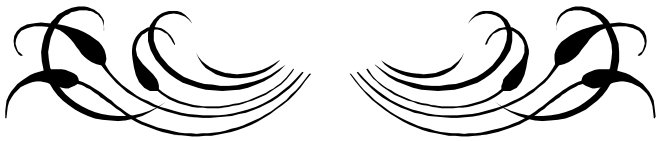
Le **chapitre 5** présente le mécanisme de classification d'instances incomplètes et incohérentes. Ce mécanisme devra principalement respecter les invariants d'instanciation et de multi-instanciation.

Dans le **chapitre 6,** nous présentons quelques modèles à base de règles. Ce chapitre permettra de situer nos travaux sur les modèles actifs.

Le **chapitre 7** décrit la modélisation et la mise en œuvre des règles actives. Dans ce chapitre, nous illustrons les différents concepts par des exemples, pour montrer que les règles actives se prêtent bien à l'expression de l'évolution.

Dans le **chapitre 8**, nous présentons le mécanisme de règles et de stratégies d'évolution. Nous montrons dans ce chapitre que les règles d'évolution offrent une expression mieux structurée que les règles actives, et que les stratégies d'évolution permettent l'expression et la gestion de différentes sémantiques d'évolution.

Nous achevons cette thèse par le **chapitre 9,** où nous présentons une expérimentation dans le cadre de la conception d'une base de connaissances dans le domaine de la construction et du bâtiment. Cette expérimentation présente une utilisation de la dynamique de SHOOD. Le système d'évolution a permis la réalisation d'une maquette mettant en œuvre les règles de contrôle de l'intégrité de l'application. Cette expérimentation utilise principalement les opérations de manipulation de SHOOD, ainsi que les règles actives.



2. UN ETAT DE L'ART SUR L'EVOLUTION

L'objectif de ce chapitre est de présenter les différentes approches possibles pour assurer la dynamique d'un système à objets. La dynamique des objets est l'ensemble des concepts qui permettent de faire évoluer un objet d'un état vers un autre tout en conservant l'état de l'ensemble des objets cohérent. Un objet (un schéma ou une instance) est dit cohérent s'il vérifie certaines règles bien précises. Ces règles traduisent les conditions de validité des informations manipulées.

Les solutions proposées dans la littérature diffèrent essentiellement par les fonctionnalités nécessaires aux applications ciblées par les systèmes et par la sémantique attachée à l'évolutivité [Zhou94] [Waller91] [Tresh91] [Schek90] [Roddick92]. Par exemple, dans les systèmes de représentation de connaissances, un mécanisme de classification est nécessaire pour faire évoluer l'état d'un objet en fonction des connaissances contenues dans cet objet [Bounaas94b]. Dans d'autres systèmes, comme ceux dédiés au Génie Logiciel, un mécanisme de versions est forcément présent [Adele93].

Le travail effectué dans cette thèse est plus axé sur l'évolution de schémas, mais comme le modèle SHOOD est tout objet, assurer l'évolution des classes a eu pour conséquence la construction d'un niveau permettant l'évolution des instances. Nous verrons dans les chapitres 7 et 8 que l'évolution des instances s'exprime de la même manière que l'évolution des classes. Pour cela, dans ce chapitre, nous parlons principalement de l'évolution de schémas et plus brièvement de l'évolution des instances.

2.1 COMMENT SUPPORTER L'EVOLUTION

L'évolution apparaît au niveau des données manipulées par les applications (évolution des instances) et au niveau des structures nécessaires ces applications (évolution des schémas).

L'évolution des schémas est généralement mise en œuvre par une taxonomie d'opérations respectant un ensemble d'invariants. Un invariant est une condition que l'ensemble des classes et des instances doit respecter à tout moment [Escamilla93]. Cette évolution est, de plus, souvent gérée par un mécanisme de versions permettant de suivre dans le temps l'évolution des structures [Kim88].

L'évolution des instances est mise en œuvre couramment par des contraintes d'intégrité permettant d'exprimer la cohérence des données [Bouaziz95].

Nous verrons dans ce chapitre qu'il existe plusieurs techniques pour gérer l'évolution telles que la classification d'instances [Marino91] [Liotard93] ou de classes [Capponi94], la réorganisation de la hiérarchie de classes etc. Une nouvelle tendance est apparue à savoir la gestion de la dynamique par les objets actifs : le principe est d'associer des règles actives aux objets. Ces règles modélisent le comportement des objets lors des différents cas d'évolution [Bouaziz95], [Collet94].

Dans la suite, nous citons les différentes approches pour supporter l'évolution de schémas et l'évolution des instances. Nous présentons uniquement une synthèse des principaux travaux dans ce domaine afin de mettre en lumière leurs forces et leurs faiblesses.

2.2 EVOLUTION DE SCHEMAS

2.2.1 Gestion de l'évolution de schémas

Il existe actuellement trois grandes approches pour gérer l'évolution de schémas [Benatallah94].

- La réorganisation de classes : elle se base sur des approches algorithmiques [Dicky94] [Casais91] ou sur des mécanismes de classification de classes [Capponi94].
- La correction : cette approche consiste à modifier la classe sans garder de trace de la classe avant modification ; elle est la plus courante et la plus classique.
- Le versionnement : une nouvelle version de la classe est créée pour contenir les changements. Cette approche est la plus utilisée dans les grandes bases de données [Kim88], particulièrement celles dédiées au génie logiciel [Ahmed-Nacer94].

Dans la suite, nous nous attardons plus sur la description de l'approche par correction, qui est celle qui est choisie pour notre système. En effet, elle s'adapte mieux aux besoins des systèmes de représentation de connaissances où l'objectif premier est de modéliser les connaissances et non pas de gérer un grand nombre d'informations comme dans les bases de données.

2.2.1.1 Correction

L'approche par correction consiste à modifier directement la définition d'une classe ; on ne garde aucune trace de l'état des classes avant modification. Cette approche peut être résumée par les phases suivantes :

♦ Définition d'une taxonomie des opérations de manipulations possibles.

- Définition des invariants d'un schéma qui devront être préservés lors de toute évolution de schéma.
- Définition d'une stratégie de propagation des modifications aux instances.

Nous présentons dans la suite la gestion de l'évolution dans ORION qui peut être considéré comme une référence pour cette approche, puis nous décrivons brièvement d'autres systèmes.

2.2.1.1.1 Orion

ORION [Banerjee87] est implémenté en CommonLisp et supporte les caractéristiques usuelles d'un modèle à objets, y compris les objets composites [Kim89], l'héritage multiple. Les concepteurs d'ORION identifient une taxonomie d'opérations respectant un ensemble d'invariants. ORION permet aussi la gestion des versions de schémas (et non de classes) et d'instances [Kim88].

Taxonomie des opérations possibles :

1. Changement d'une définition de classe
   - 1.1. Changement sur une variable d'instance
     - 1.1.1. Ajouter une variable d'instance
     - 1.1.2. Supprimer une variable d'instance
     - 1.1.3. Modifier (nom, domaine, valeur par défaut, etc.).
   - 1.2. Changement sur une méthode
     - 1.2.1. Ajouter une méthode
     - 1.2.2. Supprimer une méthode
     - 1.2.3. Modifier (nom, code)
2. Changement de lien is-a
   - 2.1. Ajouter une super-classe
   - 2.2. Retirer une super-classe
   - 2.3. Changer l'ordre des super-classes
3. Changement sur une classe
   - 3.1. Créer une classe
   - 3.2. Supprimer une classe
   - 3.3. Modifier le nom d'une classe

L'ensemble de ces opérations devra respecter les invariants suivants :

Invariants d'évolution de schéma

Invariant du graphe de classes : Le graphe de classes est un graphe connexe acyclique qui possède un ensemble de sommets (les classes) et de liens nommés (is-a). La racine du graphe est appelé Objet.

Invariant de nom : Toutes les classes, les variables d'instances et les méthodes d'une classe propres ou héritées ont des noms distincts.

Invariant d'héritage : Une classe hérite des variables d'instances et des méthodes des super-classes à condition de respecter les invariants d'identité (les variables et méthodes ont une origine unique) et l'invariant de domaine (le domaine d'une variable dans V2, héritée de V1, doit être le même ou sous-classe du domaine de la variable dans V1).

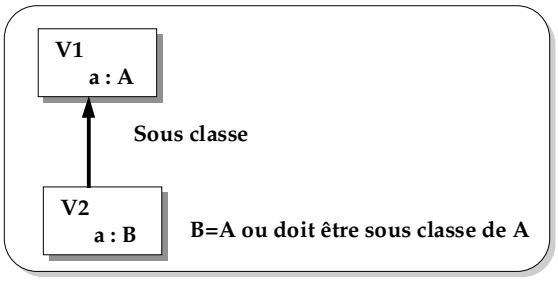


Figure 2-1 : Redéfinition d'un domaine

Règles d'évolution de schémas

La hiérarchie de classes doit toujours respecter les invariants. Cependant, il existe plus d'une manière de préserver ces invariants. Par exemple, lors d'un conflit de noms parmi des variables d'instances héritées des super-classes, l'invariant exige de garder une seule variable mais n'indique pas laquelle. Une règle d'évolution de schémas fixera la variable à conserver.

Afin de guider la sélection d'une option parmi plusieurs pour préserver les invariants, les concepteurs d'ORION ont défini un ensemble de règles partagées en quatre catégories. Certaines de ces règles sont communes à d'autres systèmes.

Règles de résolution de conflits :

- Si au moins deux super-classes d'une classe C possèdent une variable d'instance (méthode) de même nom, mais d'origine[1] distincte, la variable (méthode) héritée est celle de la première super-classe (le numéro d'arc entrant le plus petit). Dans la Figure 2-2, si l'ordre des superclasses est (V2, V3, V4), alors la classe V5 hérite de la méthode *m* de V3.

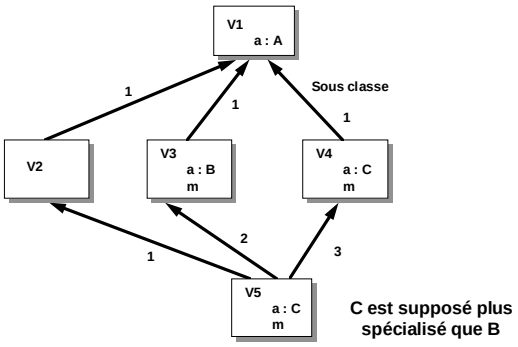


Figure 2-2 : Résolution de conflits

[1] Origine : classe représentant la première définition d'une variable ou d'une méthode.

- Mais s'il y a conflit de mêmes noms, et de même origine, on sélectionne la variable la plus spécialisée (domaine plus restreint). Si les domaines sont identiques, on hérite de la première super-classe. Dans la Figure 2-2, la classe V5 hérite de la variable *a* de V4 si le domaine C est plus spécialisé que B (domaine de *a* dans V3).

Règles de propagation

- Si une propriété est modifiée, cette modification est propagée aux sous-classes, jusqu'à redéfinition dans une sous-classe. Dans la Figure 2-2, si on modifie *a* dans V1, cette modification n'est propagée que dans V2, seule classe qui ne redéfinie pas la variable *a*.

- L'ajout d'une propriété est propagé aux sous-classes jusqu'à conflit de noms.

- On ne peut pas généraliser le domaine d'une variable d'instance. Par contre, le domaine d'une variable héritée ne peut qu' être restreint.

Règles de manipulation de graphes

Cet ensemble de règles permet de gérer les ajouts et les suppressions de nœuds et d'arcs d'un graphe de classes :

- Elimination d'un arc. Si A est supprimée de la liste des super-classes de B et si A se trouve être l'unique super-classe de B, alors B est rattachée aux super-classes de A, avec comme ordre des super-classes le même que celui de A.

- Ajout d'un arc. Si A devient une super-classe de B, elle aura le numéro d'arc suivant, ce qui fait qu'aucun nouveau conflit ne peut apparaître.

Règles de gestion des objets composites

ORION définit un ensemble de règles pour changer une variable composite en une variable non composite et vice-versa. Ces règles permettent aussi de changer les dépendances d'un lien de composition de dépendant à non dépendant, et du partagé vers l'exclusif, tout en respectant certaines conditions [Kim89].

ORION offre un modèle de données des plus complexe ; on retrouve plusieurs concepts (composition, dépendance, versions, etc.), ce qui induit un ensemble d'invariants et de règles assez complet. Tout ceci fait d'ORION un modèle de référence en matière d'invariants.

Nous verrons dans les paragraphes suivants la gestion des versions de schémas dans ORION (cf. 2.2.1.2) ainsi que l'adaptation des instances à l'évolution des classes (cf. 2.2.2).

2.2.1.1.2 GemStone

La cohérence structurelle d'un schéma est assurée comme dans ORION par la préservation d'invariants.

La modification de schéma dans GEMSTONE [Penney87] est similaire à celle d'ORION. En effet, les concepteurs de GEMSTONE ont défini des invariants de graphe de classes et d'héritage similaires mais plus simple que ceux d'ORION, car l'héritage est simple (le problème de conflits n'existant pas). Nous retrouvons d'autres invariants supplémentaires comme :

Invariant de représentation : cet invariant est implicite dans ORION ; il exprime qu'une instance doit être conforme à la définition de la classe (contraintes, attributs, etc.).

Invariant de la compatibilité des contraintes : identique à l'invariant des domaines d'ORION.

Invariant de la cohérence des références : GEMSTONE garantit l'inexistence de référence pendante (non résolue) : un objet référence toujours un objet existant dans la base. Si un objet n'est plus référencé, il est détruit. La simplicité du modèle de données simplifie le problème dans GEMSTONE qui n'a plus besoin de règles comme ORION pour choisir le meilleur moyen de maintenir un invariant.

2.2.1.1.3 D'autres systèmes utilisant la correction

OTGEN est un prototype développé à l'université de Carnegie Mellon (USA) offrant d'une part une dynamique de schémas par un ensemble d'opérations respectant des invariants et d'autre part une approche par versionnement de schémas [Lerner90]. L'ensemble des invariants est un sous-ensemble de ceux d'ORION (sans l'invariant d'identité) auquel est ajouté l'invariant de cohérence des références défini dans GEMSTONE [Penney87]. OTGEN définit un invariant (invariant du domaine d'attribut) qui assure que chaque domaine d'attribut correspond à une classe du graphe. Cet invariant est souvent implicite dans d'autres systèmes.

O2 [Bancilhon89] [Zicari91] [Delcourt92] est un système réalisé pour supporter le développement d'applications dans de larges domaines. L'évolution de schémas (dans le prototype recherche) est assuré par un ensemble d'opérations et d'invariants plus restreint que celui d'ORION, cette restriction étant due à la simplicité du modèle. Par exemple, on ne retrouve pas d'opération sur les classes indexables (voir Tableau 2-1).

2.2.1.1.4 Conclusion sur la correction

L'approche par correction est une voie appréciable pour l'évolution de schémas, afin d'assurer la consistance des schémas de classes. Si cette approche doit être adoptée, il faut assurer la cohérence et la complétude des opérations de manipulation. En effet, le maximum

d'opérations fiables doit être proposé. Certains systèmes ne proposent pas, par exemple, d'opérations pour le déplacement d'une classe dans un graphe de classes.

Nous présentons un tableau récapitulatif des taxonomies et invariants possibles pour les systèmes décrits ci-dessus ; Il existe d'autres travaux adoptant cette approche [Scherrer93] [Barbedette91].

Opérations	**Orion**	**GemStone**	**OTGen**	**O2**
Attributs ou Variables d'instance				
ajout	✔	✔	✔	✔
suppression	✔	✔	✔	✔
renommage	✔	✔	✔	✔
redéfinition type	✔	✔	✔	✔
modification de l'origine		✔		
modification de la valeur par défaut		✔		
Méthodes				
ajout	✔			✔
suppression	✔			✔
renommage	✔			✔
redéfinition de la signature				✔
modification de code	✔			✔
modification de l'origine	✔			
Valeurs partagées	✔			
Lien de composition	✔			
Classes				
ajout	✔	✔	✔	✔
suppression	✔	✔	✔	✔
renommage	✔		✔	✔
Classes indexables	✔			
Liens d'héritage				
ajout d'une super-classe	✔		✔	✔
retrait d'une super-classe	✔		✔	✔
modification de la précédence	✔			

Tableau 2-1 : Comparaison des opérations

Invariants	**Orion**	**GemStone**	**OTGen**	**O2**
Représentation		✔		
Graphe de classes	✔	✔	✔	✔
Noms distincts	✔		✔	✔
Héritage	✔	✔	✔	✔
Origine distincte	✔			✔
Compatibilité de domaine	✔	✔	✔	✔
Domaine d'attribut			✔	
Cohérence des références		✔	✔	

Tableau 2-2 : Comparaison des invariants

Certains de ces invariants comme l'invariant de représentation et le domaine d'attribut sont souvent implicites dans la plupart des modèles ; ils ne sont pas indiqués mais ils sont maintenus.

L'approche par correction possède l'inconvénient de ne pas garder une trace de l'évolution d'un schéma. Par moments, il est nécessaire de suivre dans le temps les différentes modifications d'un schéma de classes ; cette trace peut servir à partager plusieurs versions de schémas de classes par plusieurs utilisateurs. La gestion de versions est utile dans le cadre des applications CAO [Katz88].

2.2.1.2 Versionnement

Le versionnement se distingue de l'approche par correction par le fait qu'après la modification d'une classe, on conserve l'état du schéma ou de la classe avant la modification. Il en résulte le concept de version qui est un état d'une entité (objet, classe, schéma) que le système ou l'utilisateur veut conserver. Une des exigences d'un environnement de conception multi-utilisateurs est la possibilité donnée aux utilisateurs de manipuler plusieurs ensembles d'objets sous différentes versions de schémas de classes. Parmi plusieurs approches destinées à satisfaire cette exigence (versions de schémas, versions de classes, vues de schéma), ORION et OTGEN proposent l'approche de versions de schémas où tout un graphe de classes est vu comme une version ; tandis que ENCORE, GUIDE [Krakowiak87], ADELE [Ahmed-Nacer94], etc. développent un modèle de versions de classes.

2.2.1.2.1 Versions de schémas

Orion

ORION permet la gestion des traces de conception d'un objet, grâce à la notion de versions d'instances. Les versions permettent, en plus, de gérer l'évolution des objets en gardant une trace de leurs différents états. ORION permet la gestion des versions d'objets (instances terminales), ainsi que les versions de schémas, un schéma étant un graphe de classes.

Lorsque la définition d'une classe évolue, l'usager ne crée pas une version de classe mais une version du schéma contenant la classe modifiée. A un instant donné, l'usager travaille sur une version de schéma courante. Il peut accéder à l'ensemble des objets visibles dans cette version ("access scope"), qui englobe l'ensemble des objets créés sous cette version ("direct access scope") et ceux hérités des versions ancêtres.

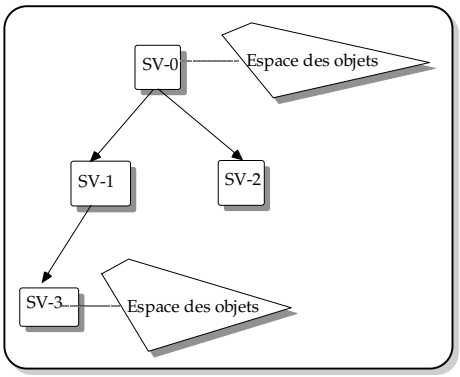


Figure 2-3 : Hiérarchie de versions de schémas

Dans la figure ci-dessus, SV-3 est un descendant de SV-0 et une fille de SV-1 et inversement SV-0 est un ancêtre de SV-3 et un parent de SV-1 et SV-2.

Comme une copie de schéma requiert un espace de stockage important, l'objectif d'ORION est de ne pas maintenir une copie de chaque version de schéma, mais de noter tous les changements de schémas. Ces changements peuvent être soit des changements de définitions de classe (une copie de la classe est alors créée), soit un changement de lien entre les classes (une copie des classes concernées est créée).

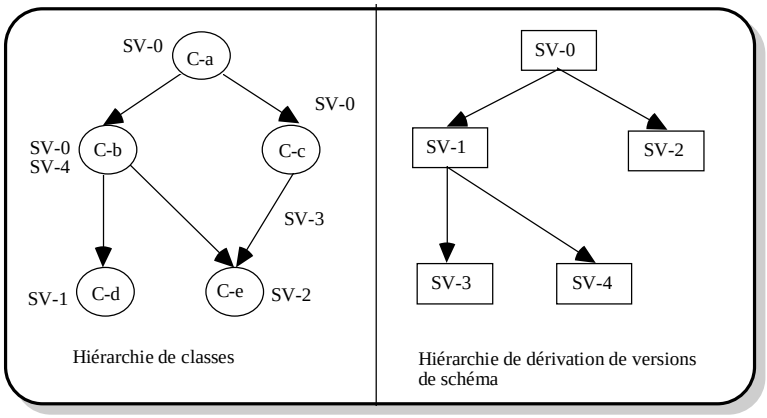


Figure 2-4 : Une hiérarchie de versions de schémas

La figure ci-dessus illustre la représentation des schémas de versions où la hiérarchie de classes est construite et modifiée en générant une hiérarchie de cinq versions de schémas. Dans la hiérarchie de classes, on marque chaque classe ou lien interclasses par les versions de schémas sous lequel ils ont été créés, manipulés ou détruits. Dans la figure ci-dessus, C-a, C-b, C-c sont crées en sous la version de schéma SV-0. Cette hiérarchie est modifiée en rajoutant des classes (C-d sous SV-1 et C-e sous SV-2) et des liens (Lien entre C-c et C-e sous SV-3).

Une instance d'ancrage matérialise l'historique des modifications d'une classe. Elle comporte toutes les copies de classe référencées par la version de schéma correspondante. Chaînées entre elles, les copies de classes représentent l'évolution de chaque classe (voir Figure 2-5).

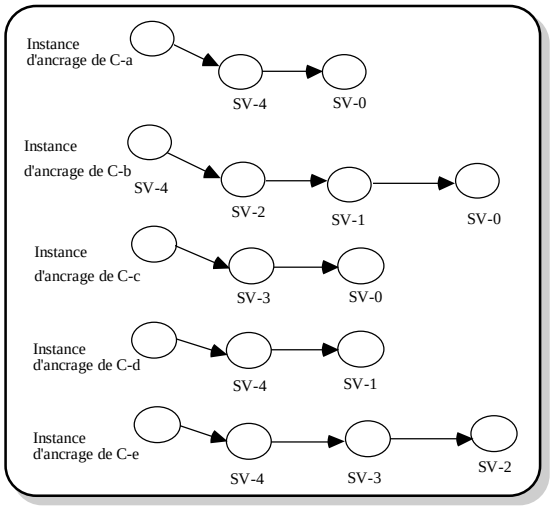


Figure 2-5 : Instances d'ancrage de classes

La Figure 2-5 illustre cette notion. En premier, sous la version de schéma SV-0, on définit les classes C-a, C-b et C-c. En second, la classe C-d est créée comme sous-classe de C-b, ce qui entraîne la création d'une copie de C-b sous SV-1. Troisièmement, sous le schéma SV-2, on crée la classe C-e comme sous-classe de C-b, d'où une copie de C-b sous SV-2. En quatrième lieu, C-c devient une super-classe de C-e, une copie de C-c et C-e est générée. Et en dernier lieu, la classe C-b est détruite sous SV-4. C-a devient une super-classe directe de C-d et C-e, ce qui entraîne une copie de C-a, C-d et C-e.

La solution proposée par ORION concernant les versions de schémas parait assez complexe, car toute modification entraîne la création d'une version du schéma global, d'où prolifération de versions de schémas. De plus, il n'est pas possible de créer une version d'une partie du schéma global.

OTGen

OTGEN [Lerner90] adopte la même politique qu'ORION : une modification de schéma entraîne une version de tout le schéma de classes plutôt qu'une version des classes modifiées. L'élaboration de ces versions est facilitée par l'utilisation d'un outil interactif. L'administrateur de la base utilise une notation déclarative pour exprimer les correspondances entre les objets de l'ancienne version et ceux de la nouvelle version de schéma. OTGEN génère un transformateur qui applique les correspondances pour mettre à jour la base avec les nouvelles définitions. Ces transformateurs constituent les entrées des tables de transformation. Une entrée d'une table peut être :

```
Class Personne
    new self : Personne
    nom : Transforme nom
    age : 0
```

Soit la classe Personne possédant l'attribut nom dans l'ancien schéma. Si dans la nouvelle version de schéma elle possède les attributs nom et âge, le transformateur peut être interprété de cette façon :

```
Pour chaque instance de l'ancienne définition de Personne :
        créer une nouvelle instance de la classe personne
        recopier l'ancienne valeur de l'attribut nom
        valuer l'attribut âge à 0.
```

Les tables de transformation permettent de transférer les anciennes données vers un nouveau schéma. Le système fournit un outil pratique et puissant pour permettre l'évolution de la base de données, sauf dans le cas extrême où on doit convertir toute la base vers une ancienne version, ce qui risque d'être coûteux pour de grandes bases de données. Lors d'une modification, ce système requiert la conversion de toute la base de données, ce qui est coûteux et inutile si la modification de schéma est localisée.

2.2.1.2.2 Versions de classes

Dans le domaine du génie logiciel, le problème de versions de classes est largement traité. Nous présentons rapidement les travaux de GUIDE et de ENCORE.

Encore

ENCORE [Skarra86] [Skarra87] est le prototype d'un SGBDOO traitant les problèmes des versions de classes. Les concepteurs de ENCORE proposent un mécanisme pour la gestion d'un ensemble de versions de classes. Un ensemble de versions de classes est une collection, ordonnée par ordre de création, initialisée avec la première définition de la classe et étendue par une nouvelle version à chaque modification de classe. De cette façon, toute instance reste cohérente par rapport à sa classe. La version courante est toujours la version la plus récemment créée. De plus, la création d'une version de classe entraîne la création d'une version de chaque sous-classe.

Une interface de l'ensemble des versions d'une classe est associée à chaque ensemble de versions de cette classe. Cette interface contient l'union de tous les attributs des versions de cette classe. C'est l'interface de l'ensemble des versions qui est visible pour l'utilisateur, c'est une vue de l'ensemble des versions où aucun attribut n'est supprimé.

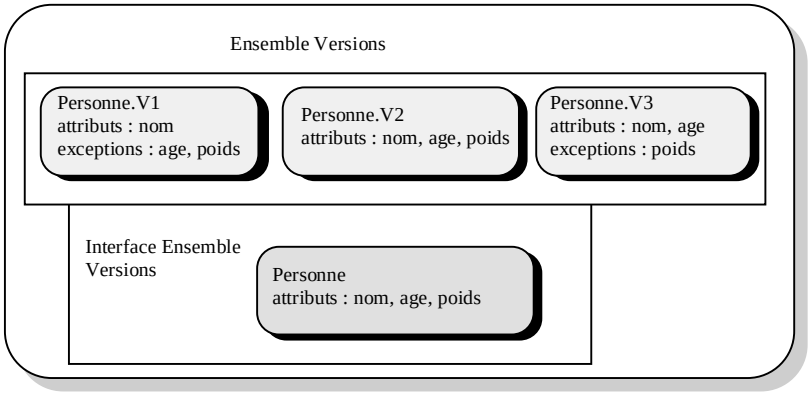


Figure 2-6 : L'interface de l'ensemble des versions dans Encore

Tout programme accédant à l'interface de l'ensemble des versions trouve toujours une référence de l'attribut même si celui ci est supprimé, sauf s'il tente d'accéder à une instance d'une version où l'attribut n'existe pas. Pour cela, ENCORE définit deux exceptions (lecture et écriture) pour tout attribut existant dans l'interface mais non dans la version ; ces exceptions peuvent rendre des valeurs. L'exception en écriture rend la valeur Vrai ou Faux selon l'existence ou l'absence de l'attribut, qu'on veut modifier, dans l'instance d'une version de classe. Les exceptions en lecture sont souvent utilisées pour rendre des valeurs par défaut.

Dans la Figure 2-6, une exception pourrait être définie pour l'âge pour rendre la valeur 0 lorsqu'on accède à l'attribut pour les instances de la classe *Personne.V1*. Si l'action de l'exception en lecture rend la valeur 0, alors la seule écriture autorisée pour l'attribut âge est l'écriture de la valeur 0.

Dans ENCORE, les attributs de même nom mais référencés dans des versions différentes représentent des informations identiques, ce qui interdit de modifier la sémantique d'un attribut d'une version vers une autre. ENCORE ne permet pas de créer des versions de schémas.

Guide

GUIDE est système d'exploitation réparti sur un réseau local. Les informations qu'il manipule sont structurées selon un modèle à objets persistants. Les applications mises en œuvre sur GUIDE sont écrites dans un langage à objets [Krakowiak87].

Le modèle d'objets de GUIDE est fondé sur la séparation des types et des classes. Un type est la description d'un comportement commun aux objets de ce type. Il comprend les signatures des méthodes. Une classe définit un schéma particulier pour la réalisation d'un type, plusieurs classes pouvant être associées à un type. L'héritage est simple.

La gestion de l'évolution dans GUIDE est statique. Les modifications des types et classes ne sont validées qu'en recompilant les types et les classes concernés par la modification, afin de

détecter les éventuelles inconsistances dues à la modification (exemple : référence d'un type inexistant).

Une étude a été réalisée pour construire une trace de l'évolution des types et classes [Bounaas91]. Cette trace est mise en œuvre par un gestionnaire de versions des types et classes. Chaque modification de classe (respectivement type) peut (à la demande de l'utilisateur) être sauvegardée dans une version. Afin de maintenir la cohérence des classes par rapport à leurs sous-classes, la création d'une version d'une classe (respectivement type) entraîne la création d'une version des sous-classes. Une entité abstraite (similaire à une instance générique d'ORION) matérialise l'historique des versions qui est associé à chaque version d'une classe ou type.

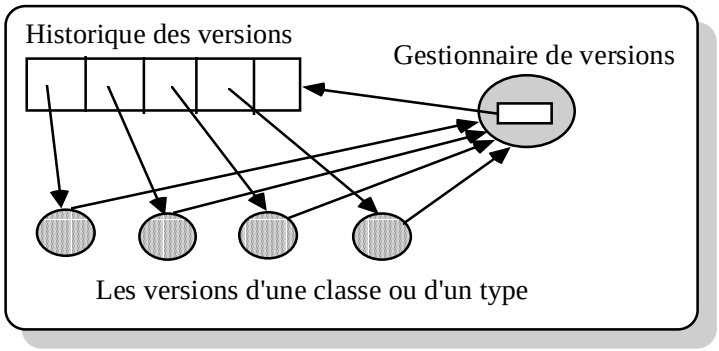


Figure 2-7 : Le gestionnaire de versions de GUIDE

Chaque version de classe référence une instance du gestionnaire de versions, ce dernier référence l'historique des versions. L'accès à une version quelconque est obtenu en appelant une méthode du gestionnaire de versions correspondant.

GUIDE, comme ENCORE, n'offre pas la possibilité de gérer des versions de schémas, qui par moments pourraient être nécessaire. De plus, l'inconvénient majeur, rencontré lors de la mise en œuvre des opérations de modification, est le manque d'extensibilité du langage et l'expression de l'évolution ; les invariants sont exprimés au niveau du compilateur du langage.

2.2.1.2.3 Conclusion sur le versionnement

Le versionnement est une approche complémentaire à l'approche par correction et indispensable dans un environnement multi-utilisateurs ; chaque utilisateur pouvant disposer d'une version de schémas d'une application. Il existe deux grandes approches pour assurer le versionnement : - soit le versionnement de classes, qui est l'approche la plus répandue [Clamen92] [Adele93] [Björnerstedt89] ; elle consiste à générer une version de classe pour toute classe modifiée, - soit le versionnement de schémas où lorsqu'une classe est modifiée tout le schéma est vu comme une nouvelle version. L'approche par version de schémas offre l'avantage d'une vue globale du schéma, mais rend le système complexe, du fait du nombre important de versions de schémas, et peut entraîner des confusions chez l'usager. Il peut, par

exemple, trouver des difficultés à déterminer la version de schéma correspondant à une modification d'une classe donnée.

Le versionnement constitue une approche nécessaire dans un système dédié pour les applications CAO [Katz87] comme SHOOD. Cette approche ne sera pas traitée dans cette thèse car nous avons estimé plus prioritaire de résoudre les problèmes d'extensibilité et d'expression des mécanismes d'évolution, qui, de plus, représentent un thème très peu étudié dans la littérature.

2.2.1.3 Réorganisation de la hiérarchie des classes

Les outils qui restructurent automatiquement la hiérarchie de classes peuvent réduire les efforts de conception [Casais91] [Dicky94]. Les méthodes disponibles vont des méthodes informelles pour détecter et corriger les imperfections, comme dans SMALLTALK [Goldberg83] ou dans EIFFEL [Meyer88], jusqu'aux algorithmes qui ajustent la définition en se basant sur des critères structurels (la loi de Demeter [Lieberherr89]) et les travaux de E. Casais [Casais91]. Ces méthodes sont invoquées lors de l'ajout ou de la suppression de classes. Nous retrouvons dans le domaine des représentations de connaissances des mécanismes similaires comme la classification de classes [Napoli92] [MacGregor91].

Dans la suite, nous verrons une approche algorithmique décrite dans la thèse de Casais et un mécanisme de classification de classes développé sur SHIRKA [Capponi94].

2.2.1.3.1 Une approche algorithmique

Casais propose une approche algorithmique pour restructurer la hiérarchie lors de l'ajout d'une classe. Les techniques de réorganisation prennent en compte la structure des classes, les liens d'héritage et les références inter-méthodes et inter-attributs.

Le processus de réorganisation est construit autour de trois catégories de procédures qui opèrent sur une définition de classe ajoutée en feuille de la hiérarchie. Cette classe peut contenir des redéfinitions d'attributs, des rejets d'attributs hérités et des définitions d'attributs propres. La réorganisation s'effectue en plusieurs étapes :

1. Toutes les redéfinitions d'attributs sont décomposées en opérations primitives s'exécutant sur un ensemble limité de caractéristiques de classe. Par exemple, la redéfinition d'une méthode peut être décomposée en un renommage des arguments et une redéclaration de leur type (voir exemple ci dessous, Figure 2-8).

2. Des types de redéfinition sont assignés à des catégories. Cette typologie correspond aux différentes possibilités de dériver une sous-classe. Elle décrit plusieurs sémantiques du

mécanisme d'héritage. Par exemple, la redéclaration du type d'un attribut correspond à une catégorie (sous-typage) et l'ajout d'un attribut à une autre.

3. En accord avec ces catégories, la définition de la nouvelle classe est restructurée. Cette première réorganisation a pour objectif, en décomposant l'opération de spécialisation en opérations primitives, de mettre en évidence la manière de restructurer la hiérarchie.

4. La hiérarchie est réorganisée pour éliminer les utilisations illégitimes du lien d'héritage, celles qui ne correspondent pas à une des catégories définies. Cette étape est mise en œuvre par des procédures qui recherchent, pour les attributs qui sont redéclarés parce qu'ils ne pouvaient être hérités ou affinés dans leur forme actuelle, les classes où ils ont été déclarés. Pour chaque classe, on crée une classe intermédiaire contenant les attributs communs aux super-classes de la hiérarchie initiale et à la nouvelle sous-classe. La sous-classe est reliée à chaque classe intermédiaire plutôt qu'aux anciennes super-classes.

A chaque étape, le résultat peut être optimisé en éliminant les définitions. La hiérarchie est optimisée par l'élimination des définitions redondantes ou des définitions de classes n'introduisant aucune information supplémentaire par rapport aux super-classes.

Représentation schématique (exemple extrait de [Casais91]) :

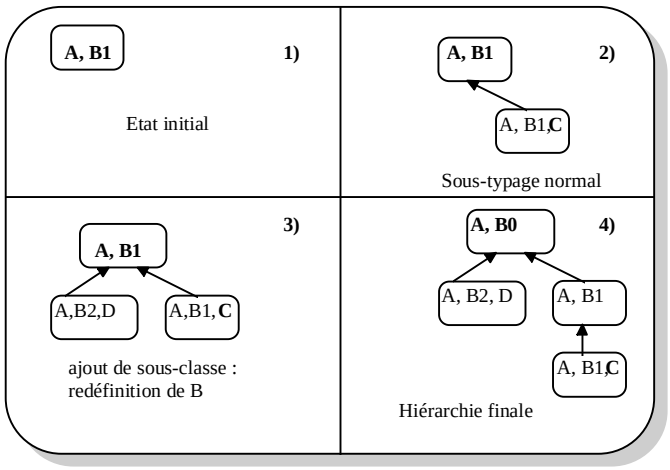


Figure 2-8 : Un exemple de restructuration simple

La configuration initiale (cas 1) possède une seule classe composée des attributs A et B1. Le cas idéal est la spécialisation d'une classe, sachant que la description de cette classe est héritée complètement dans les sous-classes comme pour la classe AB1C (2). Il existe l'autre cas, défavorable, où la classe rejette ou redéfinit un attribut ; dans notre exemple, la classe AB2D redéfinit B1 (3). Comme cette classe ne redéfinit que partiellement sa super-classe, cette modification introduit une inconsistance qui sera résolue par la réorganisation de la hiérarchie (4). B0 représente les propriétés communes à B1 et B2. Ces deux attributs modélisent deux

représentations différentes pour la même information abstraite spécifiée par B0. D'autres cas plus complexes peuvent être trouvés dans [Casais91].

Dans l'approche algorithmique, l'évolution est exprimée par l'ajout d'une classe comme feuille de la hiérarchie. La hiérarchie est réorganisée pour éliminer les incompatibilités introduites par l'évolution et obtenir une hiérarchie plus facilement exploitable et modifiable. Les algorithmes analysent les adaptations apportées par la nouvelle classe et réorganisent la hiérarchie. La nouvelle hiérarchie vérifie les contraintes de spécialisation et factorisent les propriétés. Les algorithmes garantissent la satisfaction des propriétés de non-redondance, de non-existence de références à des propriétés indéfinies, de préservation des définitions et d'une minimalité du nombre de classes.

Ce mécanisme est intéressant en phase de conception ; cela permet d'obtenir une hiérarchie correcte, suivant certains critères bien précis et facilement réutilisable. Mais en phase d'exploitation, les conséquences peuvent être coûteuses pour les objets ; une réorganisation totale des données peut s'avérer nécessaire, car la réorganisation des classes peut restructurer toute la hiérarchie.

2.2.1.3.2 La classification de classes

La classification de classes est un outil permettant aussi une construction incrémentale d'une hiérarchie de classes. L'objectif ici est de trouver la position adéquate d'une classe dans un graphe de classes [Haton91] [Napoli92]. Cette approche s'avère intéressante lorsqu'une base de connaissances atteint un nombre important de classes. En effet, elle fournit une assistance à l'utilisateur pour insérer une description de classe plutôt que de réaliser cette insertion manuellement.

Cette méthode de construction a été développée sur le modèle de représentation de connaissances SHIRKA [Rechenmann88]. Une classe dans SHIRKA est une structure regroupant un ensemble de propriétés, voire des attributs typés. L'héritage est simple et chaque instance appartient à une classe dont elle value les attributs. La base théorique de cet algorithme de classification est le formalisme de types : toute classe correspond à un type composé d'un ensemble de descriptions d'attributs ; la relation de spécialisation correspond alors à une relation de sous-typage. Comme les attributs peuvent être redéfinis, à chaque attribut de la base est associée une hiérarchie de types. La classification de classes revient alors à une succession d'insertions de types d'attributs dans le graphe de types correspondant.

La définition d'un attribut dans une classe est une combinaison des définitions associées à cet attribut dans les super-classes. Le type d'un attribut dans une classe est l'ensemble des valeurs possibles pour cet attribut qui est déterminé par l'intersection des types de cet attribut dans la classe et les super-classes où l'attribut est redéfini. La détermination du type d'un attribut

permet de construire, ensuite, la hiérarchie de types associée à cet attribut. Cette hiérarchie est construite en effectuant un parcours descendant du graphe de spécialisation.

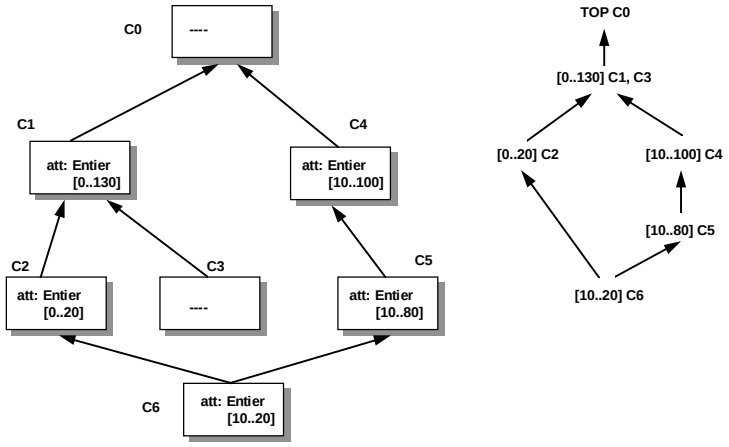


Figure 2-9 : Graphe d'attribut associé à l'attribut att

L'insertion d'une classe C dans un graphe de classes correspond à insérer chaque attribut de C dans le graphe des attributs. La position d'un attribut *att* dans son graphe fournit, d'une part, l'ensemble des super-classes possibles pour C, et d'autre part, l'ensemble des sous-classes possibles de C. La position d'une classe dans le graphe de classes est alors obtenue par combinaison des positions des attributs dans le graphe de type associé. Cette combinaison est une intersection des positions vérifiant la propriété d'héritage complet. Le résultat final peut contenir plusieurs solutions. Dans le cas d'un héritage simple, l'utilisateur devra choisir une des super-classes pour fixer la position de la classe C. Sinon, dans le cas d'un héritage multiple, la solution optimale peut être déduite sans l'intervention de l'utilisateur (la solution est rendue minimale selon la relation de transitivité [Capponi94]). Cet algorithme peut être utilisé lors d'un ajout de classe ou d'une modification d'une classe existante ne respectant pas les invariants du graphe de spécialisation.

Cette approche est intéressante dans le sens où elle tend vers une formalisation d'une hiérarchie de classes par les types. L'approche par classification permet, en général, une insertion correcte des classes d'objets, ce qui est nécessaire dans une grande base de connaissances.

2.2.1.3.3 Conclusion sur la réorganisation

Dans l'approche par réorganisation d'une hiérarchie de classes, nous avons étudié deux approches différentes. L'une (approche algorithmique) tente de restructurer la hiérarchie en éliminant, entre autres, les redondances d'informations et en factorisant les propriétés communes. Et l'autre (approche par classification) permet l'insertion d'une classe dans une hiérarchie de classes en recherchant la position commune à tous les attributs de cette classe. Ces deux approches s'avèrent utiles pour mieux définir une hiérarchie complexe de classes.

Après l'étude des différentes approches (Correction, Versionnement et Réorganisation), nous présentons les techniques concernant l'impact des modifications de classes sur les instances.

2.2.2 Adaptation des instances

L'ensemble des approches présentées ci-dessus décrit parfaitement les conséquences des modifications de schémas sur les schémas eux mêmes. Cependant rien n'est précisé sur l'impact de ces modifications sur les instances [Nguyen89b] ; cette propagation peut être effectuée de trois façons : la conversion, l'émulation et le versionnement. La conversion, la plus utilisée [Banerjee87] [Penney87] [Lerner90], consiste à adapter physiquement les instances concernées par la modification de leurs classes d'appartenance. Tandis que l'émulation [Skarra86] consiste à masquer les évolutions de schémas par des procédures d'exceptions, et le versionnement [Clamen92] a pour objectif de créer une version des instances pour chaque version de classe. L'approche par versionnement peut être combinée avec la conversion [Monk93].

2.2.2.1 Conversion

La conversion consiste en la mise à jour physique de toutes les instances qui ne sont pas compatibles avec la nouvelle définition de leurs classes d'appartenance. La conversion a pour objectif d'adapter les instances à la définition de leurs classes. Cette adaptation se traduit par une augmentation ou une réduction d'informations. Par exemple, l'ajout d'un attribut déclenche son ajout au niveau des instances. La valeur affectée à cet attribut peut être une valeur par défaut associée au type de l'attribut (null pour une classe, 0 pour un entier, etc.). Tandis que la suppression d'un attribut entraîne sa suppression dans les instances de la classe où il est supprimé.

La conversion peut être prise en compte par deux stratégies:

- La conversion immédiate où les instances sont mises à jour automatiquement dès l'évolution de leur classe. Cette technique, adoptée par GEMSTONE, offre l'avantage d'avoir à tout moment une base d'objets valide. Par contre, elle n'est pas très performante lorsque le nombre d'objets concernés est grand. Afin d'améliorer les performances de la répercussion, GEMSTONE regroupe les modifications concernant les mêmes instances, afin d'éviter les accès redondants aux objets. Par exemple, lorsqu'on ajoute plusieurs variables d'instances à une classe, il faut restructurer les instances, mais il faudrait le faire en un seul passage.

- La conversion différée ou retardée, adoptée entre autres par ORION [Banerjee87] et OTGEN [Lerner90], consiste en la mise à jour des instances dès qu'on y accède. Cette technique peut en réalité optimiser le coût d'une propagation, dans le cas d'une grande base de données. Toutefois, elle possède l'inconvénient de laisser la base physique temporairement incohérente

jusqu'à l'accès à toutes les instances concernées par la propagation. La conversion nécessite de gérer un historique des modifications de chaque classe. De plus, la conversion présente l'inconvénient de dégrader les performances du système car l'accès aux instances en est retardé.

2.2.2.2 Emulation

L'émulation, qui est l'approche suivie par le système ENCORE [Skarra86], consiste à masquer les évolutions de schémas au niveau des objets. Comme décrit au §2.2.1.2, l'évolution d'une classe engendre la création d'une version de cette classe. Il faut alors, pour chaque nouvelle version, programmer des procédures d'exception (pre et post handler) pour que la lecture et l'écriture d'un attribut ne provoquent pas d'erreur. Par exemple, si un attribut est supprimé dans une nouvelle version, il faut définir une exception en lecture qui renverra une valeur nulle et une exception en écriture qui sera sans effet.

ENCORE offre une autre façon de maintenir les instances en permettant une conversion manuelle. En effet, si une classe est modifiée, les instances de cette classe peuvent être converties conformément à la nouvelle définition ("coercion"), et ceci par l'utilisation d'une opération ("convert"), qui contraint l'instance à correspondre à la nouvelle définition. Cette opération est explicite, elle est demandée par l'utilisateur. Lors de la reconfiguration des instances, les propriétés et les informations non présentes dans la nouvelle classe sont perdues, l'instance n'est plus valide pour l'ancienne classe. L'utilisation de la conversion ne résout pas le problème de l'utilisation des anciennes versions dans les programmes.

Cette même approche de conversion est utilisée dans CLOSQL [Monk93] pour adapter les instances aux différentes versions, à la différence que ce système propose le moyen de revenir en arrière, donc de convertir une instance vers une version antérieure.

L'émulation, en utilisant les exceptions, nécessite l'adaptation des versions de classes, pour permettre une bonne utilisation des instances des autres versions de classes. Cette émulation peut entraîner une dégradation des performances si le nombre d'incompatibilités entre les instances et les programmes augmente.

2.2.2.3 Versionnement

Cette stratégie est proposée dans [Clamen92], où l'évolution d'une classe engendre comme dans ENCORE la génération d'une version de classe, qui engendre à son tour une version de toutes les instances de la classe.

Après la modification d'une classe, lorsqu'on accède à une instance, le système crée une version de cette instance, conforme à la nouvelle définition de la classe. Des contraintes et des

dépendances de partage [Djeraba93a] sont spécifiées entre les versions d'instances d'une même classe. Elles décrivent les attributs partagés, les relations de dérivation des valeurs d'attributs et les dépendances générales entre les attributs des versions. La modification d'un attribut d'une version peut provoquer la mise à jour des autres versions (par exemple, recopie de valeurs) pour maintenir les contraintes et les dépendances. La mise à jour d'une version est différée jusqu'à son utilisation.

Cette approche préserve bien la cohérence des instances et la compatibilité des programmes, mais engendre un nombre important de versions (versions de classes et d'instances) ce qui complexifie la navigation dans la base.

2.2.3 Bilan

Il existe trois grandes approches complémentaires pour assurer l'évolution de schémas :

- L'approche par correction qui est un minimum à assurer ; cette approche est mise en œuvre par un ensemble d'opérations respectant les invariants du modèle de données.
- Le versionnement, où la modification d'une classe entraîne la création d'une version de classe, ce qui permet de garder un historique des changements et de faire partager les différentes versions de classes entre plusieurs utilisateurs.
- La réorganisation de classes qui consiste à reconstruire la hiérarchie de classes (cf. § 2.2.1.3).

Les deux dernières approches sont utilisées dans des bases (de données ou de connaissances) de taille assez importante ; elles constituent des pistes intéressantes pour le futur.

Dans les trois approches citées ci-dessus, le problème de l'impact des modifications de classes sur les instances n'est pas abordé. Il existe trois techniques pour assurer cet impact.

- La conversion, qui consiste à convertir les instances concernées par la modification de leur classes d'appartenance. La conversion peut être immédiate [Penney87], après la modification de la classe, ou différée [Banerjee87], lors de l'accès aux instances.
- L'émulation [Skarra86] qui a pour objectif de masquer la dérivation d'une version de classe par des procédures d'exceptions.
- Le versionnement [Clamen92] dont le but est de créer une version des instances pour chaque version de classe.

L'inconvénient commun à toutes ces approches est la forme sous laquelle se présente l'évolution de schémas. Souvent, il faut passer par des langages de programmation qui ne facilitent ni l'expression des besoins en évolution, ni la maintenance des opérations d'évolutions. L'inconvénient précédent oblige les concepteurs des systèmes à proposer une et une seule politique d'évolution de schémas, ce qui ne correspond pas souvent aux besoins des utilisateurs.

ADELE3 [Ahmed-Nacer94], à notre connaissance, est le seul système à proposer une notion de politique d'évolution qui permet d'associer à un type ou un schéma un ensemble de déclencheurs contrôlant leur évolution. Un schéma de l'application est défini par une composition de types ; redéfinir la politique par défaut revient à redéfinir la relation de composition de ce schéma, et d'associer à cette nouvelle relation des triggers qui expriment le contrôle de l'évolution du schéma. Le concept de politique ou de stratégie n'est pas défini clairement ; rien n'est dit sur l'organisation des politiques. De plus, le schéma décrivant l'évolution (ensemble des triggers) est intégré dans le schéma de l'application, ce qui ne facilite pas la réutilisation.

2.3 EVOLUTION D'INSTANCES

Nous avons consacré la première partie de cet état de l'art à l'évolution de schémas, nous allons maintenant présenter une synthèse sur l'évolution des instances. L'évolution des instances se traduit soit par un changement de valeurs d'attributs, soit par un changement de classes d'appartenance.

Dans les différents cas d'évolution, trois questions principales se posent :

- Est-ce que la modification effectuée (ex : la valeur de l'attribut est modifiée) est cohérente ou non (ex : les cotés d'un carré ne sont plus égaux) ?
- Est-ce que la modification effectuée implique des conséquences sur d'autres objets (ex: le côté d'un carré a été modifié, faut-il recalculer la surface) ?
- Faut-il garder une trace dans le temps des modifications effectuées ?
- Est-ce qu'une instance peut changer de classes d'appartenance ?

Les réponses à ces questions constituent en elles-mêmes des grands axes de travaux.

2.3.1 Contraintes d'intégrité

Pour maintenir les objets cohérents, le modèle doit offrir à l'utilisateur la possibilité d'exprimer des contraintes que le système doit vérifier. Dans le cas où ces contraintes sont

violées, on peut soit refuser les modifications de valeurs, soit essayer de rétablir la cohérence. La notion de contraintes existe principalement dans les bases de données, et a été étendue plus tard aux bases de données orientées objet. Les systèmes diffèrent suivant la nature de la cohérence à assurer (partielle, totale, interne, externe, etc.), la nature des contraintes (structurelles, comportementales, etc.) et le processus de vérification (quand et comment vérifier).

2.3.1.1 Typologie des contraintes

Les contraintes peuvent être classifiées en fonction la cohérence qu'elles veulent exprimer :

- Les *contraintes structurelles* spécifient la sémantique du modèle de données. Parmi elles, on retrouve les contraintes qui modélisent la disjonction des classes, les liens de dépendances (exclusivité, partage), le caractère obligatoire d'un attribut, etc. [Bouaziz95]. Les contraintes structurelles sont exprimées de manière déclarative au niveau de la description des classes.

- Les *contraintes comportementales* assurent que les méthodes n'introduisent pas d'incohérences. Ces contraintes sont mises en œuvre à l'aide de pré et post conditions, vérifiées avant et après l'exécution d'une méthode [Defude93] [Meyer88]. Ce type de contraintes peut aussi être mis en œuvre par des règles actives [Bounaas94a].

- Et finalement, les *contraintes applicatives* ont pour objectif de modéliser la cohérence d'une application. Ces contraintes sont exprimées au niveau du schéma de l'application par des contraintes déclaratives (contraintes de domaine, unicité, caractère obligatoire, etc.) ou par des règles actives [Bouaziz95]. On y retrouve les contraintes qui mettent en jeu un attribut (intra-attribut), plusieurs attributs d'une classe (inter-attribut) ou plusieurs attributs de classes différentes (inter-classes).

2.3.1.2 Contrôle de l'intégrité

Le processus de vérification a besoin de connaître le moment de tester les différentes contraintes. Lorsqu'il s'agit de contraintes structurelles, elles sont vérifiées par et lors des opérations de mise à jour. Par exemple, les contraintes modélisant un lien d'exclusivité sont vérifiées avant l'opération de mise à jour d'un attribut. Le processus de vérification a pour tâche de rétablir les contraintes lorsqu'elles sont violées. Pour cela, il applique une politique qui peut être un rejet ou une propagation de l'opération.

Pour les autres types de contraintes (comportementales et applicatives), l'utilisateur indique quand il faut les vérifier et comment réagir lorsqu'elles sont violées.

La mise en œuvre d'un système d'intégrité doit donc proposer un langage d'expression des contraintes et un mécanisme de contrôle d'intégrité. L'avantage d'une approche par les règles actives est de proposer en même temps un langage d'expression, pouvant être utilisé pour les contraintes, et un mécanisme de déclenchement de règles qui assure le contrôle d'intégrité [Bouaziz95]

2.3.1.3 Les travaux sur les contraintes d'intégrité

Dans ce paragraphe, nous citons quelques travaux sur les contraintes d'intégrité.

Kheops [Bouzeghoub91] propose un langage permettant de spécifier des contraintes d'intégrité. Après un contrôle de compatibilité ou de redondance des contraintes, le système génère des méthodes (en O2C) représentant ces contraintes.

Thémis [Benzaken93] est un langage de bases de données permettant d'exprimer des contraintes d'intégrité utilisant la logique du premier ordre avec intégration des appels de méthodes dans les contraintes. Il est proposé dans ce système une technique d'optimisation pour la vérification des contraintes. L'objectif est de détecter statiquement quelles sont les contraintes qui doivent être vérifiées pendant l'exécution d'une transaction afin de réduire le coût de la vérification.

Dans ContAct [Defude93], les contraintes sont définies sur les méthodes en spécifiant les pré et les post conditions. Ces conditions sont définies dans la signature des méthodes. Une exception est déclenchée si une contrainte est violée.

2.3.1.4 Conclusion sur les contraintes d'intégrité

Les contraintes d'intégrité contrôlent l'évolution des instances car elles permettent de vérifier la cohérence des valeurs modifiées, et/ou de réagir à la violation d'une contrainte pour maintenir la cohérence. Le système d'intégrité qui assurera la gestion de ces contraintes devra assurer l'expression, la manipulation et le maintien de ces contraintes. Les règles actives existant dans la plupart des systèmes [Medeiros91] jouent les rôles de langage d'expression et de mécanisme de mise en œuvre des contraintes d'intégrité.

2.3.2 Les versions

Une version d'instance est un état de l'objet que le système ou l'utilisateur veut conserver [Agrawal90]. L'objectif des versions est de garder une trace dans le temps de l'évolution d'une instance. Le fait de créer une version d'une instance, lorsque celle-ci est modifiée, permet de garder les anciennes versions qui sont cohérentes par rapport aux autres instances. Les versions représentent un thème qui concerne plusieurs domaines tels que les bases de

données, la CAO, le génie logiciel, les bases documentaires, etc. Les pionniers sur ce thème étant bien sûr les concepteurs de logiciel. Les domaines techniques sont appropriées pour la gestion des versions. La gestion de versions est aussi utile dans le domaine de la CAO [Katz87]. Plusieurs concepteurs veulent, d'une part partager des objets ; ce partage est résolu en affectant une version de l'objet à partager à chaque concepteur. D'autre part, les concepteurs veulent garder l'historique de la conception d'un produit, qui se traduit par un arbre de versions (Figure 2-10).

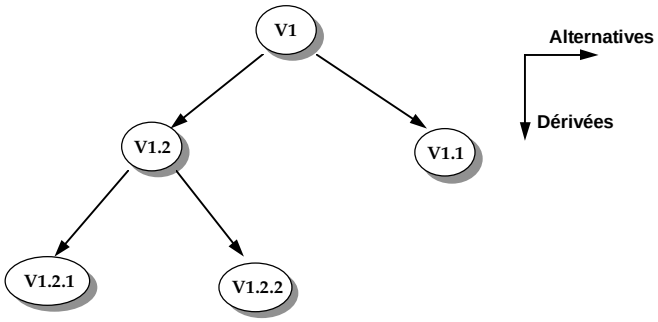


Figure 2-10 : Arbre de dérivation

Une alternative représente une évolution différente d'une même entité qui traduit le choix d'une nouvelle solution. Par contre, une dérivée représente une succession de changement d'état d'une version.

Plusieurs prototypes ou systèmes implémentent un gestionnaire de versions d'objets. Nous n'allons pas les citer tous [Skarra86] [Adele93], mais nous en présenterons un seul, celui d'ORION [Kim88] qui propose une approche un peu différente de celle des autres modèles de versions.

Les versions dans Orion

Un objet est soit “versionnalisable”, instance d'une classe déclarée versionnalisable, soit non versionnalisable. ORION distingue deux types de versions, les versions temporaires ("transient") qui peuvent être modifiées ou détruites, et les versions de travail ("working") qui peuvent être détruites mais non modifiées. L'usager peut explicitement promouvoir une version temporaire en une version de travail. La promotion d'une version temporaire en une version de travail peut être implicite si une version temporaire est dérivée de celle-ci.

Du moment qu'il est possible de dériver d'une version existante des versions temporaires, un objet versioné consiste en une hiérarchie de versions appelée hiérarchie de dérivation de versions. ORION utilise le terme de version d'instance pour référencer un nœud spécifique de la hiérarchie de dérivation, et le terme instance générique pour référencer l'entité abstraite d'un objet versionné. A chaque version d'instance est attachée une instance générique qui maintient l'historique des dérivations des versions d'instances d'un objet versionné.

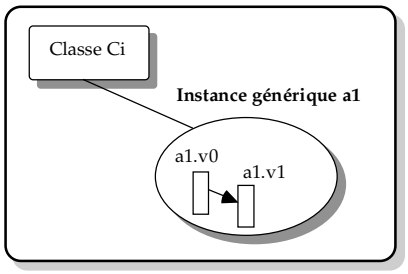


Figure 2-11 : Instance générique

Conclusion sur le versionnement

L'évolution des instances engendre plusieurs points critiques. Parmi ces points, la gestion du stockage de l'historique des données. En effet, il est nécessaire pour les utilisateurs de garder une trace de leurs données dans le temps. Le problème des versions ne constitue pas, dans le cadre de cette thèse, une priorité de notre travail, car il a déjà été étudié dans le cadre du système SHOOD [Djeraba93b].

2.3.3 La classification d'instances

La classification d'instances permet de rattacher (en général) une instance à la classe la plus spécifique. Elle permet de trouver la classe d'appartenance d'une instance en fonction des informations qu'elle détient. Ce mécanisme est souvent nécessaire lorsque l'instance, après plusieurs modifications, n'est plus conforme à sa classe.

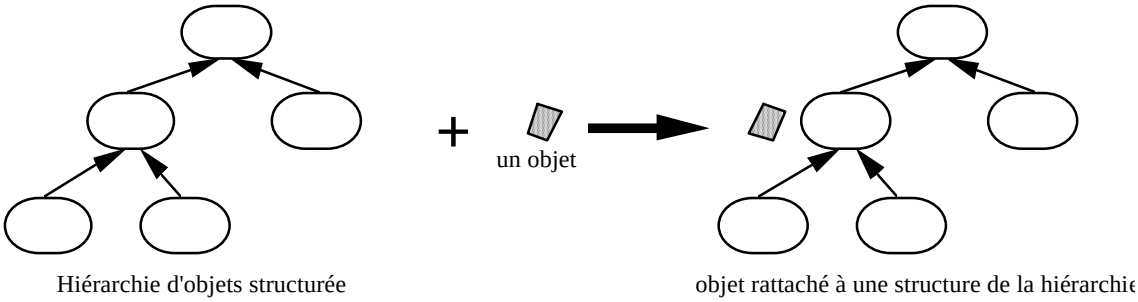


Figure 2-12 : la classification d'instances

Un mécanisme de classification dans un Système de Représentation de Connaissances (SRC) consiste en général à "comparer" l'objet à classifier avec les objets déjà présents (Figure 2-12). Cette comparaison, appelée aussi appariement, met en évidence les dépendances qui existent entre les objets, selon l'ordre partiel qui a permis de gérer la hiérarchie. Cette relation d'ordre partiel est souvent la relation d'héritage (elle peut être une relation plus générale : une relation de subsomption [Napoli92]). La classification d'un objet O divise l'ensemble des objets de la hiérarchie en trois familles :

- La famille des objets *X* avec lesquels *O* est *comparable*. Cette famille correspond aux schémas de classes qui ont un statut "sûr"[2] à la fin de la classification. Les objets X et O sont comparables, s'ils possèdent les mêmes propriétés et que les contraintes sont respectées.

- La famille des objets *X* avec lesquels *O* n'est *pas comparable*. Les objets X et O ne sont pas comparables si X et O n'ont pas de propriété commune, si la compatibilité des contraintes n'est pas vérifiée ou si les objets X et O possèdent chacun des propriétés que l'autre n'a pas.

- La famille des objets *X* avec lesquels la comparaison de *O* n'est *pas décidable*. Il manque des informations pour décider de la compatibilité ou de l'incompatibilité de l'objet O avec les objets X. Les systèmes qui traitent cette famille lors de la classification permettent de traiter dans une certaine mesure les objets incomplets (ex : dans le modèle SHIRKA, cette famille possède le statut "possible").

Nous citons maintenant un système de représentation de connaissances offrant un mécanisme de classification d'instances, en l'occurrence TROPES [Marino93].

Classification dans Tropes

Le but de la classification dans TROPES [Marino91] est la localisation la plus fine possible d'une instance d'un concept dans les différents points de vue de ce concept. Ce mécanisme est basé sur la comparaison instance/classes, un parcours original entre les points de vue grâce aux passerelles, et la possibilité de raisonnements différents. Ce mécanisme est inspiré du modèle SHIRKA [Rechenmann88].

Le résultat de la classification est matérialisé par une partition du graphe des classes de chaque perspective en trois sous-ensembles : sûr, possible, impossible. La classe sûre la plus spécialisée dans chaque perspective (point de vue) représente le but recherché par la classification.

Le partitionnement du graphe de classes, par le mécanisme de classification, en classes sûres, possibles et impossibles prend en compte les objets incomplets (statut possible), mais sans tenir compte du "degré" d'incomplétude [Nguyen89b]. En effet, certaines classes sont peut-être plus possibles à l'accueil desobjets que d'autres : le modèle n'intègre pas cette nuance. Le mécanisme de classification de TROPES n'intègre pas non plus la gestion des objets incohérents : le schéma d'instance doit coller exactement au schéma de classe le décrivant dès que les attributs possèdent une valeur.

[2] Statut "sûr" : Classe pouvant accueillir l'instance sans ambiguïté

Ce mécanisme de classification est performant du fait de la spécialisation simple dans chaque perspective, de l'hypothèse de la disjonction des classes dans une même perspective et des passerelles entre les perspectives. De ce fait, un nombre minimal de classes est exploré et beaucoup de statuts de classes sont déduits de ces connaissances. En revanche, ces choix au niveau du modèle TROPES demandent a priori beaucoup d'efforts de modélisation. TROPES propose aussi une extension de ce mécanisme de classification pour gérer la classification des objets composites [Marino91].

2.3.4 Bilan

L'évolution des instances est un thème qui regroupe plusieurs thèmes et chaque système essaie d'en traiter une partie. Dans cette section, nous avons présenté très brièvement trois grands axes :

- **Les contraintes d'intégrité** : Elles permettent d'exprimer et de mettre en œuvre la cohérence des applications. Une contrainte d'intégrité exprime un prédicat que le système doit vérifier pour assurer la cohérence des données. Ces contraintes sont soit mises en œuvre à l'aide du formalisme de règles actives, soit intégrées dans le modèle de données (par exemple : dans la description de la classe), soit mises en œuvre par les deux techniques simultanément. De même, le processus de vérification est soit pris en charge par un mécanisme de règles actives, soit intégré dans le système (dans les opérations de mise à jour). Nous constatons au niveau de cette synthèse que les règles actives sont sollicitées pour être aussi bien un langage d'expression qu'un mécanisme de contrôle de l'intégrité.

- **Les versions** : Moyen de garder une trace dans le temps des différentes évolutions d'une instance. Le mécanisme de gestion de versions est nécessaire dans les systèmes à objets qui manipulent une grande quantité d'informations, en particulier dans les environnements de conception.

- **Classification** : Mécanisme permettant de trouver la ou les classes d'appartenances d'une instance en fonction des informations qu'elle détient. La classification de l'instance peut entraîner dans certains systèmes le changement de ses classes d'appartenance et dans d'autres l'évolution de ses valeurs (l'instance est complétée).

Il existe d'autres techniques pour faire évoluer une instance, telle que la dérivation de valeurs. Ce mécanisme est souvent retrouvé dans les systèmes à base de connaissances ou les langages à base de frames ; une valeur d’un attribut peut être calculée en fonction d’autres valeurs d’autres attributs. Lorsqu'une valeur d'un attributs A est modifiée, et que celle-ci intervient dans le calcul de la valeur d'un attribut B, alors B doit être recalculé. Ce type de mécanisme,

qui permet d'inférer des valeurs d'attributs, est maintenu soit automatiquement par des mécanismes comme les systèmes de maintien de la vérité [Euzenat87], soit manuellement par des règles actives, et dans ce dernier cas on retrouve le même principe que pour le maintien des règles d'intégrité.

2.4 CONCLUSION

Nous pouvons retenir de ce chapitre qu'il existe plusieurs approches pour assurer l'évolution de schémas ou d'instances. Ces approches sont souvent complémentaires et chaque approche représente souvent à elle seule un axe de travail important. Dans le cas de notre système, nous allons choisir d'assurer l'évolution de schéma en utilisant l'approche par correction. Nous insistons sur l'expression des règles (règles d'évolution) permettant de décrire la sémantique des opérations manipulant les classes du modèle de connaissances. En effet, la plupart des travaux n'abordent pas les difficultés liées à l'expression de l'évolution de schémas et du fait que la sémantique d'évolution des schémas est figée et imposée indifféremment à tous les schémas. Seul ADELE3 propose une notion de politique d'évolution associé aux relations de composition d'un schéma. Cette notion de stratégie n'est proposée que pour l'évolution de schémas.

Pour ce qui est de l'évolution des instances, celle-ci est souvent contrôlée par des contraintes d'intégrité. Dans les systèmes à base de connaissances, on retrouve généralement un mécanisme de classification d'instances qui permet de faire évoluer une instance dans la hiérarchie de classes suivant les connaissances que l'instance détient. Dans le cadre de notre étude, nous avons d'une part réalisé un mécanisme de classification d'instances présenté dans le chapitre 5. Nous permettrons d'autre part, l'expression des contraintes d'intégrité et des calculs inférentiels à travers la notion de règle d'évolution.

Dans le chapitre suivant, nous allons décrire le modèle de connaissances SHOOD. Nous présentons les différents invariants de ce modèle que toute opération ou mécanisme faisant évoluer une classe ou une instance devra respecter.

3. LE MODELE SHOOD

SHOOD [Escamilla93] est un modèle de représentation de connaissances par objets [Masini89], inspiré des langages de programmation à objets [Giambiasi91] comme CLOS [Bobrow88], EIFFEL [Meyer88], OBJVLISP [Cointe87], des langages de représentation de connaissances tels que ROME [Carre89], OBJLOG [Dugerdil88], YAFOOL [Ducournau88], des langages à base de frames tel que KRL [Bobrow77] et SHIRKA [Rechenmann88] et des bases de données objet comme ORION [Banerjee87] et GEMSTONE [Penney87].

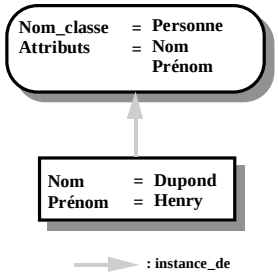
La motivation principale pour la conception du système SHOOD est la définition d'un Système de Représentation de Connaissances(SRC) adapté aux caractéristiques des objets manipulés dans les applications CAO. Ce modèle supporte la définition et la manipulation d'objets évolutifs qui peuvent être incomplets et temporairement incohérents. Le modèle utilisé pour la représentation des connaissances repose sur la notion d'objet. Tous les concepts du modèle sont représentés par des objets. C'est pour cela qu'on dit que SHOOD est basé sur la notion du **tout objet** [Nguyen92b].

Dans les paragraphes qui suivent, nous présentons les différents éléments du modèle SHOOD, la structure des objets, les méthodes, la méta-circularité, etc. Puis, nous détaillons chacun de ces concepts ; au cours de cette description nous déduisons un ensemble d'invariants que devra respecter tout mécanisme d'évolution. Un invariant décrit la cohérence statique d'un état du système. L'ensemble des invariants est composé de huit Invariants pour le graphe de Spécialisation (I.S.n) et de six Invariants pour le graphe d'Instanciation (I.I.n). Ces invariants sont en partie ceux décrits dans [Escamilla93].

3.1 LES CONCEPTS DE BASE

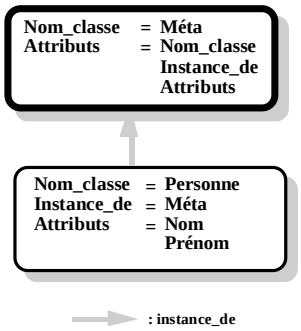
Les connaissances dans le modèle SHOOD sont représentées par une structure abstraite de données appelée "classe". Une classe est définie par un ensemble d'attributs auxquels sont associés un ou plusieurs descripteurs. Une classe décrit un ensemble d'objets appelés "instances" partageant la même structure et le même comportement.

Définition : L'appartenance d'une instance à sa classe est modélisée par un lien d'instanciation où instancier une classe signifie affecter des valeurs aux attributs de cette classe. SHOOD permet la multi-instanciation ; cela signifie qu'une instance peut être rattachée à plusieurs classes de sémantiques différentes, en valuant l'union des attributs.



L'instanciation de *Dupond* dans la classe *Personne* correspond à l'affectation de valeurs aux attributs de cette classe. Dans la figure ci-contre, la classe *Personne* est définie par les attributs *Nom* et *Prénom*, et la personne *Dupond*, instance de la classe *Personne*, possède les valeurs *Dupond* et *Henry* pour les attributs *Nom* et *Prénom*.

Comme dit précédemment, SHOOD utilise la notion de tout objet. De ce fait, une classe est modélisée par un objet ; cet objet (la classe) est, par le même mécanisme, lui-même l'instance d'une **classe**. La différence est conceptuelle : la **classe** à instancier possède des informations "systèmes" qui définissent une classe. La **classe** est dite méta-classe, elle est soit la classe Méta, soit une de ses sous-classes [Escamilla93].



La classe Personne est créée par instanciation de la méta-classe *Méta*. En tant qu'instance de *Méta*, la classe *Personne* value les attributs *Nom_classe* et *Instance_de* respectivement par *Personne* et par *Méta*. Le lien d'instanciation entre l'objet *Personne* et sa méta-classe est représenté par *Instance_de.* En valuant l'attribut *Attributs* par les attributs *Nom* et *Prénom*, la classe *Personne* possède un moule qui lui permettra à son tour d'être une classe d'instanciation.

SHOOD offre donc trois niveaux d'objets : les méta-classes, les classes et les instances terminales.

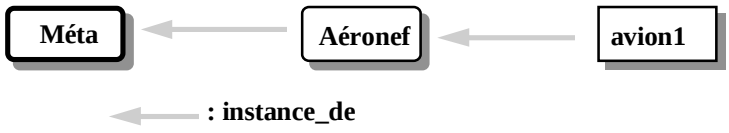


Figure 3-1 : Les trois niveaux d'objets dans SHOOD

Par ces trois types d'objets, on obtient un graphe d'instanciation à trois niveaux. Ce graphe respecte l'invariant suivant :

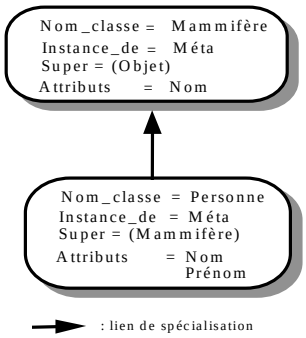
I.I.1 : Invariant du graphe d'instanciation

Le graphe d'instanciation est un graphe cyclique orienté connexe. Les nœuds de ce graphe représentent des objets. Tous les nœuds ont une étiquette unique, l'identificateur système (oid). Les liens d'instanciation sont représentés par des arcs uni-directionnels allant des objets vers leurs classes d'instanciation. Le seul cycle existant dans le graphe est le lien d'instanciation allant de *Méta_a_clef* (cf. § 3.6) vers elle même. Seuls les liens d'instanciation d'un objet vers ses classes d'appartenance les plus spécialisées sont représentés.

L'ensemble des classes forme un graphe (voir invariant I.S.1) dans lequel chaque classe-fille (sous-classe) hérite de l'ensemble des propriétés de ses classes-mères (super-classes). Cette relation est implantée par un mécanisme d'héritage.

I.S.1 : Invariant du graphe de spécialisation

L'ensemble des classes forme un graphe acyclique orienté connexe et à racine unique. Les classes sont les nœuds et les liens (classe, sous-classe) sont représentés par des arcs unidirectionnels qui sortent des classes vers les super-classes. La racine est le nœud représentant la classe Univers. Tous les nœuds sont accessibles par le nœud étiqueté Univers à travers les liens de (classe, sous-classe). Ces liens ne forment pas de cycles. Tous les nœuds ont une étiquette unique, le nom d'une classe.



Dans la figure à coté, la classe Personne hérite de l'attribut *Nom* de la classe Mammifère. L'attribut *Prénom* est ajouté à la classe Personne.

Figure 3-2 : Mécanisme d'héritage

3.2 LES ATTRIBUTS

Tout attribut est défini par un nom et un ensemble de descripteurs. SHOOD distingue différents types de descripteurs tels que les descripteurs de type, d'inférence, de contrainte, etc.

3.2.1 Les noms des attributs

Dans le modèle SHOOD, l'hypothèse de base est que deux attributs de même nom provenant de classes différentes n'ont pas le même sens. Par conséquent, si le concepteur d'une base de

connaissances définit deux attributs respectivement dans deux classes différentes, c'est parce qu'il veut exprimer deux concepts différents, même si ces attributs portent le même nom.

Ainsi, tous les attributs définis dans une même classe ont un nom différent. Et pour résoudre les conflits d'héritage entre les attributs de même nom provenant de classes distinctes, SHOOD introduit le concept de nom complet. Le nom complet d'un attribut est défini par son nom, préfixé par le nom de la classe où il a été défini pour la première fois.

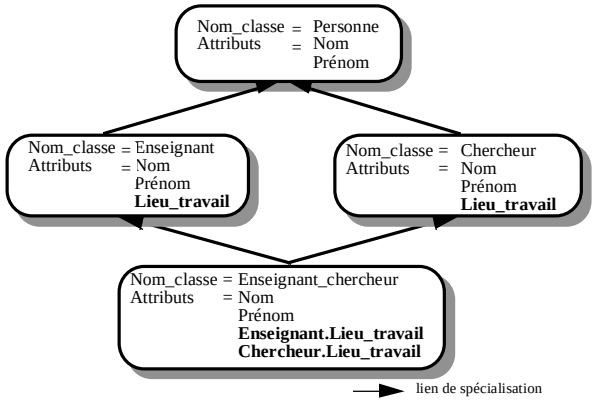


Figure 3-3 : Exemple de nom complet d'attribut

Dans l'exemple ci-contre, les classes Enseignant et Chercheur héritent des attributs *Nom* et *Prénom* de la classe Personne. La classe Enseignant_chercheur hérite des attributs communs *Nom* et *Prénom* définis dans la classe Personne. Par contre, elle hérite des deux attributs *Lieu_travail* de ces deux super-classes : ce sont deux notions différentes. En effet, les personnes appartenant à la classe Enseignant_chercheur ont bien deux lieux de travail : celui où elles enseignent et celui où elles effectuent un travail de recherche.

I.S.2 : Invariant de l'unicité du nom des attributs

Tous les attributs d'une classe ont un nom unique et chaque nom d'attribut est préfixé par sa classe d'origine (nom complet). Comme l'unicité du nom des classes est assurée par l'invariant du graphe de spécialisation, l'unicité du nom complet des attributs est assurée également.

I.I.2 : Invariant de l'unicité du noms des attributs

Tous les attributs valués par une instance ont des noms distincts. Comme une instance value les attributs de ses classes d'appartenance, d'après l'invariant précédent (I.S.2), les noms des attributs valués sont distincts. Comme une instance peut appartenir à plusieurs classes de sémantiques différentes, si dans ces dernières un même attribut est défini plusieurs fois, l'instance ne value qu'une seule fois cet attribut.

Remarque : Si une instance I est rattachée à deux classes A et B définissant chacune l'attribut AT, alors l'instance I value l'attribut AT en utilisant l'union des inférences définies respectivement dans A et B pour AT.

3.2.2 Les descripteurs

Un attribut est défini par un ensemble de descripteurs. Dans SHOOD, tout attribut est modélisé par une classe que nous appelons “classe-attribut”. Les classes attributs sont instances de la méta-classe *Méta_attribut*, ou d'une ou plusieurs de ses sous-classes. Ces méta-classes définissent spécifiquement la structure des attributs. L'ensemble des descripteurs possibles varie en fonction des méta-classes de la classe-attribut. Dans cette section nous présentons les descripteurs de base.

La définition d'un attribut est donnée par son nom et une liste de la forme ($desc_1$ val_1 $desc_2$ val_2 $desc_n$ val_n) où les $desc_i$ (i=1..n) représentent des identificateurs de descripteurs de base tels que type, inférence, contrainte ou tout autre descripteur défini dans une sous-classe de Méta-attribut (

Figure *3-4*), et les val_i (i=1..n) représentent les valeurs attribuées à ces descripteurs.

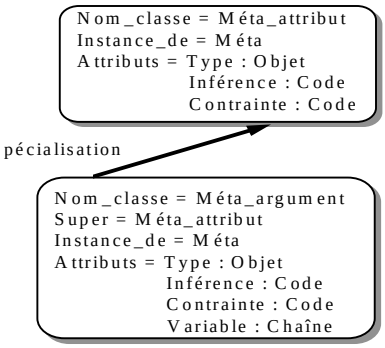


Figure 3-4 : Ajout de descripteurs

Un argument d’une méthode étant un attribut (cf. §3.3.2), il est défini par une métaclasse Méta_argument qui hérite des descripteurs de Méta_attribut ; Méta_argument ajoute le descripteur de variable.

3.2.2.1 Descripteur de type

Le descripteur de type permet de spécifier les valeurs qu'un attribut peut prendre. Ces valeurs peuvent être de deux types : des scalaires (entier, chaîne, ...) ou des instances (le domaine correspond à un nom de classe instanciable). Un descripteur de type est toujours composé de deux parties: un *constructeur* et un *domaine*.

Les constructeurs permettent de définir si un attribut est mono-valué ou multi-valué. Le seul constructeur mono-valué est le constructeur *un*. Les constructeurs multi-valués sont *tas*, *liste*, *ensemble ou séquence*.

Le *tas* est le constructeur le plus général où l’on admet la répétition des éléments et on ne tient pas compte de l'ordre. La *liste* permet la répétition des éléments tout en respectant l’ordre, c'est une spécialisation de *tas*. Une autre spécialisation de *tas*, l'*ensemble* où on ne tient pas compte de l'ordre mais on n'admet pas de répétition. Finalement, le constructeur *séquence* est une spécialisation multiple d'un *ensemble* et d'une *liste*, où l’on n'admet pas de répétition et où l’on respecte l'ordre. Du point de vue implémentation, les constructeurs sont

modélisés par des classes ne définissant aucun attribut ; chaque constructeur hérite des comportements (opérations d'accès) de son super-constructeur.

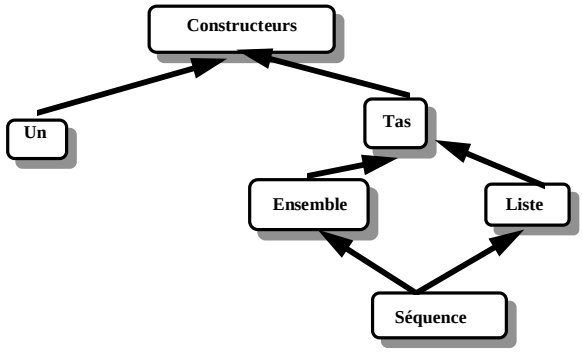


Figure 3-5 : La hiérarchie des constructeurs

Un constructeur peut être un constructeur de base (Un, Tas, Ensemble, Liste, Séquence) ou bien une composition de constructeurs de base, par exemple une liste de liste. La spécialisation d'un constructeur C composé de *n* constructeurs de base (C_i i=1..n) est la composition C' de *n* constructeurs (C'_j j=1..n) tel que C'_j est égal ou plus spécialisé que C_i avec i égal à j.

```
nom_classe = Etudiant
attributs = notes  : (Ens (Liste Entier))
```

La classe Etudiant décrit un attribut *notes* dont la valeur est un ensemble de listes d'entiers.

Figure 3-6 : Descripteur de type

3.2.2.2 Descripteur d'inférence

L'expression d'une inférence est utilisée pour obtenir la valeur d'un attribut. Une inférence est soit une expression Lisp, soit un appel de méthode SHOOD. A un attribut peuvent être associées plusieurs inférences organisées en un ordre total.

Exemple :

```
nom_classe = Personne
attributs =  age   : un Entier
             inferences ((exec_meth Cal_age $numss)
                         (exec_meth User)
                        )
```

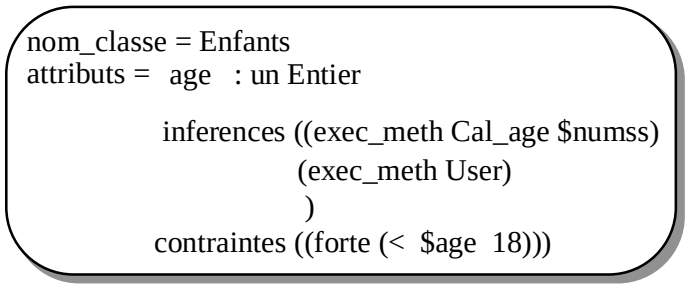
A l'attribut age sont associées deux inférences, l'une correspondant à une utilisation de la méthode Cal_age et l'autre à une utilisation de la méthode User. Si la méthode Cal_age échoue, par exemple si le numéro de sécurité sociale est inconnu, alors l'inférence User s'exécute, en demandant à l'utilisateur l'âge de la personne.

Figure 3-7 : Descripteur d'inférences

Si rien n'est précisé sur les inférences d'un attribut, celles-ci sont héritées des super-classes. Et si aucune inférence n'est héritée pour cet attribut, le système affecte automatiquement une inférence *User*. *User* est l'inférence par défaut qui accepte une valeur donnée par l'utilisateur.

3.2.2.3 Descripteur de contraintes

L'expression d'une contrainte permet une restriction sur les valeurs possibles d'un attribut. Une contrainte est réalisée par une expression Lisp ou un appel de méthode. Plusieurs conjonctions de contraintes peuvent être associées à un attribut. Si, pour une instance, l'ensemble des contraintes est satisfait, l'instance est dite cohérente [Favier90]. Un modèle de gestion de la cohérence des données permet de guider l'utilisateur vers une conception complète et cohérente des objets [Escamilla90]. Deux sous-groupes de contraintes existent : les contraintes fortes et les contraintes faibles. Les contraintes faibles peuvent être violées momentanément, tandis que les contraintes fortes doivent être impérativement respectées.

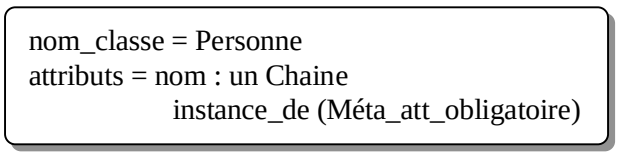


Dans cet exemple, on définit une contrainte pour vérifier que l'âge est inférieur à 18 ans pour les enfants.

Figure 3-8 : Descripteur de contrainte

3.2.3 Les attributs obligatoires

Il existe dans SHOOD des attributs obligatoires qui doivent toujours avoir une valeur et des attributs facultatifs qui peuvent ne pas avoir de valeur. Les attributs par défaut, instances de *Méta_attribut*, sont facultatifs. *Méta_attribut* admet une sous-classe *Méta_att_obligatoire* modélisant les attributs obligatoires. Pour définir un attribut obligatoire, il suffit de rattacher la classe-attribut à la méta-classe *Méta_att_obligatoire* par un lien d'instanciation. Par exemple, si l'on veut que l'attribut *nom* soit obligatoire dans la classe Personne, nous écrivons :



Le caractère facultatif des attributs permet de générer des instances incomplètes.

Figure 3-9 : attribut obligatoire

3.2.4 Les attributs pré-déclarés

Pour permettre une plus grande souplesse dans la sémantique attachée à chaque attribut, on définit la notion d'attribut pré-déclaré. Pré-déclarer un attribut dans une classe consiste à

définir un nom d'attribut dans une classe que ses instances ne peuvent pas valuer. La pré-déclaration d'un attribut définit son nom complet, car la classe de pré-déclaration est la classe d'origine de l'attribut (cf. § 3.2.1). Les attributs pré-déclarés permettent de définir la même sémantique pour un attribut dans tout le sous-graphe qui a pour racine la classe où il est pré-déclaré. Les attributs pré-déclarés sont hérités. Si un attribut pré-déclaré est ultérieurement défini dans une sous-classe, il devient instanciable dans cette classe et ses sous-classes.

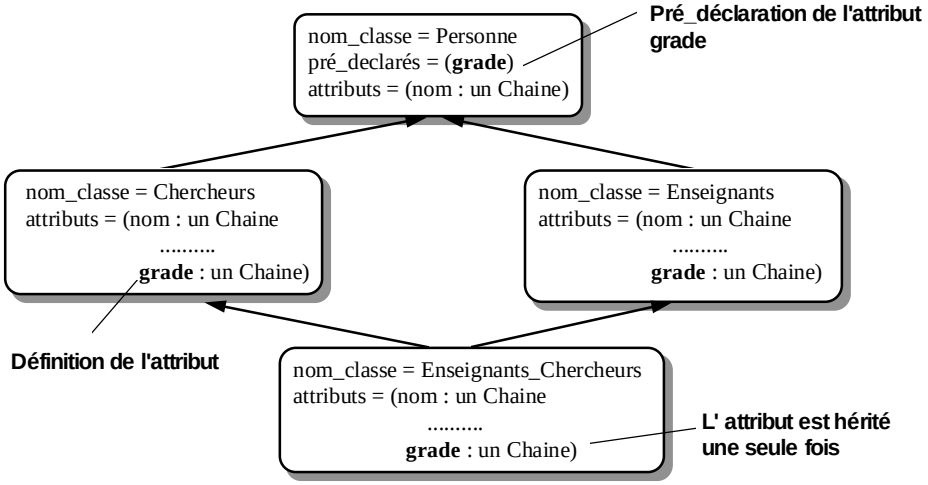


Figure 3-10 : Un attribut pré-déclaré.

Dans la figure ci-dessus, l'attribut *personne.grade* est pré-déclaré dans la classe Personne. Ceci permet de "fixer" sa sémantique. Il s'agit du même attribut qui est ensuite défini dans les classes Enseignants et Chercheurs puis hérité dans Enseignants_chercheurs. Sans la pré-déclaration, il aurait fallu définir l'attribut dans la classe Personne alors que cet attribut n'est pas une caractéristique pertinente de toutes les personnes. Par exemple, la classe Parents, sous-classe de Personne en hériterait, même si l'on juge inutile sa valuation pour les instances de cette classe.

Les attributs pré-déclarés permettent, entre autre, de résoudre les cas de conflits d'héritage d'attribut lorsque l'attribut ne doit pas être instancié dans les super-classes communes aux classes en conflit. Les attributs pré-déclarés permettent aussi de maintenir la cohérence de la base quand, par exemple, on élimine la définition d'un attribut de sa classe d'origine. Celui-ci devient alors un attribut pré-déclaré.

3.2.5 Les relations sémantiques

Généralement, dans un modèle orienté objet, la notion d'attribut permet de définir la structure d'un objet ; les attributs ont tous la même sémantique. En représentation de connaissances, il est parfois nécessaire d'exprimer une sémantique sur un attribut [Bobrow77]. Dans ce cas, l'attribut est vu comme une relation sémantique. Dans SHOOD, il existe deux types de relations sémantiques : les liens de dépendance et les liens de composition. Ces deux liens

sont indépendants pour pouvoir exprimer des liens de dépendance sur un attribut non-composite.

Nous présentons dans la suite les définitions sur les notions de lien de dépendance et de composition [Djeraba93b].

3.2.5.1 Lien de composition

Le lien de composition est plus connu dans la littérature sous le nom "est_partie_de" [Kim89]. Il permet de lier des objets appelés composants à un objet appelé composite. Un objet composite est un objet qui contient un ou plusieurs composants et donc un ou plusieurs liens de composition. Un graphe de composition est un graphe dont chaque arc est un lien de composition et dont chaque noeud est soit un objet composite, soit un objet composant, soit les deux à la fois. Exemple: Une voiture est composée de portes, d'un moteur, etc., et un moteur est composé d'un bloc-moteur, d'un démarreur, d'un alternateur, etc.

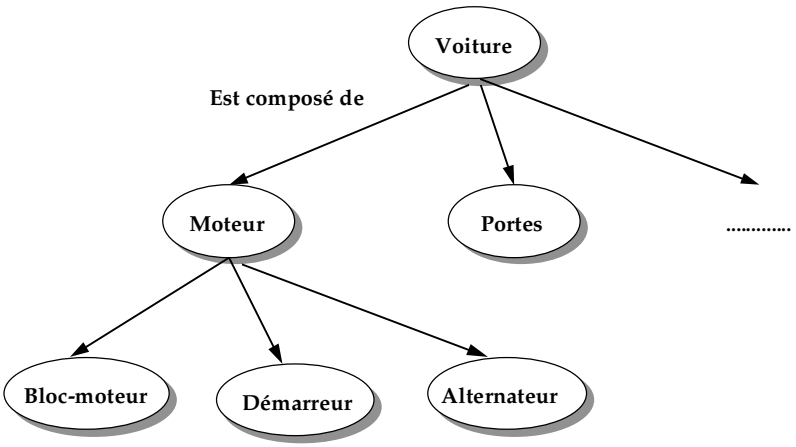


Figure 3-11 : Décomposition de la voiture

Le graphe de composition n'est pas forcément un arbre. Un composant peut être référencé par deux objets composites différents. Exemple : un chapitre peut être composant de deux livres distincts.

3.2.5.2 Liens de dépendance

Le lien de dépendance définit une sémantique descriptive et comportementale sur la relation entre deux objets. Entre autres, il peut caractériser le lien de composition décrit ci-dessus. La spécification d'un lien de dépendance entre deux objets permet d'exprimer une sémantique de partage, d'exclusivité, d'existence, etc. Ces différents types de dépendance ont des sémantiques qui interviennent dans la création, la suppression, le changement de lien, la duplication et enfin la consultation et la recherche des objets.

Une ou plusieurs dépendances peuvent être associées à chaque attribut lors de sa définition dans une classe. Un lien de dépendance peut être défini de manière générique (dans la classe), il sera alors valable pour toutes les instances de cette classe, de même qu'il est possible de le définir, de manière spécifique, pour une instance donnée.

Llien de dépendance partagé

Un objet (émetteur) peut être référencé par plusieurs objets (récepteurs) à travers des attributs de dépendance partagé. L'objet partagé (émetteur) a une représentation unique, il n'est pas dupliqué. Que ce soit du point de vue utilisateur ou système, il s'agit d'un seul objet. Les récepteurs peuvent appartenir à la même classe ou à des classes différentes. Exemple : un livre est composé d'un ensemble de chapitres et de références bibliographiques. Une même référence bibliographique peut appartenir à deux livres différents. Par conséquent, la suppression ou la modification de la référence bibliographique "bibliographie1" se trouvant dans le livre1 implique respectivement la suppression ou la modification de la même référence bibliographique "bibliographie1" se trouvant dans le livre2.

Lien de dépendance exclusif

Si un objet (récepteur) référence un objet (émetteur) à travers un lien de dépendance exclusif, alors l'émetteur ne peut pas être référencé par un autre objet. Seul l'objet (récepteur) est le propriétaire au sens commun du terme de l'objet (émetteur). L'émetteur ne peut pas être valeur d'autres attributs qui sont en mode exclusif ou partagé. Exemple : il n'est pas concevable de considérer un moteur donné comme un composant de deux voitures. Par conséquent, le moteur ne peut être composant que d’une seule voiture. On dit alors que le moteur est lié à la voiture par un lien de dépendance exclusif.

Lien de dépendance existentiel

Le lien de dépendance existentiel met l'accent sur l'idée d'existence d'un objet (récepteur/émetteur) conditionnée par l'existence d'un autre objet (récepteur/émetteur). On peut avoir un lien de dépendance existentiel soit dans le sens émetteur-récepteur (existentiel_e_r) soit dans le sens récepteur-émetteur (existentiel_r_e).

Dans le cas d'un lien existentiel_r_e (resp. existentiel_e_r), la suppression d'un objet récepteur (resp. émetteur) implique la suppression de la valeur d'un de ses attributs émetteur (resp. récepteur). Cette suppression aura lieu même si l'émetteur (resp. récepteur) est référencé par un autre objet, que ce soit en mode partagé ou non partagé.

Exemple : La destruction de la voiture implique la destruction du moteur de la voiture si un lien existentiel_r_e est défini entre la voiture et son moteur.

3.3 LES METHODES

3.3.1 Shood et les fonctions génériques

Les méthodes de SHOOD permettent d'ajouter de la connaissance déclarative à la connaissance procédurale. Une méthode pointe sur un code interprétable et contient des informations relatives à l'utilisation de ce code. Une méthode contient, par exemple, l'information nécessaire pour décider si elle peut être exécutée avec un jeu donné de paramètres effectifs. Elle peut aussi contenir des informations décrivant sa façon d'interagir avec d'autres méthodes.

SHOOD est basé sur l'utilisation de fonctions génériques. Une fonction générique est une fonction dont l'exécution dépend du type des arguments. A une fonction générique, on associe toutes les méthodes permettant de réaliser la même opération conceptuelle. Suivant les arguments fournis en entrée, seules certaines méthodes de la fonction générique sont exécutables.

Pour réaliser cela, SHOOD possède un mécanisme de sélection de méthodes. Dans le cas où plusieurs méthodes sont sélectionnées, un autre mécanisme permet de les faire coopérer [Millasseau92].

3.3.2 Les méthodes sont des classes

Dans SHOOD, les méthodes sont représentées par des classes, appelées classes-méthodes, dont les attributs représentent les paramètres de la méthode. Pour typer les paramètres, nous pouvons donc utiliser tout le système de typage et de contraintes d'attributs existant dans SHOOD. Le nom de la méthode est donné par le nom de la classe.

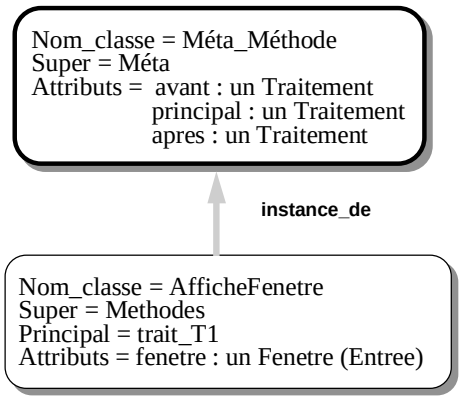
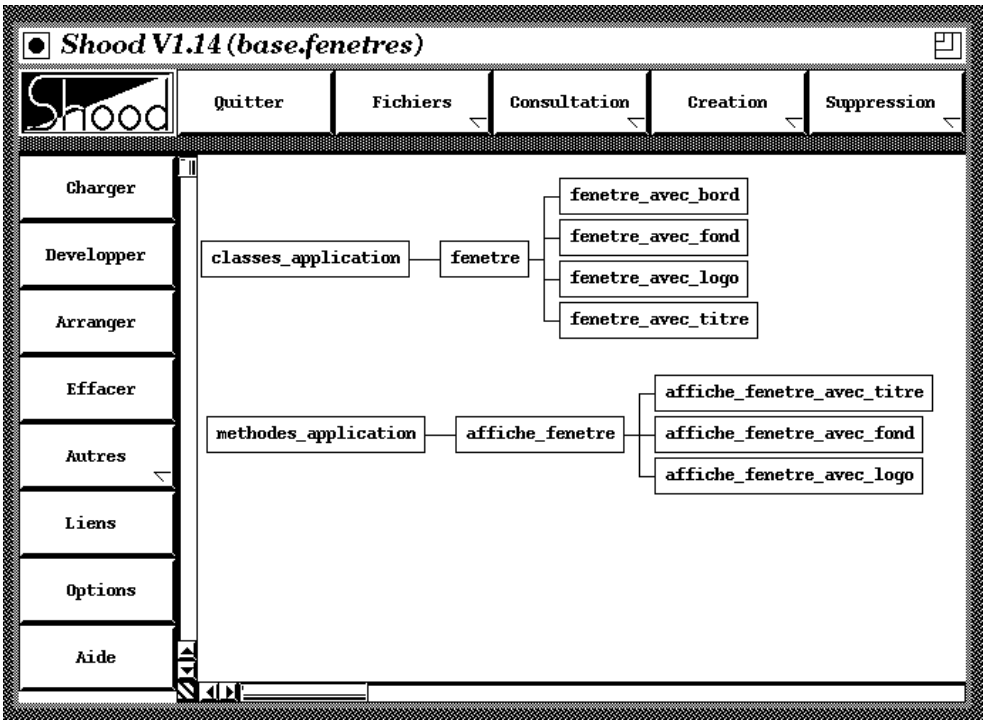


Figure 3-12 : Une méthode est une classe

Les classes-méthodes sont instances d'une méta-classe particulière, *Méta_méthode*, laquelle spécialise *Méta* en ajoutant les attributs supplémentaires *avant*, *principal* et *après*. Ces

attributs représentent les trois traitements d'une méthode et ont la sémantique de prologue, corps et épilogue de la méthode. Un traitement contient notamment un algorithme. L'exécution simple d'une méthode consistera à exécuter l'algorithme de chacun des trois traitements.



Ecran 3-1 : Graphe de méthodes

Les méthodes étant des classes, nous pouvons alors les organiser dans un graphe que nous appellerons graphe de méthodes et dont la racine est la classe *Méthodes*. Une fonction générique est alors représentée par un sous-graphe du graphe de méthodes (Ecran 3-1).

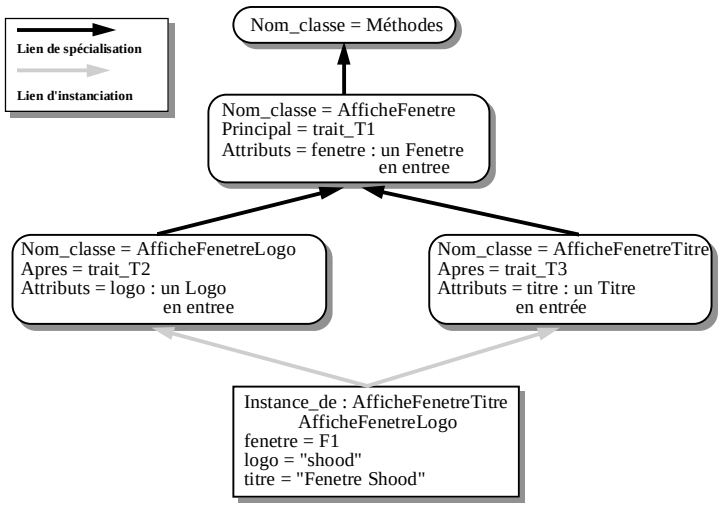


Figure 3-13 : Fonction générique

L'exécution d'une méthode est représentée par la création d'une instance de cette méthode. Cette instance possède d'abord la valeur des arguments donnés en entrée, puis est complétée au fur et à mesure de l'exécution par les arguments de sortie calculés dans les algorithmes. Dans le cas où plusieurs méthodes doivent être exécutées, l'exécution est représentée par une multi-instance des classe-méthodes.

Remarque :

Comme une méthode dans SHOOD est modélisée par une classe, elle doit respecter les mêmes invariants que celles des classes. Une méthode définit, en plus, des traitements qui sont hérités dynamiquement.

L'invocation des méthodes se fait par un appel à une fonction générique. Une fonction générique peut être sollicitée dans différentes situations. Elle peut être appelée à partir d'une inférence, d'une contrainte, de l'interface (graphique ou fonctionnelle). Le système recherche alors quelles méthodes de la fonction générique doivent être exécutées ; c'est la sélection des méthodes.

3.3.3 La sélection des méthodes

Le mécanisme de sélection de méthodes consiste à rechercher toutes les méthodes attachées à la fonction générique et qui peuvent s'appliquer avec le jeu de paramètres donné en entrée. En dernier lieu, on ne conserve que les méthodes les plus spécialisées.

Déterminer l'ensemble des méthodes applicables, c'est rechercher les méthodes susceptibles d'accepter l'instance dont les attributs sont valués avec les paramètres fournis en entrée. Le système fait appel pour cela au mécanisme de classification d'instances [Liotard93] [Bounaas94b] (cf. Chapitre 5).

Illustrons nos propos par l'exemple de la Figure 3-13. Supposons que l'on ait la méthode *AfficheFenetre* et ses deux sous-méthodes *AfficheFenetreLogo* et *AfficheFenetreTitre* qui permettent respectivement d'afficher une fenêtre simple, une fenêtre avec un logo et une fenêtre avec un titre. L'utilisateur veut afficher une fenêtre F1 avec le logo *Shood* et le titre *Fenêtre Shood.* Il appelle alors la fonction générique *AfficheFenetre* avec ces trois paramètres. Le mécanisme de classification trouve comme première méthode d'appartenance la méthode *AfficheFenetre.* Il continue sa recherche dans le graphe : la définition de la méthode *AfficheFenetreLogo* correspond aussi aux arguments (de par la présence d'un paramètre *logo*). Il en est de même pour la méthode *AfficheFenetreTitre.*

Nous avons vu que le modèle SHOOD permet la multi-instanciation : une instance donnée peut être rattachée à plusieurs classes (cf. § 3.1). La classification prend en compte la multi-

instanciation et propose alors un ensemble de classes auxquelles une instance peut être rattachée. Dans le cas des méthodes, le mécanisme retourne l'ensemble des méthodes les plus spécialisées, en l'occurrence *AffichеFenetreLogo* et *AffichеFenetreTitre*. Il va falloir les faire coopérer.

3.3.4 La coopération des méthodes

Cette coopération consiste à faire l'inventaire des traitements et à les ordonner. Pour chaque traitement, l'utilisateur peut spécifier si le traitement doit être exécuté avant un autre, être éliminé si un autre existe, etc.

Il se peut que pour une méthode, un ou plusieurs traitements (avant, principal ou après) ne soient pas définis. Dans ce cas, ils sont hérités des super-méthodes. C'est le cas du traitement principal de *AffichеFenetreLogo* et *AffichеFenetreTitre*.

L'exécution se compose de trois étapes (avant, principal et après), permettant respectivement de construire la liste ordonnée des traitements avant, principaux et après. Chaque étape fait l'inventaire des traitements correspondants (obtenus par héritage dynamique s'ils ne sont pas définis) et les ordonne en analysant les contrôles de précédence. On obtient alors une liste ordonnée (parfois vide) de traitements avant, un seul traitement principal et une liste ordonnée (parfois vide) de traitements après.

Dans l'exemple de la Figure 3-13, les traitements seront exécutés dans l'ordre T1, T2 et T3 ou bien T1, T3 et T2 (aucune priorité n'est spécifié entre T2 et T3).

Nous avons vu qu'au cours de l'exécution, une multi-instance des méthodes sélectionnées est créée. Certains des attributs de cette multi-instance sont valués avec les paramètres effectifs fournis lors de l'appel. Au cours de cette exécution, des arguments en sortie peuvent être calculés, et leurs valeurs sont immédiatement reportées dans l'instance représentant l'exécution. Ainsi, en fin d'exécution de tous les traitements, l'instance doit contenir une valeur pour tous les paramètres en sortie.

3.4 LES LIENS INTER-CLASSES

3.4.1 La spécialisation

L'ensemble des classes forme un graphe de spécialisation où chaque sous-classe hérite de l'ensemble des propriétés des super-classes. L'héritage est multiple et complet, c'est-à-dire qu'une sous-classe hérite de tous les attributs de toutes ses super-classes. Un attribut hérite de l'ensemble des contraintes et des inférences définies dans les super-classes. La spécialisation est obtenue par enrichissement de nouveaux attributs ou par redéfinition des attributs hérités.

L'attribut peut être redéfini par restriction de type, par ajout et par modification des contraintes et/ou des inférences.

L'héritage des attributs doit respecter l'invariant suivant, qui assure la cohérence de la hiérarchie des classes.

> **I.S.3 : Invariant de l'héritage des attributs**
> Une classe hérite de tous les attributs définis dans ses super-classes. De même, elle hérite des attributs pré-déclarés définis dans les super-classes et non redéfinis dans la classe.

Si une classe hérite de deux attributs de même nom, deux cas se posent : soit les classes d'origine des deux attributs sont différentes (noms complets différents) et on hérite des deux attributs ; soit la classe d'origine est la même, dans ce cas on hérite d'un des attributs.

Mais un problème se pose si les attributs ont des types différents, d'où l'invariant suivant sur la restriction des types.

> **I.S.4 : Invariant de restriction de types**
> Le type d'un attribut dans une sous-classe doit être un sous-type des définitions dans ses super-classes. Lorsqu'une classe hérite d'un attribut défini dans deux super-classes mais avec des types différents, le type résultat est l'intersection des types des attributs des super-classes.

3.4.2 La disjonction

La disjonction modélise une incompatibilité explicite de concepts. Elle est nécessaire lorsque les classes modélisent des domaines mal définis pour lesquels on ne dispose pas de propriétés discriminantes.

Cette incompatibilité de concepts peut être explicitement modélisée par un lien de disjonction : deux classes disjointes ont une intersection vide. C'est par exemple le cas des classes des avions de lignes et des avions d'affaire (Figure 3-14).

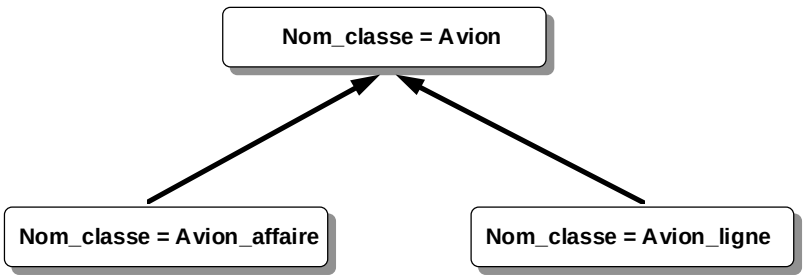


Figure 3-14 : relation de disjonction

Comme les liens de disjonction font partie des graphes de spécialisation et d'instanciation, les invariants I.S.1 et I.I.1 concernant les graphes sont complétés par les deux invariants suivants :

I.S.5 : Invariant du lien de disjonction
Les liens de disjonction sont des arcs bidirectionnels entre les nœuds du graphe de spécialisation. La sémantique du lien de disjonction est la suivante :
Les sous-graphes à racine unique, dont les racines sont deux nœuds connectés par le lien "disjointe_a", ne sont pas connectés. En d'autres termes, si deux classes sont disjointes alors leurs sous-classes le sont aussi.

I.I.3 : Invariant du lien de disjonction
Deux nœuds représentant des classes disjointes ne peuvent pas être référencés par un même objet. En d'autres termes, deux classes disjointes n'ont aucune instance en commun.

La notion de disjonction sert à prévenir des opérations inopportunes entre concepts différents. Elle interdit par exemple l'existence d'instances communes ou de spécialisations communes à deux classes dont l'incompatibilité n'a pu être explicitée par des propriétés discriminantes.

3.5 LE LIEN D'INSTANCIATION

L'appartenance d'une instance à sa classe est modélisée par le lien d'instanciation. Instancier une classe veut dire en particulier valuer les attributs de cette classe. Un lien d'instanciation ne modélise qu'un point de vue de l'objet. La multi-instanciation consiste à rattacher une instance à plusieurs classes. Ce concept permet d'éviter la prolifération des classes creuses. On imagine le nombre de combinaisons de classes qu'il faudrait pour représenter tous les points de vue possibles et imaginables ! La multi-instanciation permet de représenter différents points de vue d'un même objet.

3.5.1 L'instanciation

L'instanciation dans SHOOD permet le rattachement d'une instance à une classe dont elle n'est pas un moulage parfait. Ceci est en partie pris en compte par les concepts de contraintes fortes /faibles et d'attributs obligatoires /facultatifs (§0).

Définitions :

- Une instance qui respecte l'ensemble des contraintes (faibles et fortes) associées aux attributs valués, est dite *cohérente* [Escamilla90]. Une instance est *fortement cohérente* si toutes ses contraintes fortes sont respectées.

• De même, une instance valuant tous les attributs de sa classe est dite *complète.* Une instance est *minimale* si tous ses attributs obligatoires sont valués.

L'invariant I.I.4 fixe les conditions minimales d'appartenance d'une instance I à une classe C. Si ces conditions sont respectées, la classe C est dite *possible* pour I.

> **I.I.4 : Invariant d'instanciation**
> Une classe C est possible pour une instance I ssi :
> • I est fortement cohérente dans C,
> • I est minimale dans C,
> • tous les attributs I ont été inférés dans le graphe de C [3].

La détermination du statut d'une classe (possible ou impossible) pour une instance est faite en exécutant les inférences associées aux attributs. Elles permettent l'obtention de valeurs d'attributs et donc, en particulier, la valuation d'attributs obligatoires. Les résultats des inférences sont ensuite vérifiés par l'évaluation des contraintes. Si une contrainte n'est pas évaluable (c'est par exemple le cas où un de ses arguments n'est pas valué) SHOOD pratique la politique du "bénéfice du doute" : la contrainte est considérée comme respectée.

3.5.2 La multi-instanciation

La multi-instanciation consiste à rattacher une instance à plusieurs classes possibles et qui ne sont pas directement ou indirectement des spécialisations les unes des autres [Nguyen92a]. Elle autorise donc le rattachement d'une instance à des classes comportant des sémantiques différentes, chacune d'elles modélisant un point de vue de l'objet. Ainsi, des utilisateurs ayant des centres d'intérêt différents peuvent voir le même objet selon le point de vue qui les intéresse. Le chercheur *Dupond* peut être aussi vu comme le père de l'enfant *Dupond-fils* (Figure 3-15).

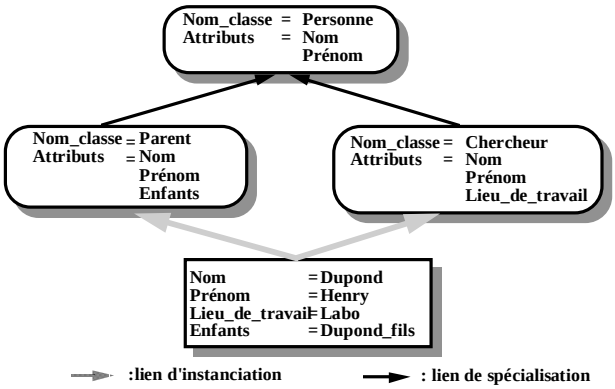


Figure 3-15 : exemple de multi-instanciation

[3] Le graphe de C est constitué de C et de toutes ses super-classes directes ou indirectes.

L'invariant I.I.4 doit être étendu pour prendre en compte les conditions minimales d'appartenance d'une instance I à un ensemble E de classes.

Par extension, nous appellerons graphe de E, le graphe constitué de toutes les classes de E et de toutes leurs super-classes directes ou indirectes.

I.I.5 : Invariant de multi-instanciation
Un ensemble E de classes est possible pour une instance I si et seulement si :
• I est fortement cohérente dans toutes les classes de E,
• I est minimale dans toutes les classes de E,
• tous les attributs de I ont été inférés dans le graphe de E,
• il n'existe pas de disjonction explicite dans le graphe de E.

3.6 LA META-CIRCULARITE

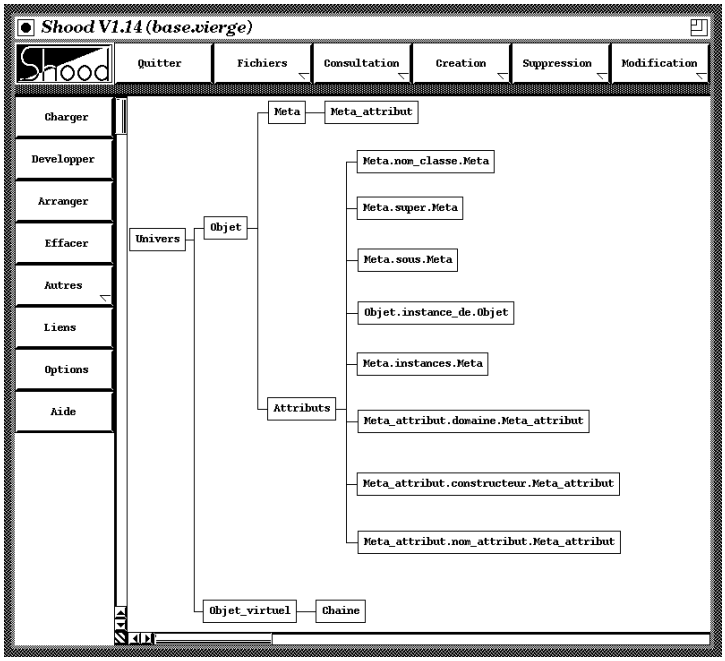
Le modèle SHOOD a été défini de façon à gérer des connaissances dynamiques et évolutives. Il offre la possibilité d'ajouter, de détruire ou de modifier à tout moment les définitions des classes, à savoir leur nom, leurs descripteurs d'attributs et les relations qu'elles entretiennent entre elles (cf. chapitre 4). Pour cela, le modèle suit l'approche "tout objet" comme ObjVlisp [Cointe87] ; un objet peut être une connaissance élémentaire ("Dupond"), une connaissance générique ("toute personne a une tête"), une méthode ("la différence entre deux entiers"), une inférence ("l'âge d'une personne est calculé par la différence entre l'année courante et l'année de sa naissance").

Le modèle est réflexif : il possède la propriété de se décrire lui-même [Escamilla93]. Tous les concepts tels que les classes, les attributs, les méthodes peuvent être décrits à partir d'eux-mêmes. De même, **le modèle est méta-circulaire : il s'implante avec ses propres concepts**. Dans cette optique, un niveau méta a été introduit afin de spécifier le comportement des différents objets (Ecran 3-5). Au niveau du modèle, aucune différence n'est faite entre la classe *Méta* et un autre objet ; c'est dans notre esprit que nous faisons la différence afin de mieux comprendre le fonctionnement du système.

Les informations absolument nécessaires pour la génération automatique du système concernent la définition d'une classe au niveau système comme les attributs *Nom_classe*, *Super*, *Sous*, *Attributs*, *Instance_de* et *Instances*. Pour définir et typer les attributs précédents, il faut des informations comme *Nom_attribut*, *Domaine* et *constructeur*.

Pour élaborer le graphe de spécialisation, ce sont les attributs *Super* et *Sous* qui sont utilisés. Les attributs *Instance_de* et *Instances* permettent d'intégrer les classes dans le graphe d'instanciation.

Ce minimum de connaissances doit exister pour que le modèle "fonctionne tout seul". Il existe un graphe d'héritage et un graphe d'instanciation minimums nécessaires pour décrire tous les autres concepts (Ecran 3-2).



Ecran 3-2 : Le graphe d'héritage minimum

Le graphe représente non seulement le graphe d'héritage, mais aussi une sorte de rangement des classes par rapport à leurs fonctionnalités. Ainsi, la classe *Univers* est créée pour servir de racine à deux sous-graphes indépendants : *Objet* et *Objet_virtuel*.

Un concept peut être décrit de deux manières : par extension ou par intention. Par extension, on fait une énumération de tous les individus qui sont représentés par le concept. Par intention, on fait une énumération des propriétés communes à tous les individus représentant le concept.

La classe *Objet* est la racine du graphe de spécialisation de toute classe instanciable. Une classe instanciable est définie en intention : pour que *Dupond* appartienne à la classe *Personne*, il doit être explicitement créé dans cette classe. La classe *Objet* admet des sous-classes prédéfinies telles que *Méta* et *Attributs* qui contiennent le comportement minimum des objets à créer.

La classe *Objet-virtuel* est racine du graphe de spécialisation des classes non instanciables. Ces dernières concernent les classes dont l'ensemble des instances est défini en extension, comme les classes des entiers et des réels, et qui n'auront pas d'instances.

La classe *Méta* et ses sous-classes vont servir à créer toutes les autres classes d'objets : la classe Méta est racine du graphe d'instanciation.

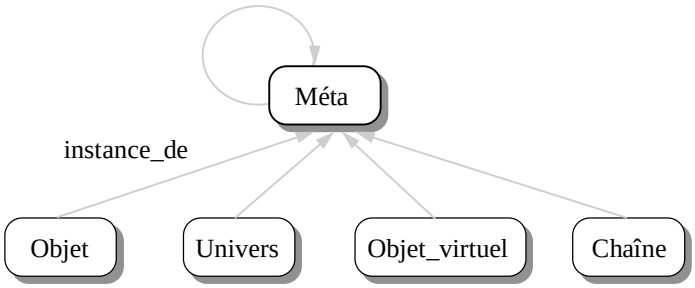
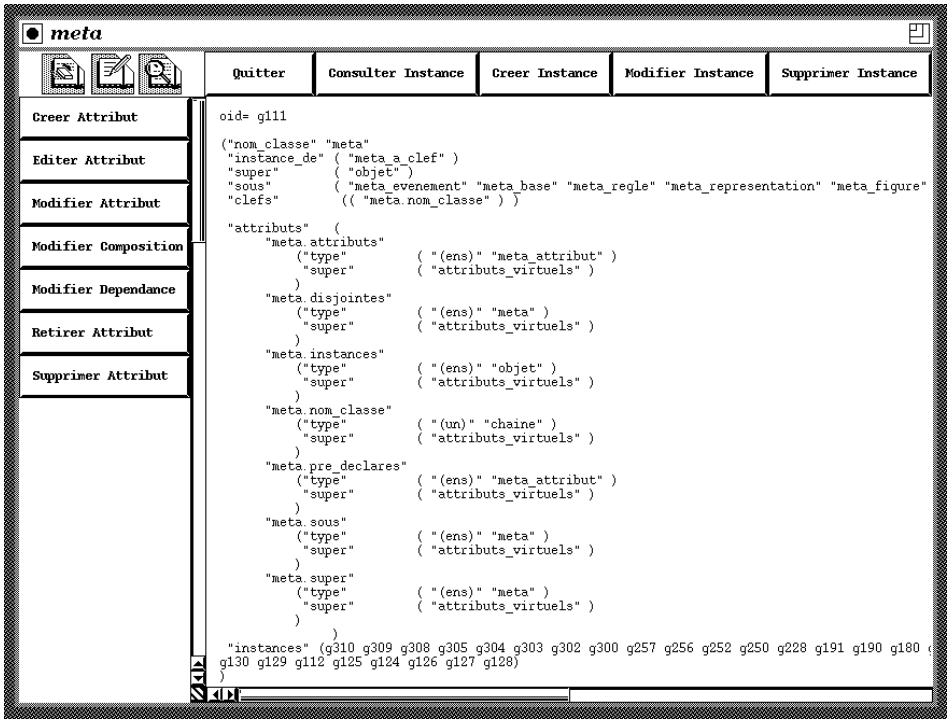


Figure 3-16 : Le graphe d'instanciation minimum

Pour bien comprendre le mécanisme d'instanciation, nous allons détailler la création d'une classe, c'est-à-dire l'opération d'instanciation d'une méta-classe.



Ecran 3-3 : Schéma de la classe Méta

Lors de la création d'une classe, une instance est créée dans la classe *Méta* et les attributs de *Méta* sont valués (Ecran 3-3). La valuation de l'attribut *Nom_classe* effectue réellement la création de la classe en lui affectant un nom unique. Les valeurs affectées aux attributs *Super* et *Sous* permettent d'intégrer la classe dans le graphe d'héritage. La valuation de l'attribut *Instance_de* permet d'intégrer la classe dans le graphe d'instanciation. L'attribut *Instances* sera valué au moment de la création d'une instance de la nouvelle classe. Ainsi, tout concept peut être créé par instanciation d'une classe faisant partie du graphe de spécialisation de *Méta*.

Parmi toutes les instances d'une classe, on cherche parfois à trouver une instance particulière. On peut pour cela utiliser l'OID, qui est l'identificateur système d'un objet (Object IDentifier). Il est unique et est affecté à l'objet lors de sa création par le système.

Mais on peut tout aussi bien retrouver l'instance par des valeurs uniques pour un ensemble d'attributs, si l'instance en possède. Cet ensemble d'attributs constitue alors une clef permettant d'identifier d'une manière unique l'instance.

Si on veut que la classe *Personne* ait une clef constituée, par exemple, des attributs *Nom* et *Prénom*, il faut que cette classe soit une instance d'une méta-classe qui définisse des classes à clefs, c'est-à-dire des classes dont les instances seront identifiées par des valeurs de clefs. Pour cela, on crée une méta-classe *Méta_a_clef*, sous-classe de *Méta*, qui admet un attribut supplémentaire *Clefs* défini par une liste d'attributs. Lors de la création de la classe *Personne*, tous les attributs hérités de *Méta* seront valués, et l'attribut *Clefs* spécifique à *Méta_a_clef* sera valué par *Nom* et *Prénom*.

Toute classe (ou méta-classe) est identifiée dans SHOOD par son nom qui est donc unique dans la base. L'attribut *nom-classe* défini dans *Méta* est donc un attribut clef. Pour pouvoir définir *nom_classe* comme étant un attribut clef, il est nécessaire que *Méta* et ses sous-classes soient instances de *Méta_a_clef*. Finalement, c'est *Méta_a_clef* qui est racine du graphe d'instanciation :

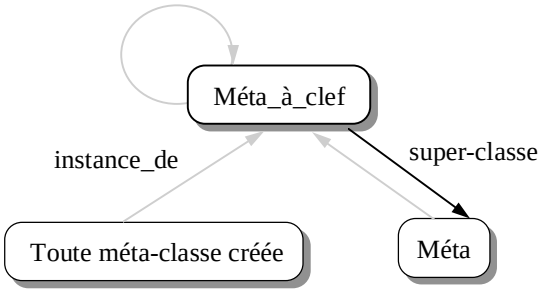


Figure 3-17 : Graphe d'instanciation

Afin de maintenir la cohérence des clefs dans le graphe des classes, il est nécessaire de maintenir les deux invariants suivants :

I.S.7 : Invariant de l'héritage des clefs
Une classe hérite de toutes les clefs de toutes ses super-classes.

I.S.8 : Invariant des restrictions des clefs
Une clef dans une classe doit être composée d'attributs ayant un type défini. Toute nouvelle clef définie dans une classe doit être minimale par rapport aux clefs héritées.

Lorsqu'une classe est instance d'une méta-classe (par exemple *Meta_à_clef*), elle value les attributs de sa méta-classe (par exemple les clefs). Ses sous-classes doivent hériter de ces valeurs (les clefs) d'où l'invariant suivant :

> **I.I.6 : Invariant des méta-classes**
> Une classe doit hériter de toutes les propriétés de ses super-classes. Donc toute classe doit instancier au minimum les mêmes méta-classes que ses super-classes.

3.7 SYNTHESE DU MODELE

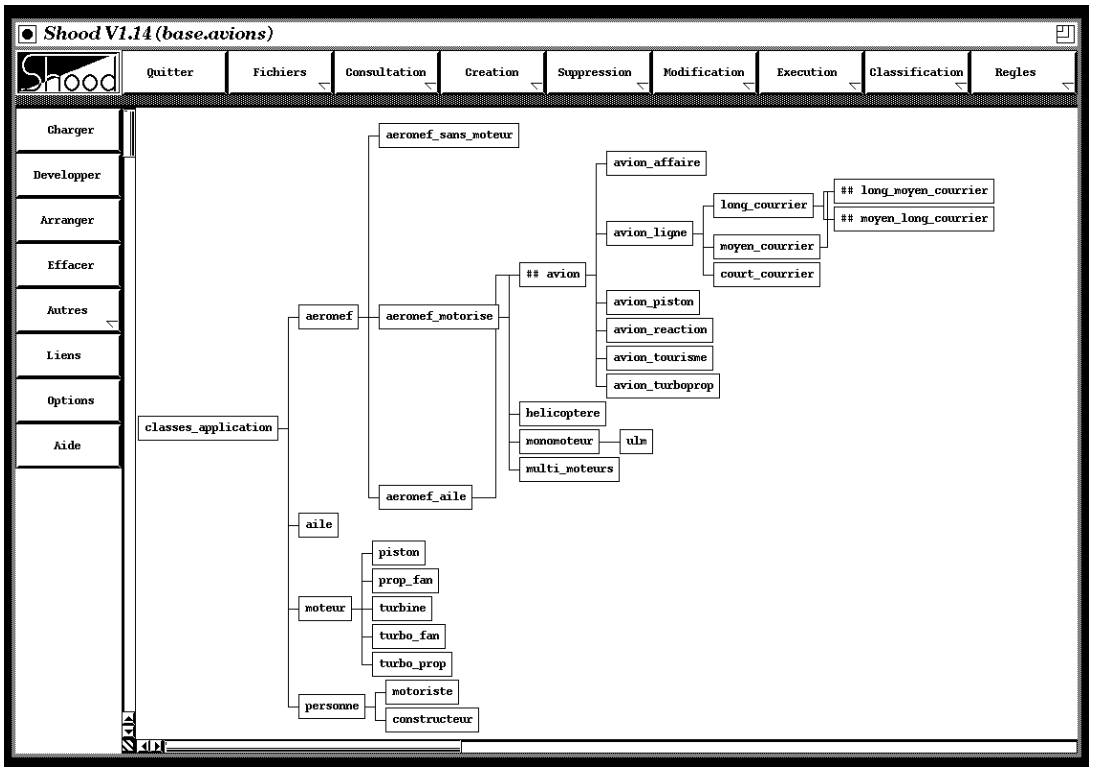
De ce chapitre, il faut retenir que tous les concepts du modèle sont modélisés par des classes qui sont instances de méta-classes et qui génèrent des instances. Une classe décrit la structure des objets par un ensemble d'attributs. Le comportement des objets est représenté par les méthodes, qui ne sont pas encapsulées dans les classes et qui sont exécutables par le mécanisme d'appel de fonctions génériques. Un attribut est défini par un ensemble de descripteurs : un descripteur de type définit le domaine de définition de l'attribut, un descripteur d'inférence permet la valuation de l'attribut sans l'intervention de l'utilisateur, et finalement, un descripteur de contraintes assure la cohérence des valeurs. Il est possible d'enrichir la sémantique de l'attribut par des dépendances (dépendance exclusive, existentielle, etc.).

Les classes sont organisées en un graphe de spécialisation obéissant à un ensemble d'invariants. En parallèle au graphe de spécialisation, les objets sont organisés dans un graphe d'instanciation respectant aussi des invariants.

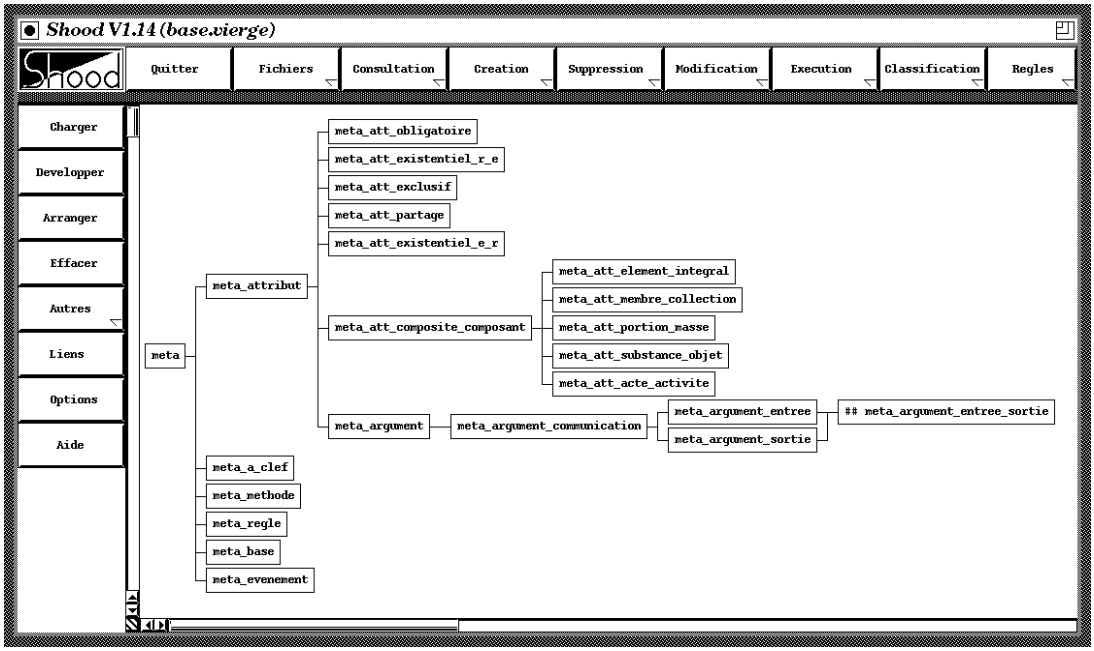
Grâce à la méta-circularité, le système permet une évolution des concepts. Il est possible d'en rajouter de nouveaux (exemple : les clefs) sans remettre en cause ce qui a déjà été généré. Dans le chapitre 7, nous allons voir un autre cas d'évolution de concepts. Nous présentons l'extension du modèle SHOOD par des règles actives. Par cette extension, le système SHOOD devient actif.

Le modèle tel qu'il est présenté ici a fait l'objet du développement d'un prototype écrit en Le_Lisp sur station de travail Unix. L'ensemble des concepts présentés dans ce chapitre peut être manipulé à travers une interface graphique simple (voir Ecran 3-4 et Ecran 3-5).

Dans le chapitre suivant, nous allons présenter les propriétés et l'architecture du système qui va permettre l'évolution des concepts présentés dans ce chapitre. Le chapitre sera principalement consacré à la présentation des opérations de manipulation de ces concepts.



Ecran 3-4 : Un graphe de classes d'une application



Ecran 3-5 : Le niveau méta

4. LE SYSTEME D'EVOLUTION DE SHOOD

De nombreux travaux ont été réalisés pour gérer et supporter l'évolution de schémas [Banerjee87] [Penney87]. La plupart proposent, à travers un ensemble d'invariants et d'opérations de mise à jour, une sémantique particulière d'évolution de schémas. Cette sémantique peut être formalisée par un ensemble de règles définissant une stratégie d'évolution. Une stratégie est souvent dictée par les besoins d'un type d'application ou par les objectifs du système. Il est souhaitable de pouvoir adapter la stratégie d'évolution en fonction du type d'application, tout en offrant une stratégie par défaut. Par exemple, dans certaines applications, la suppression d'une classe possédant des instances peut être refusée en phase d'exploitation, en revanche elle peut être acceptée en phase de conception moyennant la suppression de ses instances. Afin d'aider l'utilisateur, il est aussi souhaitable de lui proposer une bibliothèque de stratégies dans laquelle il choisira celle qui lui convient.

Le système d'évolution que nous présentons dans cette thèse suit cette approche. Dans ce chapitre, nous nous contentons uniquement de présenter l'architecture, les propriétés et le noyau de ce système. Les autres mécanismes composant ce système sont présentés dans les chapitres suivants.

4.1 MOTIVATIONS

Nous désignons par système d'évolution l'ensemble des mécanismes permettant de définir et d'exploiter l'évolution des concepts d'un modèle de données. Le système d'évolution de SHOOD est défini par un ensemble de concepts (opération, règle, stratégie) et de mécanismes sous-jacents (classification, règles actives, héritage) permettant la définition et l'exploitation de ces concepts.

Ce système d'évolution est bien adapté aux systèmes dits "extensibles", où les concepts et leurs évolutions respectives ne sont pas figés [Cointe87]. Ces règles et stratégies d'évolution permettent, en effet, d'étendre le modèle par de nouveaux concepts et d'adapter les opérations de manipulation existantes à cette extension.

Les systèmes où tout est statique et prédéfini n'ont besoin d'exprimer ni des stratégies multiples, ni des règles de manière déclarative. Il n'y a qu'une stratégie d'évolution imposée à

toutes les classes. Cette approche n'offre pas la possibilité au programmeur d'exprimer plusieurs stratégies et d'activer à tout moment la stratégie la plus adéquate pour l'application. Par exemple, dans SHOOD, la suppression d'une classe entraîne le rattachement de ses instances aux super-classes. C'est une sémantique par défaut. Il faudrait pouvoir exprimer, pour certains types de classes ou pour certaines applications, que la suppression d'une classe entraîne la suppression de ses instances.

Une sémantique d'évolution d'un concept est souvent exprimée dans le langage hôte (langage d'implémentation du système), noyée dans le code du système (non accessible à tout programmeur), au mieux dans le code des méthodes, et le programmeur se voit confronté au problème d'extension. En effet, dans un système extensible, l'ajout de nouveaux concepts entraîne souvent, d'une part, la mise en œuvre des opérations de mise à jour permettant de faire évoluer les concepts, et d'autre part, la modification des opérations de mise à jour existantes pour tenir compte de l'évolution des concepts ajoutés. Il serait intéressant de rajouter ces fonctionnalités plus aisément. Par exemple, lors de l'ajout du concept d'événement, il faut rajouter les opérations manipulant ce concept (suppression, modification) ; il faut aussi modifier la sémantique d'évolution d'une méthode (si le nom d'une méthode change, celui de l'événement associé à son exécution change aussi) (cf. Chapitre 7).

Tenant compte des problèmes cités ci-dessus, le système d'évolution doit être caractérisé par les propriétés décrites ci dessous.

Propriétés

Convivialité : Possibilité d'exprimer les règles d'évolution de classes et d'instances de manière déclarative. Le système d'évolution doit permettre, à travers la syntaxe des règles d'évolution, d'exprimer des règles d'évolution de schémas, des contraintes d'intégrité, des calculs inférentiels et tout autre comportement entraînant l'évolution des informations manipulées par le système.

Extensibilité : On doit pouvoir rajouter de nouvelles fonctionnalités (règles ou opérations d'évolution) assez facilement sans altérer les fonctionnalités existantes. Le système ne doit pas être fermé, l'ensemble des opérations doit pouvoir être étendu à tout moment. De même, il doit être possible de redéfinir la sémantique des opérations existantes.

Multi-Stratégie : Possibilité de définir plusieurs stratégies d'évolution d'un concept ; aucune stratégie n'est imposée. En effet, il sera possible d'exprimer plusieurs stratégies, mais une seule sera activée à un instant donné.

Par exemple : lors de la suppression d'une classe, plusieurs possibilités se présentent : soit supprimer les instances, soit les rattacher aux super-classes de la classe supprimée, soit refuser tout simplement la suppression de la classe. Ces trois possibilités représentent trois stratégies d'évolution d'une classe ; chacune d'elles pourra être utilisée à des moments distincts.

Complétude : L'ensemble des fonctionnalités doit permettre de faire évoluer tous les concepts du modèle de données, quelle que soit la portée de la modification. L'ensemble des opérations accessibles ne doit pas être restreint, par exemple, uniquement aux opérations de mise à jour. Les opérations peuvent être des méthodes de l'application (par exemple, pour l'expression des contraintes). De plus, avec la propriété d'extensibilité, ces fonctionnalités peuvent toujours être complétées ; le système n'est pas figé.

Cohérence : Cette évolution ne doit pas introduire d'incohérence dans la base de connaissances. En effet, à l'exécution d'une opération de manipulation, le système doit vérifier le respect des invariants du modèle et éventuellement réaliser les propagations nécessaires pour maintenir la cohérence des connaissances.

Dans ce système, nous ne prenons en compte que les cohérences structurelles, les cohérences comportementales constituant à elles seules un sujet de recherche. Par exemple, lors de la suppression d'un attribut, le code des méthodes utilisant cet attribut (exemple : une projection) n'est jamais remis en cause alors qu'il devrait l'être.

Uniformité : Toute évolution doit s'écrire de manière uniforme pour les différents concepts du modèle (classe, métaclasse, instance). En effet, dans ce système, les règles d'évolution (cf. Chapitre 8) des classes et des instances ne sont pas séparées car c'est la sémantique des opérations qui fait la différence.

Architecture

Pour exprimer et mettre en œuvre l'évolution dans le système SHOOD, celui-ci dispose de plusieurs moyens complémentaires qui peuvent être des moyens de base, tels que les opérations de manipulation des classes, ou des moyens plus complexes, tels qu'un mécanisme de règles actives [Bounaas94a] [Bergues94], un mécanisme de classification d'instances [Bounaas94b], ou tout simplement les règles et les stratégies d'évolution définies dans le chapitre 8. Afin de mieux comprendre les liens entre ces différents mécanismes, nous proposons l'architecture suivante pour le système d'évolution :

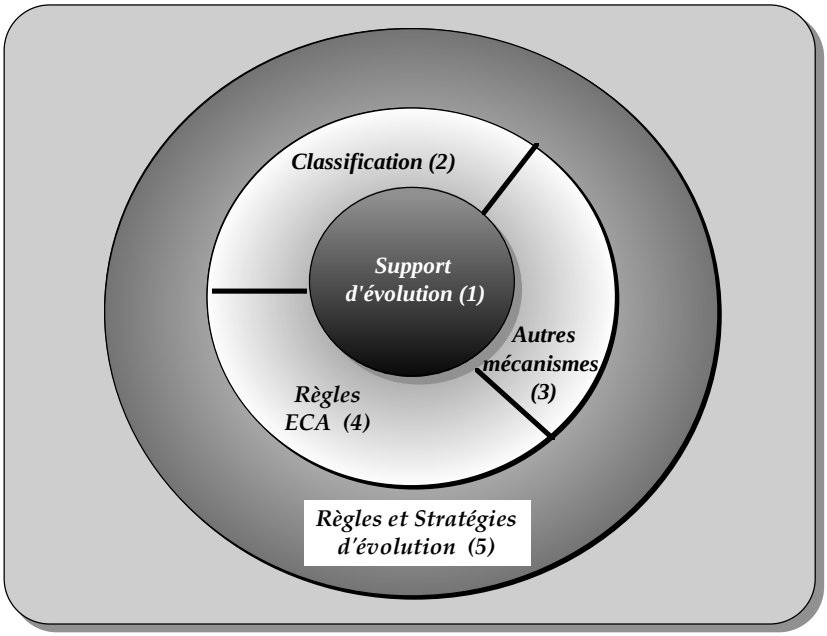


Figure 4-1 : Architecture du système d'évolution

- Le support d'évolution (1) est constitué de l'ensemble des opérations manipulant les concepts de base du modèle SHOOD (cf. 4.2). Ces opérations sont munies d'une sémantique minimale par défaut. Par exemple, l'opération de suppression d'une classe a pour rôle de supprimer les attributs propres, de retirer ses instances (rattachées aux super-classes) et de supprimer physiquement cette classe. D'autres contrôles sont réalisés par des règles d'évolution (voir (5)).

- Au dessus de ce support d'évolution, des mécanismes ont été construits (3) tels que le mécanisme de classification d'instances (2) [Liotard93] et celui de génération de structures significatives [Favier90]. Ces mécanismes sont "câblés" et ne peuvent être modifiés ou étendus facilement.

- Au dessus du support d'évolution, nous avons construit également un mécanisme de règles ECA (4) (cf. Chapitre 7) qui permettra la mise en œuvre du mécanisme de règles et de stratégies d'évolution, et qui offre, de plus, la possibilité d'exprimer des sémantiques d'évolution de concepts. En effet, les règles actives constituent elles-mêmes un outil permettant l'expression et la mise en œuvre des contraintes d'intégrité [Bouaziz95] et des règles d'évolution de schéma [Ahmed-Nacer94] [Bounaas95c]. De plus, nous verrons dans le chapitre 7 une utilisation des règles ECA pour exprimer l'évolution de schémas ou d'instances. Bien que ces règles permettent d'utiliser toute la puissance du modèle actif (lien d'inhibition, désactivation des règles etc.), les règles d'évolution permettent une expression plus abstraite et offrent la possibilité d'utiliser les stratégies d'évolution.

- Le support d'évolution peut être étendu par un ensemble de stratégies (5) où une stratégie est composée d'un ensemble de règles d'évolution (cf. Chapitre 8) [Bounaas95d]. Par exemple, pour la suppression d'une classe, on définit une règle d'évolution qui interdit la suppression si cette classe est utilisée par un domaine d'attribut. Cette règle pourra être redéfinie ou désactivée.

  Il sera possible d'élaborer une bibliothèque de stratégies contenant les sémantiques usuelles des opérations de manipulation ; elle permettra à l'utilisateur de choisir la stratégie la mieux adaptée à son application. L'utilisateur pourra redéfinir la stratégie choisie, en ajoutant ou en redéfinissant les règles d'évolution. Il sera libre de choisir la manière dont ses connaissances devront évoluer. Cette redéfinition de stratégies permet une meilleure réutilisation des règles d'évolution. De plus, toute stratégie définie par l'utilisateur peut être elle-même redéfinie.

Dans la suite, nous allons décrire le support d'évolution qui représente la stratégie par défaut du système d'évolution.

4.2 LE SUPPORT D'EVOLUTION

Le support d'évolution constitue la couche basse du système d'évolution. Il est composé d'un ensemble d'opérations de manipulation de classes et d'instances. Cet ensemble d'opérations doit respecter les invariants du graphe de spécialisation et du graphe d'instanciation. Afin de respecter ces invariants, plusieurs solutions peuvent apparaître lors d'une évolution ; des principes permettent d'en choisir une. L'ensemble de ces principes constitue une stratégie d'évolution par défaut. L'utilisateur pourra étendre le support d'évolution par de nouvelles stratégies. Sur ce support d'évolution, un mécanisme de classification d'instances a été développé (cf. Chapitre 5), qui a pour but de trouver les classes d'appartenance d'une instance. Ce mécanisme permettra, par exemple, de trouver la nouvelle classe d'appartenance d'une instance lorsque celle-ci, après modification, ne sera plus conforme à sa classe de rattachement [Bounaas94a].

Pour l'évolution de schémas, nous adoptons l'approche par correction (cf. Chapitre 2) où les classes sont automatiquement modifiées. Après la modification d'une classe, ses instances sont immédiatement modifiées (conversion immédiate).

Présentons d'abord l'ensemble des invariants et des principes du modèle, que les opérations du support d'évolution devront respecter. Certains de ces invariants et principes sont tirés de [Escamilla93]. Dans le paragraphe suivant, nous allons récapituler les différents invariants du modèle présentés dans le chapitre 3.

4.2.1 Les invariants du modèle

Un invariant est une condition que l'ensemble des classes et des instances doivent respecter. Ils sont classifiés par rapport au concept du modèle qu'ils traitent.

Nous avons défini huit invariants pour le graphe de spécialisation et six pour le graphe d'instanciation. Parmi les invariants du graphe de spécialisation, l'invariant I.S.1 concerne la définition du graphe, les invariants I.S.2, I.S.3 et I.S.4 assurent l'unicité et l'héritage des attributs, l'invariant I.S.5 concernant le lien de disjonction et finalement les invariants I.S.6 et I.S.7 garantissent l'héritage des clefs.

Pour les invariants du graphe d'instanciation, l'invariant I.I.1 définit la structure du graphe d'instanciation, l'invariant I.I.2 assure l'unicité des noms d'attributs, l'invariant I.I.3 garantit la cohérence du lien de disjonction.

4.2.2 Les principes de propagation

Ces principes représentent la meilleure politique permettant de préserver les invariants lors d'une modification si plusieurs solutions sont possibles. La politique jugée meilleure pour le modèle SHOOD a pour critère principal **la perte minimum d'information**. L'ensemble de ces principes constitue une stratégie d'évolution par défaut, qui peut être remise en cause par l'utilisateur.

Modification d'une classe

Toute modification de propriétés d'une classe est propagée dans les sous-classes, si cette propagation n'entraîne pas de perte d'information. Par exemple, l'élimination d'un attribut ne doit pas être propagée car sa propagation entraîne des pertes d'informations. Pour que l'attribut garde le même nom complet, on le pré-déclare dans sa classe d'origine (Figure 4-2)

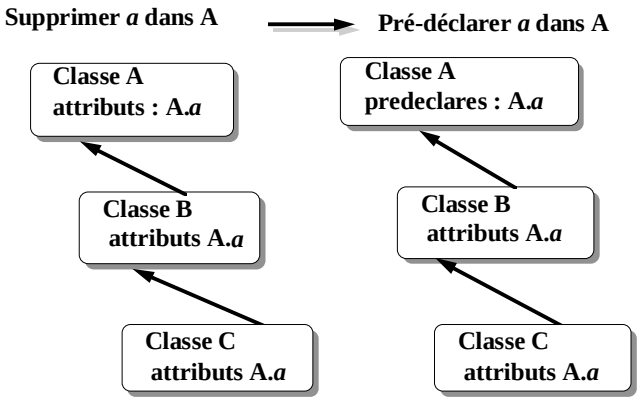


Figure 4-2 : Elimination d'un attribut

Suppression d'une classe

On ne peut pas supprimer une classe si elle est la classe d'origine d'un de ses attributs, car les noms complets des attributs n'auront pas de sens dans les sous-classes. Une solution est de changer la classe d'origine de l'attribut, en pré-déclarant l'attribut dans une super-classe. Si la classe possède plusieurs super-classes, L'attribut est pré-déclaré dans une super-classe commune (Figure 4-3).

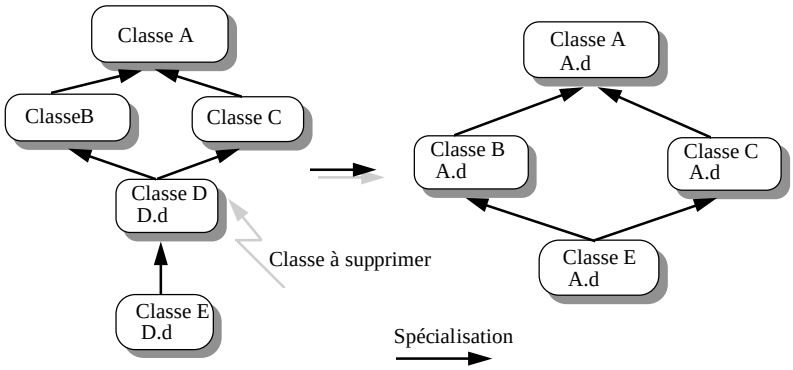


Figure 4-3 : Suppression d'une classe

De même, on ne peut pas supprimer les classes systèmes telles que (Objet, Méta, Méta_attribut, ...).

Suppression d'une méthode

La suppression d'une méthode correspond à la suppression d'une classe. Et si la méthode à supprimer définit des traitements, alors la suppression de cette méthode ne doit pas entraîner la suppression de ces traitements. Ces traitements doivent être hérités dans les sous-méthodes directes qui ne redéfinissent pas les mêmes traitements.

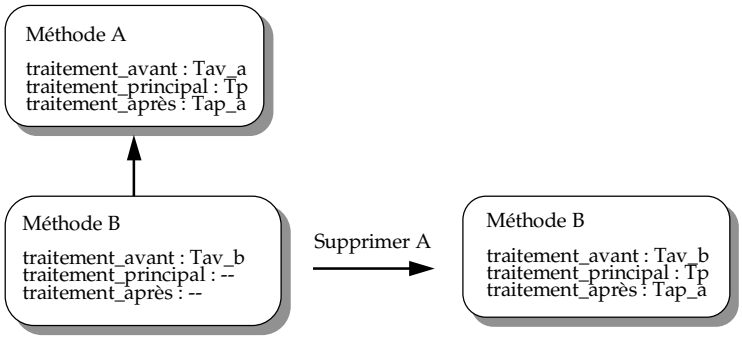


Figure 4-4 : Suppression d'une méthode

La suppression de la méthode *A* entraîne la suppression du traitement *Tav_a* et l'héritage de *Tap_a* dans la sous-méthode *B*, car *Tav_a* était redéfinie et *B* héritait dynamiquement de *Tap_a* avant la suppression de *A*.

Suppression d'un lien de super-classe

Si un lien de super-classe est éliminé et qu'il est le seul lien par lequel un attribut est hérité, l'élimination de ce lien entraîne l'élimination de cet attribut dans les sous-classes de la classe d'où on a retiré le lien. Ce principe surcharge le principe de propagation des modifications.

Une autre solution consisterait à renommer l'attribut (changement de classe d'origine) hérité par ce lien. Mais cette solution peut induire une partition des attributs dans la sous-classe. Analysons l'exemple de la Figure 4-5 : la classe C hérite de l'attribut A.a défini dans A. On veut supprimer le lien entre la classe A et C.

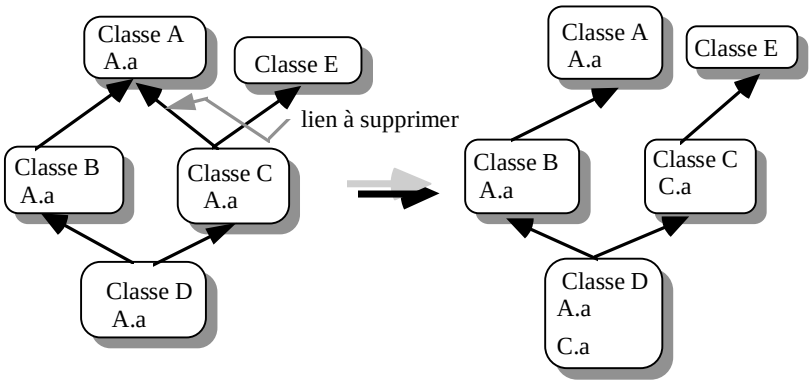


Figure 4-5 : Suppression d'un lien de super-classe

En supprimant ce lien on renomme l'attribut A.a dans la classe C par C.a, et ce dernier est hérité dans les sous classes. Dans la classe D, on hérite de l'attribut A.a d'une autre super-classe (B), ce qui génère la partition de l'attribut A.a en deux attributs dans la classe D (A.a et C.a).

Conclusion

Ces différents principes constituent une stratégie d'évolution par défaut ; avec la notion de stratégie, l'utilisateur pourra toujours définir sa propre stratégie qui, bien sûr, respectera les invariants (codés dans les opérations) mais peut mettre en oeuvre d'autres principes particuliers.

4.2.3 Taxonomie des opérations

Dans ce paragraphe, nous présentons l'ensemble des opérations possibles au niveau du support d'évolution. Cet ensemble d'opérations est composé des opérations manipulant les classes et celles manipulant les instances. Comme une méthode est modélisée par une classe, les opérations sur les classes sont autorisées sur les méthodes, mais la sémantique de ces

opérations peut être différente. Il existe des opérations supplémentaires sur les méthodes, telles que les opérations concernant les traitements (cf. Chapitre 3).

(1) Opérations sur la définition d'une classe
    (1.1) Ajout d'un attribut
    (1.2) Retrait d'un attribut
    (1.3) Modification de la définition d'un attribut
        (1.3.1) Modification du type
        (1.3.2) Ajout et suppression d'une inférence
        (1.3.3) Ajout et suppression d'une contrainte
        (1.3.4) Modification du nom de l'attribut
        (1.3.5) Modification de la classe d'origine
        (1.3.6) Modification de la sémantique de l'attribut
    (1.4) Ajout et suppression d'un attribut pré-déclaré
    (1.5) Ajout et suppression d'une clef
(2) Opérations sur les classes
    (2.1) Création d'une classe
    (2.2) Suppression d'une classe
    (2.3) Modification du nom d'une classe
(3) Opérations sur les liens inter-classes
    (3.1) Ajout d'un lien de spécialisation
    (3.2) Suppression d'un lien de spécialisation
    (3.3) Ajout et suppression d'un lien de disjonction
(4) Opérations sur les méthodes
    mêmes opérations que (1), (2) et (3)
    (4.1) Ajout, modification, suppression des traitements
(5) Opérations sur les instances
    (5.1) Création d'une instance
    (5.2) Modification d'une instance
    (5.3) Suppression d'une valeur d'attribut
    (5.4) Suppression d'une instance
(6) Opérations sur les liens d'instanciation
    (6.1) Ajout d'un lien d'instanciation
    (6.2) Suppression d'un lien d'instanciation
    (6.3) Retrait d'un lien d'instanciation

4.2.4 Sémantique des opérations

Dans la suite, nous commentons certaines opérations qui ne sont pas triviales.

4.2.4.1 Opérations sur les classes

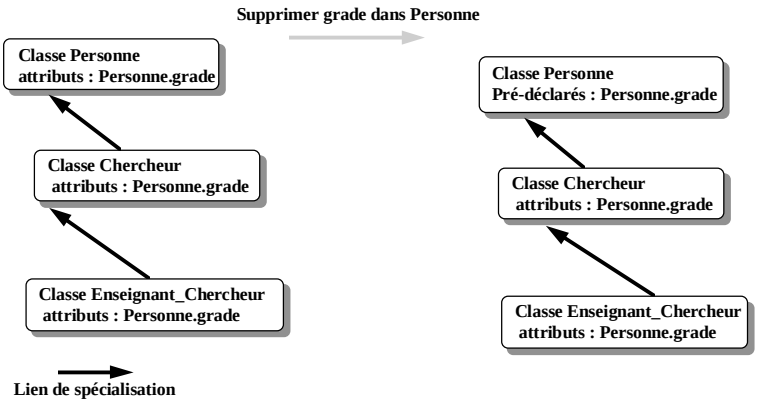
(1) Opérations sur la définition d'une classe

Ces opérations concernent essentiellement la manipulation des attributs d'une classe :

(1.2) Retrait d'un attribut

Le retrait d'un attribut consiste à supprimer la définition de l'attribut dans une classe. D'après le principe de modification d'une classe, le retrait n'est pas propagé dans les sous-classes. Si le retrait est dans la classe d'origine ou dans la première définition de l'attribut, alors l'attribut est pré-déclaré. La définition de l'attribut est supprimée et l'attribut hérite de l'intersection des définitions des attributs définis dans les super-classes.

Un système d'aide est fournie à l'usager pour éliminer l'attribut dans tout le sous-graphe. Lorsqu'on retire un attribut, les instances de la classe où on retire cet attribut perdent éventuellement la valeur de l'attribut retiré.



Pour supprimer "grade" dans la classe *Personne,* tout en gardant le même nom complet dans les sous classes, il suffit de le prédéclarer dans la classe *Personne*.

Figure 4-6 : Suppression d'un attribut

(1.3) Modification d'un attribut

La modification d'un attribut se traduit par la modification de ses descripteurs (domaine, inférence, contrainte). De même, on peut modifier la définition d'un attribut en rajoutant ou en retirant des attributs d'attributs. Comme un attribut est modélisé par une classe, on réutilise les opérations sur les classes.

Si l'attribut modifié est un attribut hérité, alors cette opération correspond à un affinement de l'attribut ou spécialisation.

La modification de la sémantique de l'attribut correspond à la modification de sa(ses) métaclasse(s) d'instanciation. Par exemple, changer le type de dépendance ou de composition ou rendre obligatoire un attribut facultatif, etc.

(1.3.1) Changer le type d'un attribut

La modification d'un type correspond à la modification du constructeur, du domaine ou des restrictions de domaine. La modification de domaine est autorisée si l'invariant de restriction de type n'est pas violé. Dans le cas où la modification est acceptée, elle est propagée dans les

sous-classes. La modification du constructeur par un constructeur plus spécialisé ne peut se faire que dans la classe d'origine. Cette modification est propagée dans les sous-classes.

On rajoutera par la suite, des règles d'évolution ou des règles actives pour vérifier que le type modifié est plus général que l'ancien type. L'utilisateur pourra aussi définir une règle pour reclassifier les instances si celles-ci ne sont plus conformes à leurs classes (Figure 4-7).

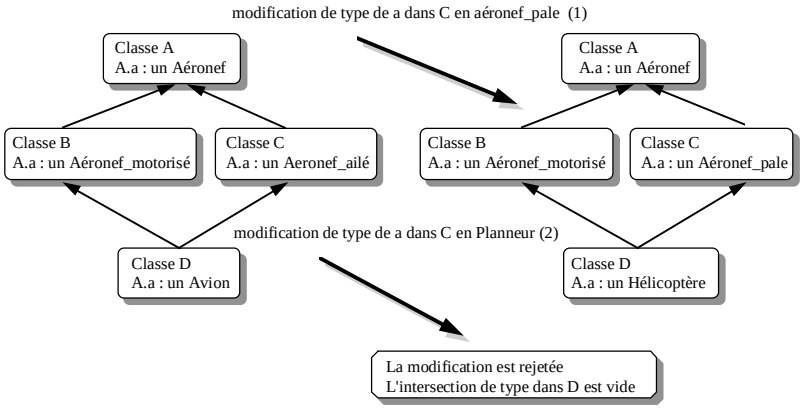


Figure 4-7 : Modification de type

La modification du type de l'attribut *a* est propagée si l'intersection des types d'une sous-classe n'est pas vide (1), et rejetée dans le cas contraire (2). L'intersection entre un aéronef à moteur et un planneur étant vide, la modification est rejetée.

(2) Opérations sur les classes

(2.2) Suppression d'une classe

D'après le principe d'élimination des classes, pour supprimer une classe il faut qu'il n'existe pas d'attributs ayant pour domaine cette classe. Et dans le cas où l'opération est autorisée, les attributs définis dans cette classe seront pré-déclarés dans une super-classe. Pour éviter une perte d'information, les instances de la classe à supprimer sont retirées (opération (6.3)) de celle-ci, et rattachées aux super-classes. Afin de respecter le principe de la suppression de classe, les super-classes de la classe à supprimer sont rattachées aux sous-classes de celle-ci. Un exemple est fournie par la Figure 4-8.

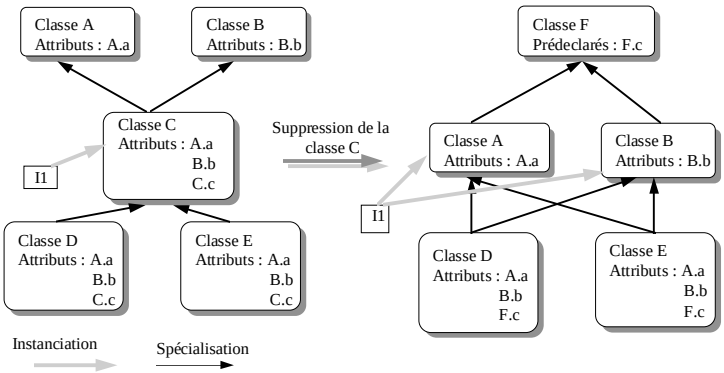


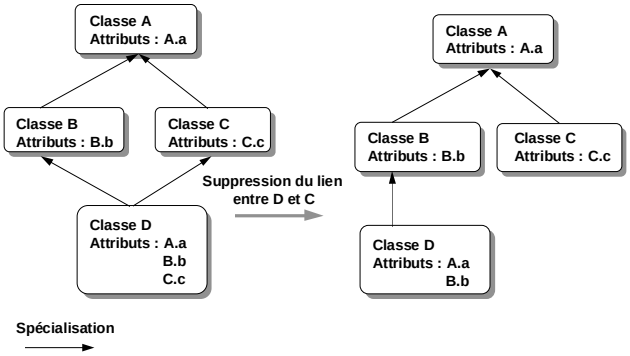
Figure 4-8 : Exemple de suppression de classse

La suppression de la classe C entraîne la pré-déclaration de l'attribut *C.c* dans une superclasse commune (F) et le rattachement des instances aux super-classes (i1). De plus les sous-classes (D et E) sont rattachées aux super-classes (A et B).

(3) Opérations sur les liens inter-classes

(3.2) Suppression d'un lien de spécialisation

La suppression d'un lien de spécialisation entre deux classes est autorisée si la sous-classe possède au moins deux super-classes (respect de l'invariant du graphe de spécialisation I.S.1). Dans le cas où cette suppression est autorisée, les attributs venant uniquement de ce lien sont supprimés du sous-graphe des classes (sauf si l'attribut à supprimer n'est pas hérité par un autre lien d'héritage), de même pour les clefs (Figure 4-9).



La suppression du lien entre D et C entraîne la perte de l'attribut c pour la classe D. La suppression, par exemple, du lien entre C et A sera refusée car ceci entraînerait une classe isolée (C).

Figure 4-9 : Suppression d'un lien de super-classe

Dans le cas où on désire déplacer une classe, il faut supprimer le lien de spécialisation avec sa super-classe : le système refusera si la classe à déplacer possède une seule super-classe. Pour pallier ce problème, on fournit une opération de plus haut niveau, *modifier_super* qui permet le déplacement d'une classe dans un graphe d'héritage ; cette opération est atomique, sinon on obtiendrait un noeud isolé, ce qui est contraire à l'invariant du graphe de spécialisation.

(4) Opérations sur les méthodes

Une méthode étant modélisée par une classe, certaines opérations sur les classes sont autorisées pour les méthodes, en particulier les opérations sur les arguments qui sont des attributs particuliers. Du fait que les méthodes sont des classes organisées dans un graphe d'héritage, les opérations sur les liens de spécialisation et de disjonction sont aussi valables. Pour les opérations de suppression et de création de méthodes, la sémantique est différente. La suppression d'une méthode, contrairement à la suppression d'une classe, n'est autorisée que si cette méthode ne possède que des attributs hérités. La suppression d'une méthode entraîne l'héritage des traitements non redéfinis dans les sous-méthodes directes de la méthode à supprimer (l'héritage des traitements étant dynamique). Si la méthode à supprimer ne possède pas de sous-méthodes, les éventuels traitements (pré, principal, post) sont supprimés (Figure 4-10).

Pour la création d'une méthode, l'utilisateur peut redéfinir les traitements des super-méthodes ou les faire hériter. La définition d'une méthode est évolutive ; il est possible d'ajouter, de supprimer et de modifier les arguments (cf. opérations (1), les arguments étant des attributs). De même, il est possible de modifier et de supprimer les traitements.

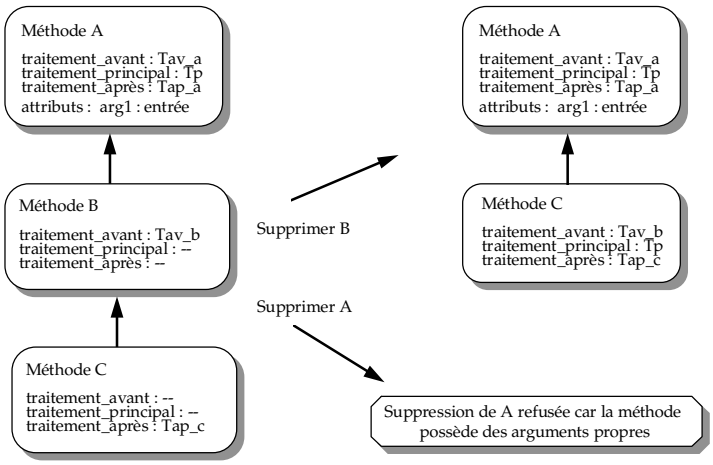


Figure 4-10 : Suppression d'une méthode

*4.2.4.2 **Opérations sur les instances***

Dans ce paragraphe, nous présentons les opérations permettant de manipuler une instance (suppression, modification, déplacement etc.). Ces opérations pourront être utilisées lors de l'expression des contraintes d'intégrité par les règles d'évolution. De même que pour les opérations sur les classes, nous présentons seulement quelques opérations de manipulation d'instances.

(5) Opérations sur les instances

(5.1) (5.2) Création et Modification d'une instance

Les inférences associées à un attribut sont évaluées en création et en modification. L'utilisateur ne peut modifier que les attributs possédant une inférence "user". Les autres attributs sont recalculés.

Lors de la modification d'une instance, seules les inférences et les contraintes de la classe où elles sont définies sont déclenchées. Le maintien de la cohérence de la base n'est donc que partiel. En effet, si la valeur d'un attribut dans une instance I1 (par exemple, l'autonomie d'avion1) est obtenue à partir des valeurs d'une autre instance I2 (la consommation de moteur1), alors le résultat ne peut être maintenu en cas de modification de I2. Les règles d'évolution d'instances ont pour objectif de pallier ces lacunes. Ces règles pourront exprimer des calculs inférentiels de valeurs ou inférences et des contraintes portant sur plusieurs instances d'une même classe ou de classes différentes.

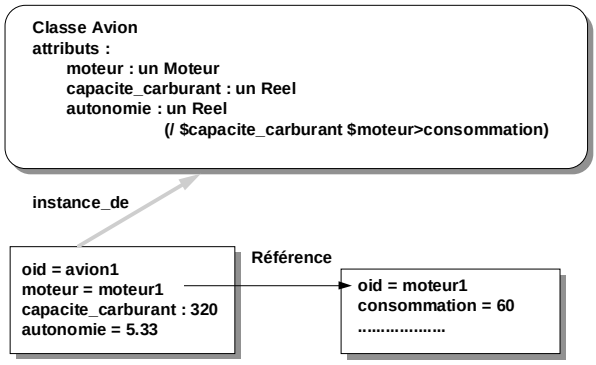


Figure 4-11 : Création d'une instance

L'autonomie de l'avion1 est calculée en fonction de la consommation du moteur1. Si la consommation du moteur1 est modifiée l'autonomie n'est pas recalculée automatiquement malgré l'inférence sur l'autonomie ; il faut que l'utilisateur écrive une règle assurant l'évolution correcte d'une instance de Moteur.

L'opération de modification ne prend pas en compte la suppression d'une valeur d'attribut. Pour supprimer une valeur d'attribut, il faut utiliser une opération spécifique (opération (5.3)).

(5.4) Suppression d'une instance

La suppression d'une instance s'effectue de la manière suivante :

- vérification des droits de suppression (liens de dépendances),
- suppression des instances qui en dépendent existentiellement,
- suppression des valeurs d'attributs,
- suppression des références vers cette instance dans d'autres objets,
- destruction physique de l'instance.

On vérifie que l'instance à supprimer n'est pas partagée entre deux objets (lien de dépendance partagé). Si c'est le cas, l'opération de suppression est refusée. Dans le cas contraire, on supprime récursivement les instances qui sont dépendantes existentiellement.

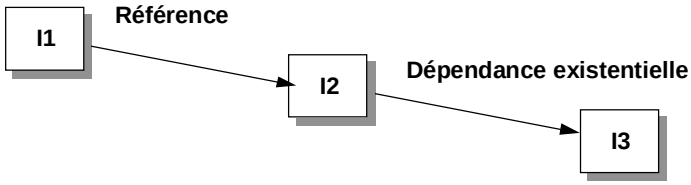


Figure 4-12 : Suppression d'une instance

La suppression de I2 entraîne celle de I3, et I1 perd la référence vers I2. Si le lien entre I1 et I2 était un lien partagé, la suppression de I2 serait refusé.

Les différentes valeurs d'attributs sont supprimées (opération (5.3)). Avant de détruire physiquement l'instance, les références vers l'instance sont supprimées. L'information sur les références est contenue dans la valeur de l'attribut système *est_utilisé_par* de l'instance à supprimer.

Si l'instance à supprimer est référencée par un attribut, alors cet attribut est valué à NIL. On rajoutera par la suite une règle qui supprimera les instances contenant un attribut obligatoire

référencant l'instance à supprimer. Cette règle pouvant entraîner une perte d'informations, elle pourra être redéfinie, par exemple en rejetant la suppression de l'instance de départ.

(6) Opérations sur les liens d'instanciation

Dans le système SHOOD, une instance peut appartenir à plusieurs classes (cf. Chapitre 3). De même une instance peut changer de classe d'appartenance, d'où la nécessité de faire évoluer le lien d'instanciation. Les opérations sur les liens d'instanciation seront utilisées pour faire évoluer un lien d'instanciation entre une classe et sa métaclasse, par exemple rendre facultatif un attribut obligatoire: comme un attribut est modélisé par une classe (instance de *Méta_attribut*), le rattacher à la métaclasse *Méta_att_obligatoire* lui attribue le caractère obligatoire (cf. Chapitre 3).

(6.2) Suppression d'un lien d'instanciation

La suppression d'un lien d'instanciation consiste à défaire l'opération d'ajout. La suppression est autorisée si le lien à supprimer n'est pas unique (invariant d'instanciation I.I.1). La suppression de ce lien entraîne la suppression des valeurs d'attributs venant uniquement de ce lien d'instanciation. Ensuite, ce lien est supprimé.

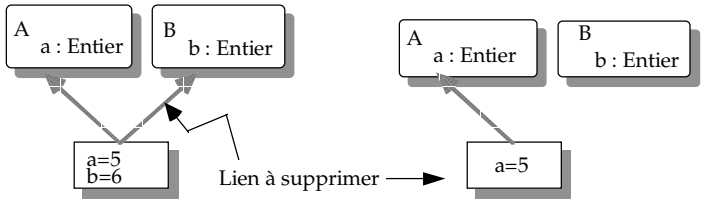


Figure 4-13 : Suppression de lien d'instanciation

(6.3) Retrait d'un lien d'instanciation

L'opération de retrait est similaire à l'opération de suppression (6.2), sauf que le lien d'instanciation n'est pas supprimé mais déplacé vers les super-classes. Donc le retrait d'un lien d'instanciation consiste à retirer une instance d'une classe et à la rattacher aux super-classes de la classe de retrait. Toutes les valeurs d'attributs définis dans la classe de retrait sont supprimées. Le lien à retirer n'est pas forcément unique.

4.2.5 Réalisation

Les opérations, citées précédemment, sont implémentées dans le système SHOOD par des méthodes organisées en plusieurs graphes suivant le type d'opération ; le code des méthodes est écrit en langage Le_Lisp. Les méthodes modélisant les opérations d'évolution peuvent être hiérarchisées du fait que tout est objet. En effet, une méthode est une classe, qui est elle

même une instance de méta-classe, d'où on déduit que le comportement d'une méthode est plus spécifique que celui d'une classe, qui est plus spécifique que celui d'une instance.

Ce graphe est évidemment extensible par de nouvelles méthodes d'évolution. Les méthodes des sous-graphes de création, de suppression et de modification possèdent la même organisation. La figure ci-dessous (Figure 4-14) présente une vue globale de ces méthodes en explicitant uniquement le graphe de suppression.

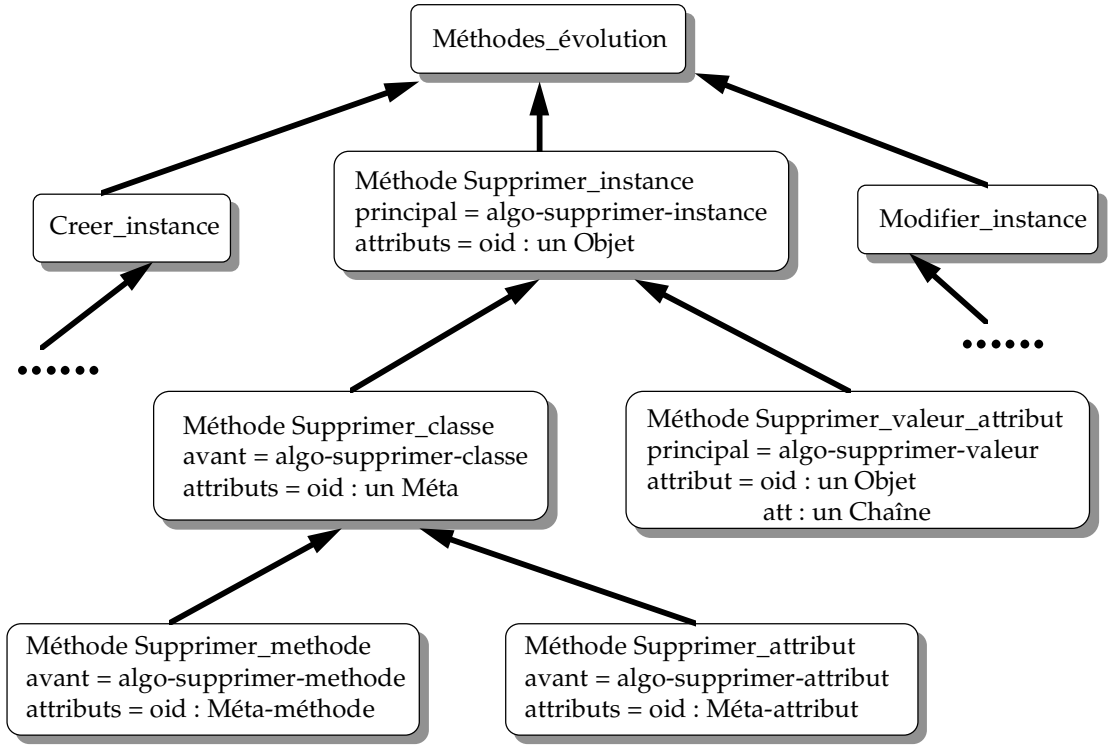


Figure 4-14 : Hiérarchie de méthodes d'évolution

La méthode Supprimer_instance est la méthode la plus générale du graphe de suppression. Elle est redéfinie par la méthode permettant de supprimer la valeur d'un attribut en redéfinissant le traitement "principal" et en ajoutant un argument : *att* pour désigner l'attribut dont on doit supprimer la valeur. Trois autres méthodes redéfinissent la méthode supprimer_instance : Supprimer_classe, Supprimer_méthode et Supprimer_attribut. Ces méthodes héritent du traitement "principal" (suppression d'une instance) et définissent chacune un traitement "avant" qui permet de tenir compte du type de l'instance (classe, méthode, attribut).

4.3 CONCLUSION

Pour clore la section sur les opérations de manipulation, nous présentons les tableaux comparatifs des opérations possibles et des invariants (cf. Tableau 2-1 et 2-2 du Chapitre 2), complétés avec les opérations et les invariants dans SHOOD.

Opérations	Orion	GemStone	OTGen	O2	Shood
Attributs ou Variables d'instance					
ajout	✔	✔	✔	✔	✔
suppression	✔	✔	✔	✔	✔
retrait					✔
renommage	✔	✔	✔	✔	✔
redéfinition type	✔	✔	✔	✔	✔
modification de l'origine		✔			✔
modification de la valeur par défaut		✔			
Méthodes					
ajout	✔			✔	✔
suppression	✔			✔	✔
renommage	✔			✔	✔
redéfinition de la signature				✔	✔
modification de code	✔			✔	✔
modification de l'origine	✔				✔
Valeurs partagées	✔				✔
Lien de composition	✔				✔
Classes					
ajout	✔	✔	✔	✔	✔
suppression	✔	✔	✔	✔	✔
renommage	✔		✔	✔	✔
Classes indexables	✔				
Liens d'héritage					
ajout d'une super-classe	✔		✔	✔	✔
retrait d'une super-classe	✔		✔	✔	✔
modification de la précédence	✔				

Tableau 4-1 : Comparaison des opérations

Invariants	Orion	GemStone	OTGen	O2	Shood
Représentation		✔			✔
Graphe de classes	✔	✔	✔	✔	✔
Noms distincts	✔		✔	✔	✔
Héritage	✔	✔	✔	✔	✔
Origine distincte	✔			✔	
Compatibilité de domaine	✔	✔	✔	✔	✔
Domaine d'attribut			✔		
Cohérence des références		✔	✔		

Tableau 4-2 : Comparaison des invariants

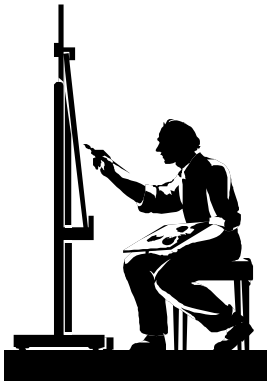
Certains invariants et opérations, comme les invariants et opérations concernant les méta-classes et le lien de disjonction, ne sont pas précisés dans ces tableaux car ils sont spécifiques au système SHOOD.

Nous présentons dans le tableau suivant les différents impacts de modification sur les instances

Invariants	Orion	GemStone	OTGen	Encore	Shood	Clamen
Conversion immédiate		✔			✔	
Conversion différée	✔		✔			
Versionnement						✔
Emulation				✔		

Tableau 4-3 : Comparaison des impacts de modification

Dans ce chapitre, nous avons présenté uniquement le noyau du système d'évolution. Nous allons, dans les chapitres suivants, présenter les différents mécanismes composant notre système. Pour cela, dans le chapitre suivant, nous décrivons le mécanisme de classification d'instances qui permet l'évolution d'une instance en complétude et en cohérence dans un graphe de classes. Ce mécanisme utilisera particulièrement les invariants d'instanciation (I.I.4 et I.I.5) décrites dans le chapitre 3, ainsi que les opérations d'évolution d'instances présentées dans ce chapitre.



5. LE MECANISME DE CLASSIFICATION D'OBJETS EVOLUTIFS

Ce chapitre est consacré à la présentation d'un mécanisme de classification d'objets évolutifs, incomplets et temporairement incohérents tels qu'on les rencontre dans les applications de conception [Nguyen92a] [Liotard93] [Bounaas94b]. Ce mécanisme a pour but de permettre le passage de l'Objet A Concevoir (OAC) d'un certain état vers un autre, plus complet et donc intuitivement plus proche de l'objet final. Cette évolution peut se faire simultanément suivant plusieurs représentations ou points de vue de l'objet [Nguyen92a] [Carre90] [Ra94]. L'idée consiste donc à raisonner sur un ensemble de connaissances incomplètes, modélisant l'OAC à un certain niveau de résolution, afin de dériver des objets informatiques plus complets et donc plus proches de l'objet conçu. Ce mécanisme, mis en oeuvre dans le système SHOOD, s'appuie sur un raisonnement consistant à permettre l'utilisation des méthodes de calcul en cours de classification.

La tâche d'un mécanisme de classification[4] consiste en général à gérer une hiérarchie d'objets ordonnés par une relation d'ordre partiel qui est souvent la relation d'héritage. Classifier un objet consiste à rechercher la place qui lui convient le mieux dans le graphe d'héritage, en "comparant" l'objet à classifier avec les objets déjà présents [Haton91] [Marino91] [Girard95]. Cette comparaison, appelée aussi appariement, met en évidence les dépendances existantes entre les objets selon l'ordre partiel qui a permis de gérer la hiérarchie [MacGregor91]. Selon la connaissance traitée, deux niveaux de classification sont possibles : la classification des instances et celle des classes [Napoli92]. Seule la classification d'instances sera abordée ici. Elle consiste à rechercher dans un graphe de classes, à quelles classes une instance particulière peut être rattachée. La classification des classes est traitée par exemple dans [Capponi94]

Dans SHOOD, l'appariement d'une instance à une classe consiste à déterminer le statut de l'instance à classifier (possible ou impossible) en vérifiant l'invariant d'instanciation (contraintes satisfaites, attributs obligatoires valués, etc.). De plus, lors de cet appariement, l'instance à classifier est complétée en exécutant les inférences (calcul de valeurs d'attributs) associées aux attributs.

---

[4] pris ici dans "le sens qui lui est communément accordé en IA : celui de classement " [Haton91].

Avant de décrire le principe de ce mécanisme, nous allons insister sur deux caractéristiques importantes du modèle SHOOD, qui sont la multiplicité des inférences et la multi-instanciation et qui en font des éléments fondamentaux pour le mécanisme de classification.

5.1 ELEMENTS FONDAMENTAUX

Nous rappelons que dans SHOOD, une classe est définie par un ensemble d'attributs auxquels sont associés des descripteurs de type (domaine de valeur), d'inférence (calcul de la valeur) et de contrainte (vérification de la cohérence de la valeur).

SHOOD offre la possibilité de modéliser une disjonction explicite de concepts. Elle est nécessaire lorsque les classes modélisent des domaines mal définis pour lesquels on ne dispose pas de propriétés discriminantes. Cette incompatibilité de concepts peut être explicitement modélisée par un lien de disjonction. La disjonction interdit l'existence d'instances communes ou de spécialisation commune à deux classes dont l'incompatibilité n'a pu être explicitée par des propriétés discriminantes.

Pour une meilleure compréhension du mécanisme de classification, nous allons décrire plus en détails le problème de la redéfinition des inférences et celui de la multi-instanciation.

5.1.1 Multiplicité des inférences

Un même attribut peut avoir des définitions différentes dans les classes où il est hérité. En particulier, les inférences permettant de déterminer sa valeur peuvent être redéfinies (cf. Chapitre 3). Cette redéfinition peut avoir plusieurs buts : celui d'exprimer un autre moyen de calcul plus rapide et/ou mettant en jeu des paramètres différents, ou celui d'exprimer un calcul plus précis.

Comment traiter cette redéfinition?

Tout dépend du sens et de l'utilisation de l'attribut dans l'application. Prenons l'âge d'une personne, s'il s'agit de calculer les impôts des personnes, une simple estimation de la valeur de l'attribut de l'âge est suffisante. Par contre, pour la détermination d'équipe sportive, un calcul exact et précis de l'âge des personnes est indispensable. Pour traiter ces deux types d'attributs nous parlerons d'attribut à valeur unique et d'attribut à valeur dépendante.

Un attribut à valeur unique est une propriété dont la valeur ne dépend pas du contexte (ici la classe) dans lequel elle a été calculée.

Un attribut à valeur dépendante est une propriété dont la valeur est fonction du contexte (ici la classe) dans lequel elle a été calculée.

Redéfinition des inférences pour des attributs à valeur unique

Dans le cas des attributs à valeur unique, la redéfinition des inférences est la plupart du temps due à l'existence, dans la sous-classe, de nouvelles propriétés permettant un calcul équivalent. L'équivalence de calcul ne signifie pas que les deux modes de calcul rendent toujours un résultat identique, ce qui d'ailleurs n'est pas vérifiable, mais qu'une estimation de la valeur étant suffisante, les différences qu'ils peuvent engendrées sont sans importance pour l'application. Intuitivement si l'on souhaite établir le montant des impôts d'une personne, il importe peu de calculer l'âge d'une personne à partir de l'année de naissance ou du numéro de sécurité sociale (Figure 5-1).

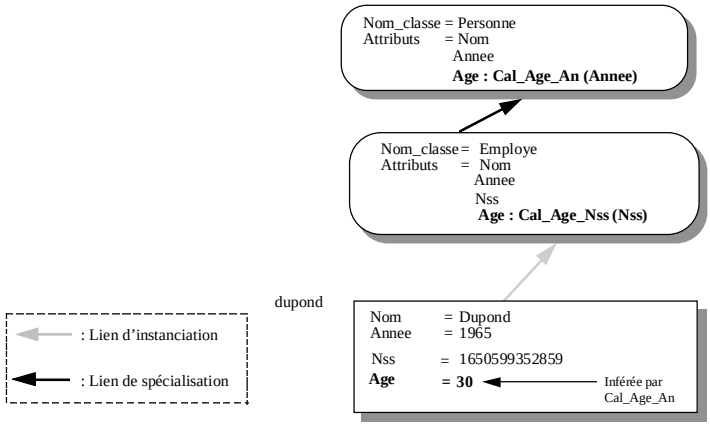


Figure 5-1 : Attribut à valeur unique

Dans le cadre de la conception, ce type d'attribut est fréquemment utilisé dès lors qu'une estimation de valeur est suffisante pour poursuivre l'activité de conception. C'est ainsi, que dans les applications mécaniques la représentation technologique d'une pièce s'appuie sur une géométrie minimale (et donc imprécise) de celle-ci ; il est donc inutile de recalculer, même si elles existent, par des méthodes plus précises les dimensions géométriques de la pièce.

Redéfinition des inférences pour des attributs à valeur dépendante

Dans le cas des attributs à valeur dépendante, la redéfinition des inférences est due à l'existence dans les sous-classes de calculs plus précis dont la validité dépend du contexte (ici de la classe) dans lequel ils sont définis. La multiplicité des inférences correspond alors à des moyens de calculs différents et ne rendant pas des résultats équivalents : la valeur rendue n'est valide que dans la classe où elle a été inférée. Le calcul de l'autonomie en vol d'un Aéronef dépend fortement du type de l'Aéronef (Figure 5-2). L'autonomie calculée par une méthode spécifique dans la classe hélicoptère est plus précise que celle estimée par l'utilisateur dans la classe Aéronef. Cependant cette valeur ne sera valable que pour les hélicoptères.

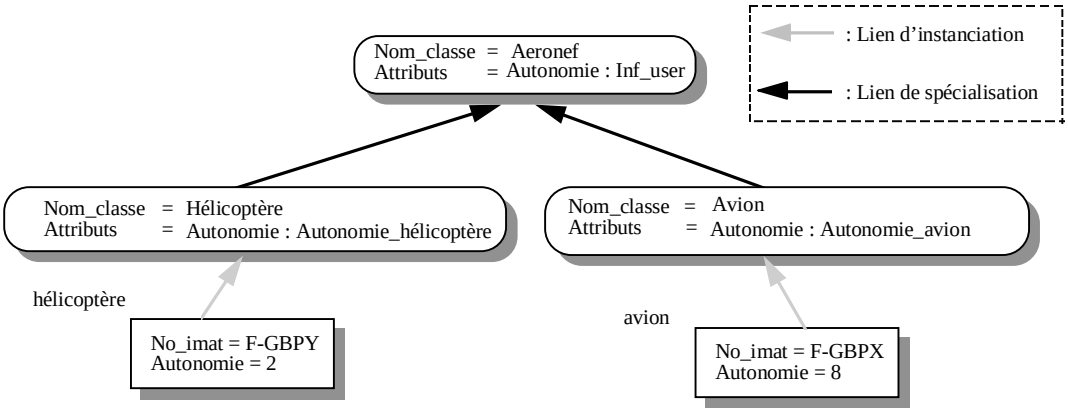


Figure 5-2 : Attribut à valeur dépendante

Dans le cadre de la résolution de problèmes de conception, ce type d'attribut est fréquemment utilisé pour modéliser les paramètres fondamentaux du problème dès lors que l'on dispose de calculs précis et spécifiques. Lorsqu'il s'agit par exemple d'obtenir une géométrie complète de la pièce, des calculs précis et fonction de la forme de la pièce devront être utilisés pour établir ses dimensions géométriques.

La détermination du statut d'un attribut : à valeur unique/dépendante et obligatoire/facultatif n'est pas fonction de la connaissance représentée, mais de son utilisation dans le contexte applicatif. Dans le cadre de la conception, le contexte applicatif est lié au problème ou au sous-problème à résoudre. C'est ainsi que pour les problèmes liés à la technologie de la pièce, les dimensions géométriques sont des attributs obligatoires mais à valeur unique. Par contre, pour les problèmes liés à la cotation géométrique de la pièce les dimensions sont des attributs obligatoires à valeur dépendante.

Toute modification est soumise à un ensemble de règles permettant de préserver la cohérence globale du schéma. Cette cohérence est définie par un ensemble d'invariants de schéma que le modèle doit respecter (cf. Chapitre 2). De même, toute évolution d'instance est soumise à un ensemble de règles permettant de préserver la cohérence des instances. Cette cohérence est définie par un ensemble d'invariants (cf. Chapitre 3). Certains d'entre eux garantissent la conformité d'une instance par rapport à sa classe, ils doivent être respectés par tout mécanisme d'évolution d'instances et en particulier par le mécanisme de classification.

5.1.2 Multi-instanciation

Dans le chapitre 3, nous avons décrit le lien d'instanciation et de multi-instantiation avec les invariants (I.I.4 et I.I.5) correspondant à ces concepts. Nous allons rappeller la définition de ces deux liens et leur invariants, car ils déterminent les conditions pour qu'une ou plusieurs classes soient possibles pour une instance donnée.

L'instanciation dans SHOOD est un lien qui permet le rattachement d'une instance à une classe dont elle n'est pas un moulage parfait. Ceci est en partie pris en compte par les concepts de contraintes fortes/faibles et d'attributs obligatoires/facultatifs (cf. Chapitre 3). Et, la multi-instanciation permet de rattacher une instance à plusieurs classes possibles et qui ne sont pas directement ou indirectement des spécialisations les unes des autres [Nguyen92a]. Elle autorise donc le rattachement d'une instance à des classes comportant des sémantiques différentes, chacune d'elles modélisant un point de vue de l'objet.

- **Cohérence d'une instance**. Une instance qui respecte l'ensemble des contraintes (faibles et fortes) associées aux attributs valués, est dite *cohérente* [Escamilla90]. Une instance est *fortement cohérente* si toutes ses contraintes fortes sont respectées. On parle d'*incohérence forte* lorsqu'au moins une des contraintes fortes est violée.

- **Complétude d'une instance**. Une instance valuant tous les attributs de sa classe est dite *complète*. Une instance est *minimale* si tous ses attributs obligatoires sont valués. Si une instance n'est pas minimale, c'est à dire qu'au moins un de ses attributs obligatoires n'est pas valué, l'instance est dite *fortement incomplète*.

L'invariant suivant fixe les conditions minimales d'appartenance d'une instance I à une classe C. Si ces conditions sont respectées, la classe C est dite *possible* pour I.

> **I.I.4 : Invariant d'instanciation**
> Une classe C est possible pour une instance I ssi :
> • I est fortement cohérente dans C,
> • I est minimale dans C,
> • tous les attributs I ont été inférés dans le graphe de C [5].

Cet invariant doit être étendu pour prendre en compte les conditions minimales d'appartenance d'une instance I à un ensemble E de classes. Par extension, nous appellerons graphe de E, le graphe constitué de toutes les classes de E et de toutes leurs super-classes directes ou indirectes.

> **I.I.5 : Invariant de multi-instanciation**
> Un ensemble E de classes est possible pour une instance I si et seulement si :
> • I est fortement cohérente dans toutes les classes de E,
> • I est minimale dans toutes les classes de E,
> • tous les attributs de I ont été inférés dans le graphe de E,
> • il n'existe pas de disjonction explicite dans le graphe de E.

[5] Le graphe de C est constitué de C et de toutes ses super-classes directes ou indirectes.

Conséquence : Deux classes de E ne peuvent pas avoir d'attributs à valeur dépendante en commun. En effet, la valeur d'un attribut à valeur dépendante ne peut pas être simultanément inférée dans les deux classes. Deux classes ayant un attribut à valeur dépendante en commun, sont donc incompatibles.

Définition : Une instance qui ne justifie pas ses valeurs est une instance dont les attributs ont été inférés dans d'autres classes que celles précisées dans les invariants I.I.4 et I.I.5.

Lorsqu'il s'agit de classes modélisant des domaines bien connus, l'incompatibilité est souvent modélisée par des propriétés discriminantes et conflictuelles. Elles correspondent à un conflit de type sur un attribut commun, à un attribut commun à valeur dépendante, à des contraintes fortes conflictuelles, etc. C'est par exemple le cas des classes *Enfants* et *Adultes* pour lesquelles, il existe une contrainte forte bien connue sur l'*âge*. Ces deux classes sont disjointes de par des contraintes fortes discriminantes et conflictuelles. De même si l'autonomie d'un Aéronef est un attribut à valeur dépendante (cf. Figure 5-2), une même instance ne pourra pas être rattachée simultanément à deux sous-classes d'Aéronef, par exemple les classes Avion et Hélicoptère, comportant des inférences spécifiques pour cet attribut. Ces deux classes sont disjointes de part une spécificité de calcul conflictuel : une même instance ne peut être simultanément un hélicoptère de 2h d'autonomie et un avion de 8h d'autonomie. Par contre, lorsque des classes modélisent des domaines mal définis, l'absence de propriétés discriminantes peut masquer une incompatibilité de concepts. Cette incompatibilité est alors explicitée dans SHOOD, par un lien de disjonction (cf. 5.1). Un ensemble E, possible pour une instance I, ne peut donc pas comporter de lien de disjonction dans son graphe.

5.2 LIMITES DE LA REPRESENTATION PAR UN OBJET INFORMATIQUE

La multi-instanciation autorise la modélisation d'un objet réel par une instance potentiellement rattachable à un ensemble de classes.

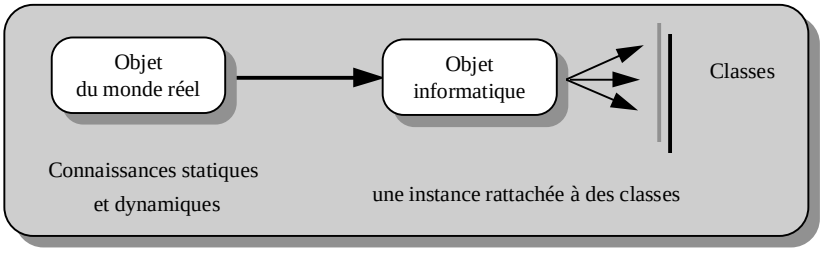


Figure 5-3 : Représentation par un objet informatique

Cette interprétation (Figure 5-3) ne tient compte que des connaissances statiques. Or le modèle permet l'expression :

- des inférences, qui permettent d'augmenter les connaissances détenues sur l'objet de façon dynamique,

- de leur multiplicité, qui permet d'augmenter ces connaissances en fonction des classes potentielles de rattachement de l'objet.

Lorsqu'il s'agit, par exemple, d'un attribut ATT à valeur dépendante défini dans une classe C, une redéfinition d'inférence dans une sous-classe C' représente un autre moyen de calcul : on est dans le cas où la valeur inférée dans C n'est pas valide dans la sous-classe C'. Potentiellement, l'objet réel peut être modélisé soit par une instance de C dont la valeur de ATT a été inférée dans C, soit par une instance de C' dont la valeur a été inférée dans C'. L'exploitation de la multiplicité des inférences permet donc de générer les différentes valeurs possibles de l'objet en proposant différents objets informatiques, chacun correspondant à une instance rattachable à un ensemble de classes (cf. 5.1.1).

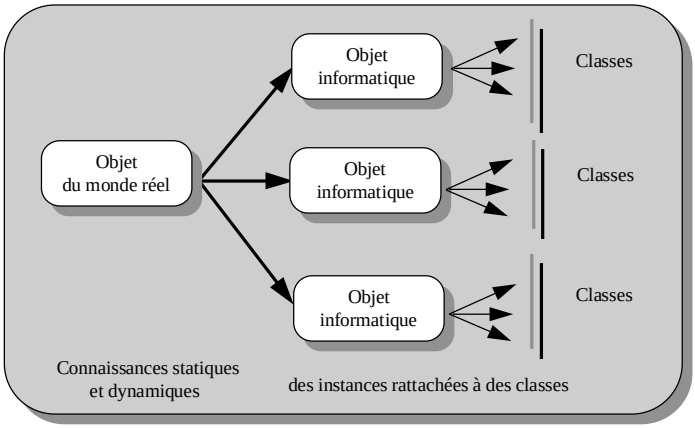


Figure 5-4 : Plusieurs objets informatiques

Les conflits d'inférences sur un même attribut ne sont pas les seuls cas de modélisation d'un objet réel par plusieurs objets informatiques. Nous verrons en particulier que le mécanisme de classification génère plusieurs objets informatiques lors du traitement des disjonctions explicites et des attributs sans valeur (cf. 5.4).

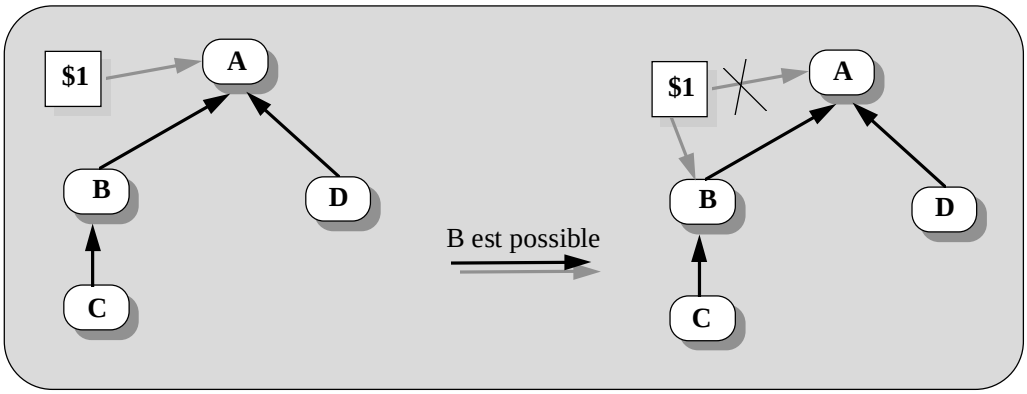
5.3 DESCRIPTION INFORMELLE DE LA CLASSIFICATION

Le rôle du mécanisme de classification est donc, à partir d'un objet du monde réel, de générer un ensemble d'objets informatiques, c'est à dire des instances rattachables à un ensemble de classes. Pour ce faire, il s'appuie sur un ensemble de règles prenant en compte les invariants cités précédemment (cf. § 5.1.2) [Bounaas94b] [Liotard93].

L'exécution du mécanisme de classification s'effectue dans un sous-graphe de classes que nous appellerons le graphe de recherche. Il s'exécute sur la classe racine, puis examine les autres classes du graphe afin de déterminer l'ensemble des classes possibles. Lors de

l'appariement d'une instance à une classe, le statut de cette instance (possible ou impossible) est déterminé en vérifiant l'invariant I.I.4.

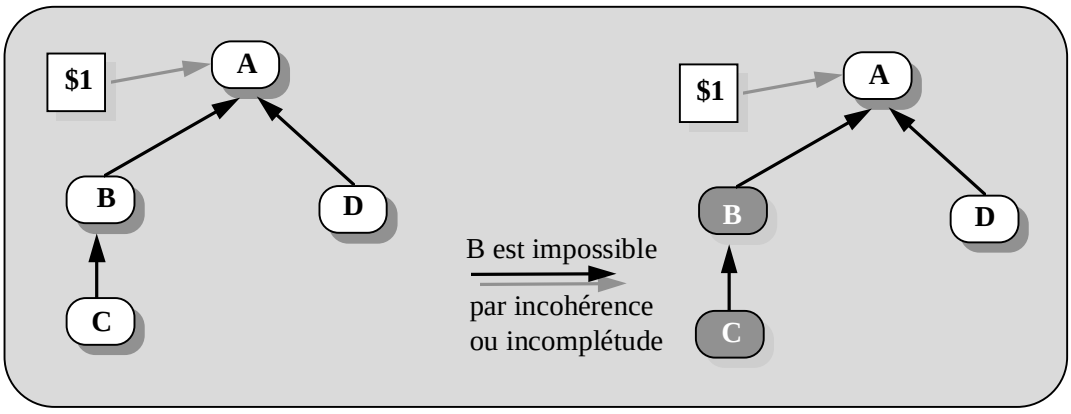
Si une classe est possible, au sens de l'invariant d'instanciation (Invariant I.I.4), alors l'instance est rattachée à cette classe et le processus d'appariement est réitéré dans les sous-classes.



La classe A étant possible pour l'instance $1, le mécanisme examine la classe B qui à son tour est possible. L'instance est donc rattachée à B, plus spécifique que A.

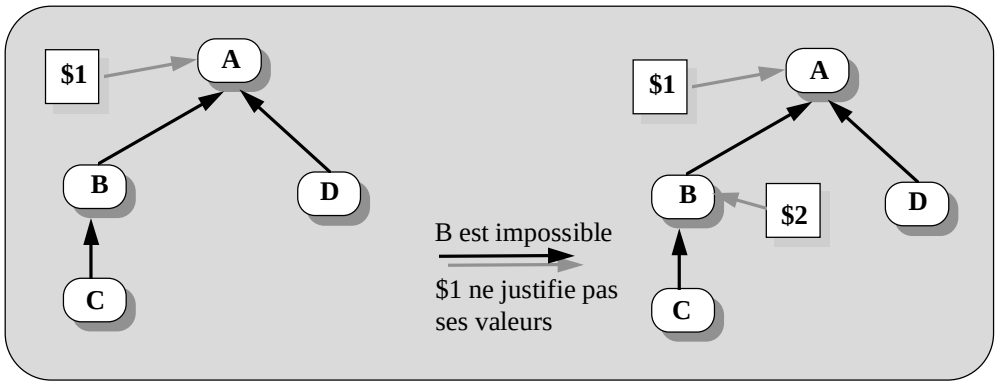
Dans le cas où la classe d'appariement est impossible pour l'instance (non respect de l'invariant I.I.4), on distingue alors deux cas.

1. L''instance est fortement incohérente ou fortement incomplète (cf. § 5.1.2) ; elle ne peut être rattachée à la classe ni aux sous-classes de celle-ci, la spécialisation étant stricte.



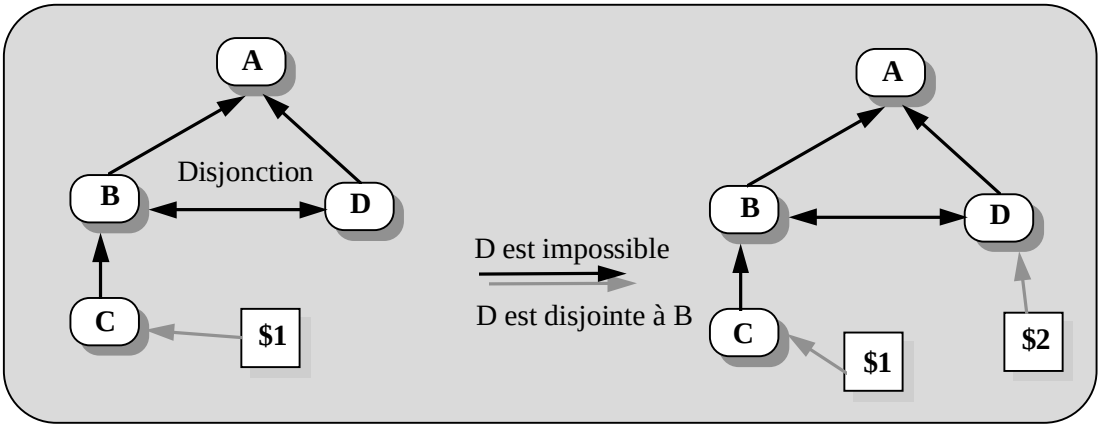
La classe B n'est pas possible pour $1, par incomplétude ou incohérence forte, la classe C est donc impossible, mais la classe D sera examinée.

2. La classe est impossible car l'instance ne justifie pas ses valeurs, une autre instance est alors générée à partir de l'instance en cours de classification (instance génératrice) (cf. § 5.4). L'instance générée est une copie différentielle de son instance génératrice. Elle possède une information différente qui lui permet de justifier ses valeurs.



La classe B n'est pas possible pour $1, car l'instance ne justifie pas ses valeurs, l'instance $2 est générée à partir de $1. Les classes C et D seront examinées pour $2, mais pour $1 on examinera que la classe D.

Après examen de la classe la plus spécifique, le mécanisme tente de rattacher l'instance aux autres classes du graphe par multi-instanciation, tout en respectant l'invariant I.I.5. En particulier, il devra vérifier que deux classes de rattachement ne sont pas disjointes. Dans le cas où il y a disjonction, il faut générer une autre instance, afin de respecter l'invariant de multi-instanciation (cf. § 5.4.3).



L'instance $1 est possible pour la classe C, mais la classe D disjointe de B, est impossible pour $1 (Invariant I.I.5). $1 ne sera pas classifiée dans D, mais une instance $2 (générée de $1) sera rattachée à D mais pas à B.

Nous regardons maintenant, plus en détail, les différents cas de générations d'instances.

5.4 GENERATION D'OBJETS INFORMATIQUES

5.4.1 La multiplicité des inférences

Soit C une classe possible pour l'instance en cours de classification. Pour l'attribut A1, on dispose d'une valeur inférée dans C par une inférence Inf1. Lors de l'examen d'une sous-classe

de C le problème à résoudre par le mécanisme de classification est alors : "une autre inférence (Inf2) pour l'attribut A1 doit-elle ou non être exécutée? ".

Lorsqu'il s'agit d'un attribut à valeur unique et possédant pour l'instance en cours de classification une valeur dans la classe C, la valeur est conservée dans les sous-classes de C. Autrement dit, la valeur de l'attribut reste correcte tant que l'instance appartient directement ou indirectement à la classe qui a permis de l'inférer. Aucune autre inférence rencontrée sur le même attribut ne sera donc prise en compte.

Supposons à présent, qu'il s'agisse d'un attribut à valeur dépendante et possédant une valeur dans la classe C pour l'instance à classifier. Cette instance sera impossible pour les sous-classes, car elle ne justifiera pas ses valeurs (Invariant I.I.4 et I.I.5). Nous proposons, dans les sous-classes de C détenant une inférence propre pour A1, de générer une nouvelle instance. La nouvelle instance est créée par copie de l'objet en cours de classification dont elle a les mêmes valeurs d'attributs, sauf pour l'attribut où il y a un conflit d'inférence (attribut A1). La génération de cette instance est régie par la règle suivante :

***Règle 1** : Soit I une instance appartenant à une classe C et valuant un attribut A à valeur dépendante. I ne peut appartenir à une classe C' avec la même valeur pour A, mais une instance I', générée à partir de I avec une autre valeur de A, peut appartenir à C' (mais pas à C).*

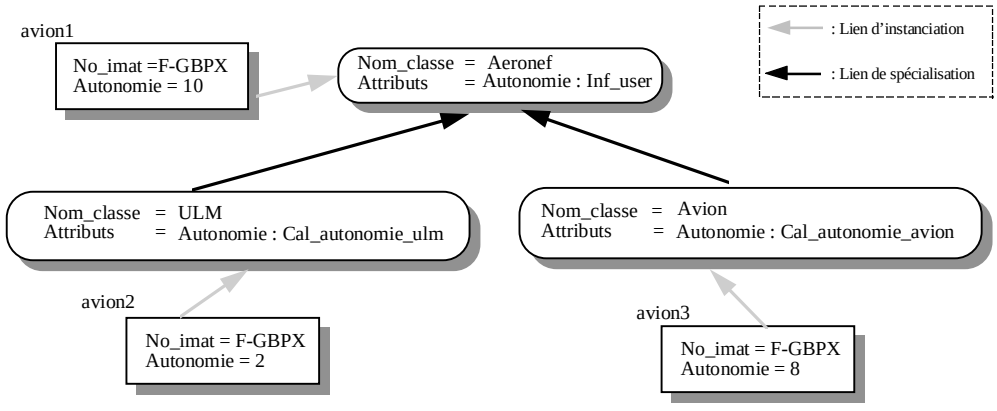


Figure 5-5 : Multiplicité des inférences

En classifiant un *Aéronef* (Figure 5-5), le mécanisme trouve une première classe d'appartenance : la classe *Aéronef*, et affecte la valeur (10) de l'autonomie saisie par l'utilisateur. Le mécanisme continue la classification de l'instance *avion1* dans les sous-classes. Les inférences *Inf_user* et *Cal_autonomie_ulm* sont en conflit. Le mécanisme de génération d'instances est alors utilisé : l'instance *avion2* est générée à partir de l'instance *avion1* avec une valeur pour l'autonomie inférée par *Cal_autonomie_ulm* (autonomie = 2). De la même manière, lors de la classification de l'instance *avion1* dans la classe *Avion*, l'instance

avion3 est générée avec une valeur pour l'autonomie inférée par *Cal_autonomie_avion* (autonomie = 8).

L'utilisation de la règle 1 est correcte dans ce contexte : Il ne serait pas cohérent de garder la valeur de l'autonomie calculée dans le cas d'un ULM pour un Avion. La classification de avion1 terminée, le mécanisme tente de classifier avion2 et avion3 respectivement dans Avion et ULM. Ces deux classes étant disjointes, avion2 (respectivement. avion3) ne peut être rattaché à Avion (respectivement ULM).

5.4.2 La gestion d'un attribut sans valeur

Illustrons le problème de l'attribut sans valeur par un exemple. Nous disposons pour une personne de son nom *Dupond* et de sa date de naissance 1965. L'utilisateur demande à classifier cette instance à partir de la classe *Personne* (la modélisation suivante n'est pas forcément la plus correcte).

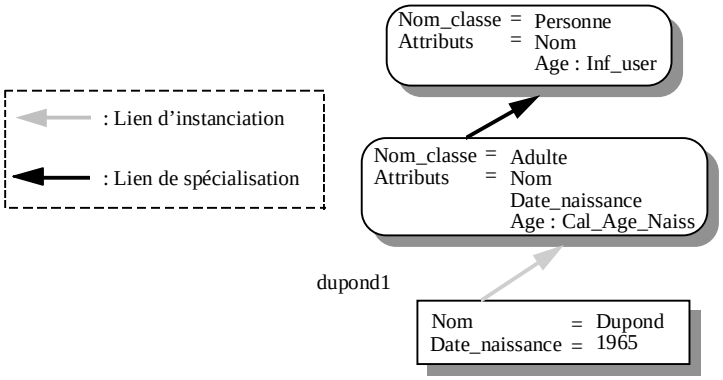


Figure 5-6 : Doit-on utiliser Cal_Age_Naiss pour l'instance dupond1?

Le mécanisme commence par la classe *Personne* où l'inférence associée à l'attribut *âge* est exécutée. L'utilisateur n'ayant pas fourni cette information, l'inférence échoue : l'attribut *âge* est sans valeur. Puis la classification s'effectue sur la classe *Adulte*. Doit-on utiliser l'inférence *Cal_Age_Naiss* pour l'instance *dupond1*? Si on ne l'utilise pas, nous perdons l'occasion de calculer l'âge. Mais si l'inférence est utilisée, *Adulte* sera une classe possible pour *Dupond*, mais cette dernière ne peut être rattachée à la classe *Personne*, car elle ne justifie pas ses valeurs dans la classe Personne.

Pour pallier ce problème, nous traitons un attribut sans valeur avec le mécanisme de génération d'instance. Lorsque de nouvelles inférences sont définies pour un attribut sans valeur, une nouvelle instance est générée. Pour ce cas, nous définissons la règle 2.

***Règle 2** : Soit I une instance appartenant à une classe C et n'ayant pas de valeur pour un attribut A. I ne peut appartenir à une sous-classe C' où A est valué, mais une instance I', générée à partir de I, peut appartenir à C' (mais pas à C).*

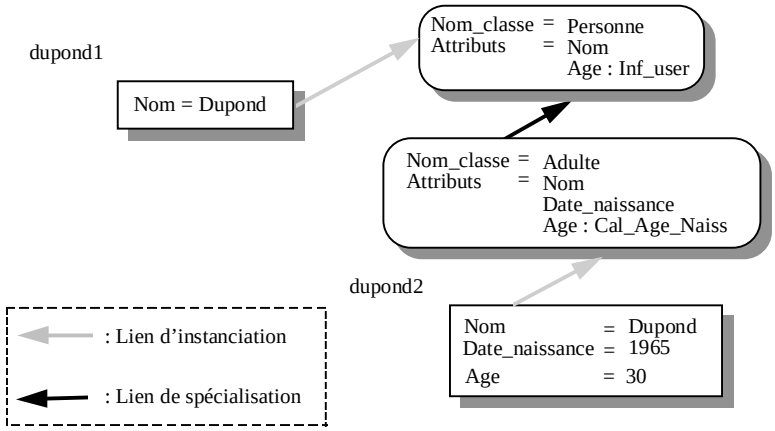


Figure 5-7 : Utilisation de Cal_Age_Naiss pour l'instance dupond2

Après la classification, nous avons une instance *dupond1* représentant les connaissances de l'utilisateur sur *Dupond*. Cette instance peut être rattachée à la classe *Personne*. L'autre instance *dupond2* représente un autre objet informatique modélisant *Dupond* avec un âge de 27 ans. Cet objet ne peut être rattaché qu'à la classe *Adulte*.

5.4.3 La disjonction

Lorsqu'une instance appartient à une classe C disjointe d'une classe C', la première idée est de dire que cette instance ne pourra jamais appartenir à la classe C'. Or le modèle SHOOD prend en compte des objets incomplets et incohérents. Une instance peut donc être rattachée à une classe avec un certain nombre d'informations incomplètes et donc pourrait être rattachée à une classe disjointe avec des informations différentes. Nous ne pouvons donc pas privilégier une classe plutôt qu'une autre. Mais l'instance ne peut pas appartenir aux deux classes disjointes qui sont globalement impossibles pour l'instance (§ 5.1 et invariant I.I.5).

Pour répondre à ce besoin, nous utilisons le mécanisme de génération d'instance vu précédemment. Lors de la classification d'une instance dans une classe disjointe d'une classe d'appartenance, nous générons un objet informatique qui représente une information différente.

Du fait de l'incomplétude des connaissances, l'avion est représenté par deux objets informatiques : *avion1* vu comme un *Avion_affaire* et *avion2* comme un *Avion_ligne* (*Avion_affaire* étant disjointe d'*Avion_ligne*) (Figure 5-8).

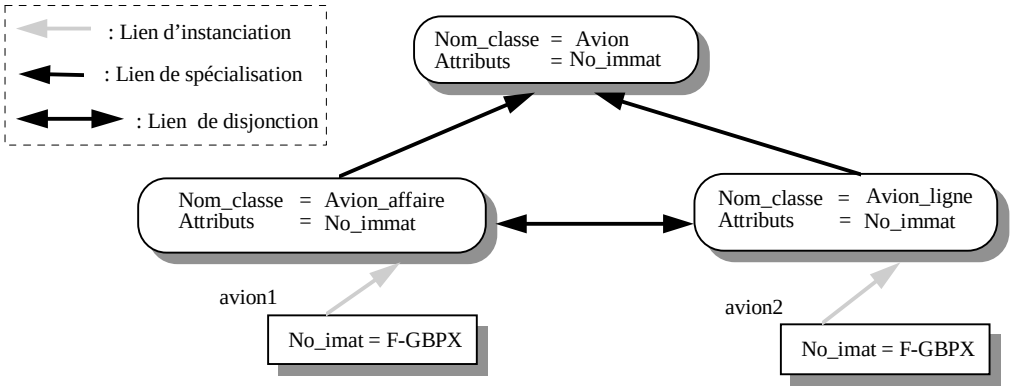


Figure 5-8 : La disjonction

La relation de disjonction interdit une instance d'appartenir à deux classes disjointes. Autrement dit : si I est rattachée à une classe C ou à une de ses sous-classes, alors elle ne peut être rattachée aux classes disjointes de C ou à leurs sous-classes. En revanche, un objet du monde réel peut être modélisé par deux objets informatiques instanciant des classes disjointes. Nous définissons la règle 3 :

Règle 3 *: Soit I une instance appartenant à une classe C. I ne peut pas appartenir à une classe disjointe C', mais une instance I', générée à partir de I, peut appartenir à C' (et pas à C).*

5.5 CONTEXTE DE VIE

La classification a produit plusieurs instances rattachées à des classes et modélisant les connaissances connues ou inférables d'un objet du monde réel. L'utilisateur choisit une ou plusieurs instances qu'il juge les plus représentatives. Pour chaque instance, les classes de rattachement proposées par la classification sont les classes possibles les plus spécialisées (Figure 5-9). Elles constituent le "contexte de vie" maximal de l'instance. Ces classes n'étant pas forcément les plus significatives, l'utilisateur peut choisir un contexte de vie plus restreint pour chaque instance choisie. Un contexte de vie d'une instance est constitué d'un sous ensemble des classes possibles, au sens de la multi-instanciation (invariant I.I.5). Toutes les combinaisons ne sont cependant pas possibles. En effet, pour respecter l'invariant I.I.5 l'instance doit, en particulier, pouvoir justifier ses valeurs. Elle doit donc appartenir directement, dans le cas des attributs à valeur dépendante, ou indirectement dans celui des attributs à valeur unique, aux classes où ses valeurs ont été inférées.

Si une classe C est possible pour I, alors une super-classe C' de C est un contexte de vie possible pour I, si I peut justifier ses valeurs dans C'. Dans le cas où C' est impossible pour I alors il existe une autre instance I', génératrice de I, qui est rattachée à C'. Dans la Figure 5-9, la classe *Aéronef_ailé* est impossible pour *avion3* mais possible pour *avion1*, car l'autonomie

est un attribut sans valeur dans la classe *Aéronef_ailé*. De même, si deux classes (C1 et C2) constituent un contexte de vie possible pour I, alors C1 est un contexte de vie possible pour I, si cette instance justifie ses valeurs dans C1. La classe *Avion_tourisme* est un contexte de vie possible pour *avion2* (Figure 5-9).

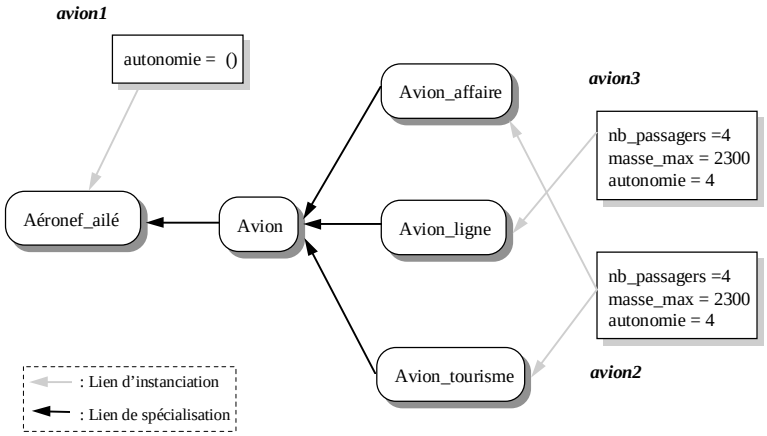
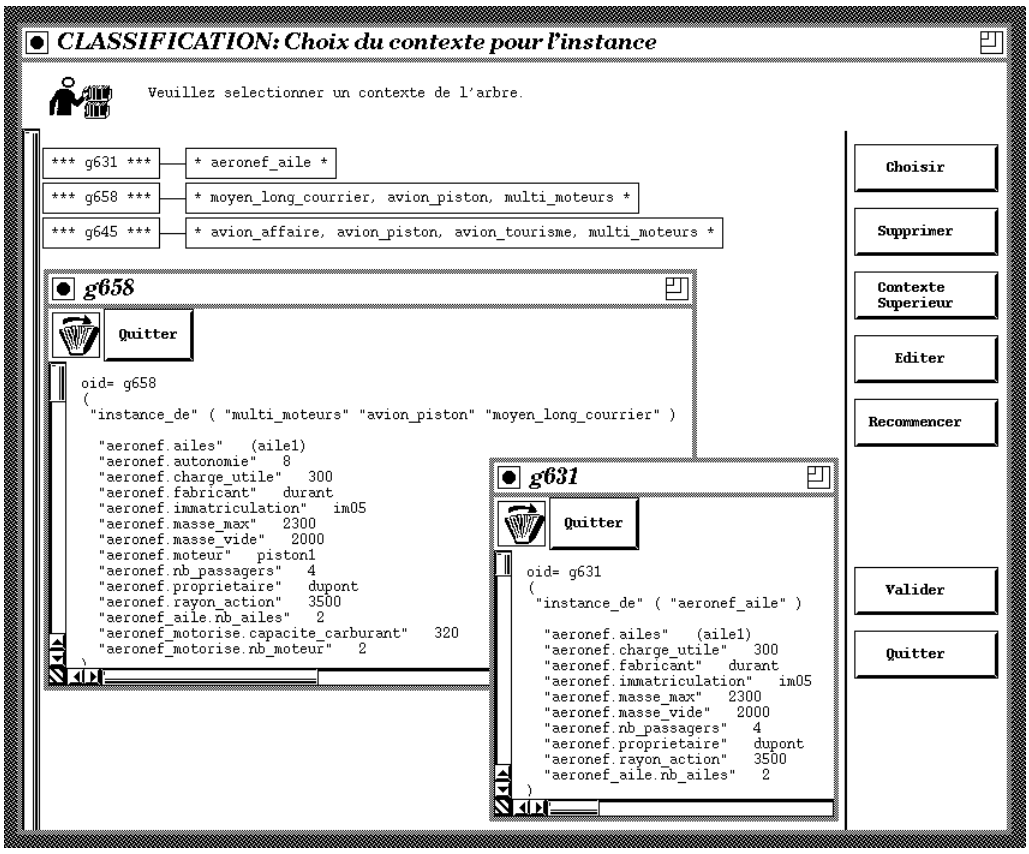


Figure 5-9 : Contexte de vie



Ecran 5-1 : Choix d'un contexte de vie

L'ensemble des instances générées par le mécanisme, lors de la classification d'un objet du monde réel, sont des objets temporaires qui correspondent aux différentes solutions d'un même problème initial. Le choix final reste donc à la charge de l'utilisateur, qui choisit

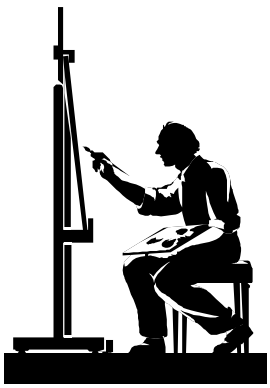
l'ensemble des instances parmi toutes celles générées (les autres instances sont détruites) et un contexte de vie pour chaque instance [Liotard93] (Ecran 5-1).

5.6 CONCLUSION

Dans ce chapitre, nous décrivons un mécanisme de classification d'instances prenant en charge l'insertion et l'évolution simultanée des différentes représentations d'objets incomplets, incohérents et multi-inférables. L'objet à classifier est complété en utilisant les inférences existantes dans le graphe de recherche. Du fait de la multiplicité des inférences, la classification propose plusieurs solutions matérialisées par des instances différentes.

Les connaissances initiales détenues sur l'objet peuvent être assimilées à l'énoncé d'un problème à résoudre. La classification peut alors être perçue comme un mécanisme de résolution de problème délivrant plusieurs solutions (les différentes instances provisoires). Ce mécanisme de résolution utilise des connaissances inhérentes au domaine d'application (la hiérarchie de classes) et des connaissances inhérentes à leur utilisation dans le contexte applicatif (attribut obligatoire/facultatif, à valeur unique/ dépendante, etc.). Pour l'instant, dans le modèle SHOOD, les connaissances du domaine d'application et celles inhérentes à leur utilisation sont confondues. Nous envisageons de les séparer afin de permettre une meilleure réutilisation des connaissances.

Une extension possible du mécanisme présenté dans ce chapitre est la classification des objets composites [Marino93]. Cette extension nécessitera un effort de programmation pour la modification du code actuel. En effet, ce mécanisme est aujourd'hui "câblé", et n'offre pas la possibilité d'être étendu facilement. Cet inconvénient constitue une des motivations pour étudier l'apport de solutions aux problèmes d'extensibilité des mécanismes d'évolution existants. Pour cela, dans le chapitre 7, nous présentons un mécanisme de règles actives qui tente de résoudre les problèmes d'extensibilité, en offrant une expression déclarative de l'évolution. Nous présentons tout d'abord dans le chapitre qui suit (Chapitre 6) quelques modèles à base de règles qui nous ont servi à définir ce mécanisme de règles actives.



6. LES MODELES A BASE DE REGLES

Un système actif est un système capable de détecter des situations et de réagir avec ou sans l'intervention de l'utilisateur en exécutant des actions ou des programmes. Les réactions d'un système sont généralement réalisées par des déclencheurs ou règles actives [Dayal88]. La notion de règle active est basée sur le formalisme Evénement, Condition, Action. La sémantique générale d'une règle active est la suivante "Lorsqu'un Evénement se produit, si la Condition est satisfaite, alors l'action est exécutée".

L'objectif de ce chapitre n'est pas de présenter une liste de Systèmes à Objets Actifs (SOA), mais plutôt de justifier les choix des différents concepts du modèle actif de SHOOD. Nous verrons que le système actif (cf. chapitre 7) permet la mise en œuvre du mécanisme de règle et de stratégies d'évolution (cf. chapitre 8). Par la suite, nous ne citons que les systèmes de référence comme HIPAC, et les systèmes comme URDOS [Tchnoukine93], et NEOPUS [Pachet92] dont nous nous sommes inspirés pour la conception de notre modèle actif. D'autres systèmes [Collet94] [Bouaziz95] [Roncancio94] [Riet89] [Simon92] modélisant des règles actives ne seront pas décrits en raison de la similitude de leur modèle avec celui d'HIPAC.

6.1 CONCEPTS FONDAMENTAUX

La notion d'activité est apparue pour accroître la capacité de réaction des Systèmes de Gestion de Bases de Données SGBD ; elle a donné naissance au concept d'objet actif. Dans les systèmes à objets, en particulier dans les SGBD Orientés Objets (SGBDOO), chaque objet attend de recevoir un message, indiquant l'opération demandée. Dans ce cas, l'objet est dit passif. L'activité permet de spécifier pour un objet (dit actif) les moments de déclenchement de ses méthodes. Ces moments sont identifiés par des événements. L'objet actif n'a plus besoin d'attendre un message, il réagit de lui-même à son environnement en fonction des événements qu'il détecte.

L'activité est apparue essentiellement à travers les concepts de déclencheurs (triggers en anglais) ou de règles "*actives*". Une règle active est définie en général par trois composants :

un événement, une condition et une action. Elle est aussi appelée règle ECA (Evénement, Condition, Action) [Dayal88].

L'activité d'un objet modélise un comportement interne ou externe. Lorsqu'un objet actif détecte un événement, il doit réagir. Cette réaction le concerne souvent lui-même, il réagit par l'exécution de ses propres méthodes ; c'est le comportement interne. Mais l'objet peut aussi stimuler le comportement d'autres objets, par l'intermédiaire du mécanisme d'envoi de messages ou en déclenchant explicitement d'autres événements ; c'est le comportement externe. Ce qui permet de modéliser un comportement inter-objets.

Deux aspects fondamentaux composent un système actif : (1) le modèle de règles qui permet d'exprimer l'activité d'un objet et (2) le modèle d'exécution qui permet l'exécution de cette activité.

(1) L'activité d'un objet est exprimée par des règles actives, dont la composition varie selon les systèmes. En général, elle est définie par trois composantes : l'événement, la condition et l'action. Certaines de ces composantes sont soit implicites, soit absentes. L'événement, qui indique le moment du déclenchement des règles, peut être du type primitif : *Bases de Données* (modification de la base, fin de transaction, etc.), *temporels* ou *externes* (déclenchés par l'application). De même, il peut être de type composite défini en appliquant des opérateurs de disjonction, composition, séquence, fermeture etc. (cf. § 6.2) [Chakravarthy94]. La condition permet de raffiner le moment de l'exécution de la règle. Celle-ci est exécutée si la condition est satisfaite. Dans certains systèmes, cette composante est soit absente, soit intégrée dans la composante événement [Tchounikine93] [Gehani91]. Quant à l'action, elle correspond à un bloc d'instructions dans lequel on peut déclencher d'autres règles.

(2) L'exécution d'une règle est généralement prise en charge par un moteur d'exécution, qui a pour but d'évaluer les conditions et d'exécuter les actions des règles sélectionnées lors de la détection d'un événement. Dans les SGBD, l'évaluation de la condition et l'exécution de l'action s'effectuent à l'intérieur d'une transaction [Atkinson89] ; Le modèle d'exécution est lié au support transactionnel du système.

La détection d'un événement peut se faire soit au niveau des programmes (événements utilisateurs) soit au niveau du gestionnaire des opérations sur la base (appels de méthodes, fin de transaction). Si on construit l'ensemble des règles référençant l'événement détecté, on obtient ainsi l'ensemble des règles candidates à l'exécution. L'ensemble des règles sélectionnées est construit en éliminant les règles désactivées, et en adoptant une stratégie d'ordonnancement de ces règles (selon la priorité, séquentiel, aléatoire etc.).

Le modèle d'exécution détermine la politique des traitements des règles sélectionnées. Plusieurs stratégies d'exécution existent en fonction des traitements des événements et des différents modes d'exécution ou mode couplages de la condition par rapport à l'événement et celui de l'action par rapport à la condition. La condition d'une règle est évaluée après la détection de l'événement (mode immédiat) ou bien à la fin de la transaction où l'événement est détecté (mode différé). L'évaluation de la condition peut aussi être effectuée dans une autre transaction séparée de celle où l'événement est déclenché (mode séparé) [Dayal90]. Nous retrouvons ces mêmes modes pour l'évaluation de l'action. Ces différents modes de couplages sont résumés dans le tableau suivant.

		Couplage C-A		
		Immédiat	**Différé**	**Séparé**
Couplage E-C	**Immédiat**	C et A sont évaluées après E.	C est vérifiée après E. A est exécutée à la fin de la transaction.	C est vérifiée après E, A est exécutée dans une transaction séparée.
	Différé	Non autorisé	C est vérifiée et A exécutée à la fin de la transaction.	C est vérifiée à la fin de la transaction et A est exécutée dans une transaction séparée.
	Séparé	C est vérifiée et A exécutée dans une transaction séparée.	Non autorisé	C est testée et A est exécutée dans une transaction séparée.

Tableau 6-1 : Les modes de couplage possibles

Les systèmes à objets actifs que nous avons étudiés sont tous des SGBDOO Actifs. Certains modélisent leur activité par des déclencheurs (triggers) [Beeri91] [Gehani91], d'autres par des règles ECA [Medeiros90] [Diaz91] [Dayal88] [Tchounikine93] [Gatzui92] [Chakravarthy94] [Collet94] etc. Les premiers ne sont pas présentés dans cet état de l'art, leurs principes étant similaires à ceux des règles ECA. Ils seront cités dans le bilan pour les comparer aux autres modèles. Les triggers de [Beeri91] sont des mécanismes qui permettent d'associer des actions à des événements ; la composante *Condition* n'existe pas. Les triggers (et contraintes) de ODE [Gehani91] permettent de spécifier des règles de la forme Condition-Action, où la composante *Evénement* est implicite (c'est la modification d'un objet).

Dans la présentation des modèles, on s'intéressera plus au modèle de règles qu'au modèle d'exécution vu que notre objectif est de proposer un modèle actif pour un système de représentation de connaissances où la notion de transaction [Delobel91], contrairement aux SGBD, est moins importante. Nous présentons en premier HIPAC, qui est le modèle actif ayant servi de référence aux autres modèles, et qui est l'un des plus complets et des plus anciens. Ensuite, nous présentons URDOS qui, à l'opposé d'HIPAC, est récent. Puis, nous achevons cet état de l'art par la présentation des particularités d'autres travaux [Diaz94] [Medeiros90] sur

les modèles actifs à base de règles ECA, et par la présentation d'un système à base de règles de production NEOPUS [Pachet92] dont nous nous sommes inspiré pour introduire la notion de "Bases de règles" dans notre modèle.

6.2 HIPAC

HIPAC (HIgh Performance ACtive database) [Dayal88] était un projet mené au laboratoire de recherche Xerox (Cambridge, USA). Son modèle de règles peut être ajouté à tout SGBDOO ; le prototype a été bâti au-dessus du SGBD PROBE.

6.2.1 Le modèle de règles

Les règles y sont modélisées par des objets. Toutes les règles sont instances d'une classe unique qui décrit la structure et le comportement des règles. Cette classe peut être spécialisée pour créer des règles plus spécifiques. Les attributs d'une classe règle sont les suivants : l'événement, la condition, l'action et le mode de couplage.

L' événement

L'événement est défini par un identificateur et une liste de paramètres. Ces paramètres permettent à l'action et à la condition d'accéder au contexte de déclenchement de l'événement. HIPAC distingue trois types d'événements primitifs: Base de données, Temporels et Abstraits.

- Les événements ***Base de données*** sont associés aux opérations exécutées sur la Base de données (BD). Ces opérations peuvent être réalisées par le système (modification de la base, transaction, ...) ou par l'utilisateur (fonctions définies par l'utilisateur).

Une opération n'étant pas instantanée, pour chacune d'elles deux événements peuvent être définis, l'un correspondant au début de l'opération, l'autre à la fin de celle-ci. L'identificateur de l'événement est alors l'identificateur de l'opération préfixé de *Beginning* ou *End*. Pour les opérations système, la liste des arguments est définie par les concepteurs d'HIPAC. Tandis que pour les fonctions utilisateur, les arguments de l'événement *Beginning* sont les paramètres d'entrée de la fonction et les arguments de l'événement *End* sont les paramètres de sortie.

Par exemple, l'événement E1 ci-dessous est déclenché lors de l'appel de la fonction *Update_position*. Ses arguments sont *T* la cible qui va changer de position, et *Position* qui indique les coordonnées initiales de la cible. L'événement *E2* est déclenché au retour de la fonction, ses arguments sont les mêmes mais *Position* indique alors les nouvelles coordonnées de la cible.

E1 : Beginning_Update_Position (T : Target, Position(T) : (Lat, Long))

E2 : End_Update_Position (T : Target, Position(T) : (Lat, Long))

- Les événements ***Temporels*** sont déclenchés par l'horloge système. Ils peuvent être désignés de manière absolue (exemple : 9:00:00 a.m. April 10. 1988), relative (exemple : 30 secs after event E occurred) ou périodique (exemple : every day at midnight).

- Les événements ***Abstraits*** sont les événements qui ne peuvent pas être détectés par le système HIPAC. Leur identificateur et leur liste d'arguments sont définis dans le modèle, mais ils sont déclenchés par l'utilisateur ou un programme annexe, au moyen de l'appel de fonction *signal*.

Ces trois types d'événements représentent les événements primitifs. HIPAC permet aussi l'expression d'événements composés grâce aux opérateurs de disjonction, de séquence et de fermeture.

- L'opérateur de disjonction "|" (Exemple: $E_1|E_2$) permet de spécifier la sélection d'une règle soit lors de la détection d'un événement E_1, soit lors de celle d'un événement E_2.

- L'opérateur de séquence ":" (Exemple : $E_1:E_2$) permet d'indiquer la sélection d'une règle lorsque, durant une même transaction, les événements E_1 et E_2 sont détectés suivant l'ordre spécifié.

- L'opérateur de fermeture "*" (Exemple: E_1*) permet d'indiquer qu'une règle qui référence un événement E_1 avec l'opérateur de fermeture, ne sera sélectionnée qu'en fin de transaction . Si, lors d'une transaction, l'événement E_1 est déclenché *n* fois, ses arguments effectifs seront stockés séquentiellement. A la fin de cette transaction, la règle utilise la séquence des *n* listes d'arguments effectifs.

Les autres composants d'une règle active dans HIPAC sont :

La condition

La condition est exprimée par une conjonction de requêtes à la base de données . Elle est satisfaite si toutes les requêtes retournent une réponse non-vide. L'ensemble des réponses pourra être utilisé par l'action lors de son exécution.

L'action

L'action est soit un programme incluant des opérations sur la BD et des déclenchements d'événements abstraits, soit un message à destination d'un programme externe.

Les couplages E-C, C-A

Au niveau d'une règle, les couplages expriment respectivement le moment de l'évaluation de la condition relativement au déclenchement de l'événement et celui de l'exécution de l'action par rapport à l'évaluation de la condition. Ils peuvent prendre comme valeurs : *immédiat, différé, séparé dépendant, séparé indépendant*.

- Le mode *immédiat* indique l'interruption de la transaction dans laquelle l'événement est déclenché, pour permettre l'évaluation de la condition (ou respectivement l'exécution de l'action).
- En mode *différé*, la condition est évaluée à la fin de la transaction (de même pour l'exécution de l'action).
- En mode *séparé dépendant*, l'évaluation de la condition et l'exécution de l'action sont réalisées dans une autre transaction, si la transaction déclenchante n'est pas interrompue.
- Le mode *séparé indépendant* est identique au précédent ; cependant si la transaction déclenchante est interrompue, la condition est tout de même évaluée et l'action exécutée.

6.2.2 Bilan

HIPAC est la référence des modèles à base de règles ECA par le nombre de concepts qu'il introduit. Certains de ces concepts restent cependant flous. En particulier, la définition de l'opérateur de séquence ne spécifie pas si la séquence est stricte (*E1* suivi immédiatement de *E2)*, ou si des événements peuvent s'intercaler : *E1 E3 ... En E2*. En ce qui concerne l'opérateur de disjonction, Dayal ne précise pas comment il utilise, dans la condition et l'action, les paramètres des deux événements. En effet, un paramètre formel ne peut pas être utilisé si un paramètre effectif ne lui est pas associé. Les modes de couplages sont aussi discutables. Les modes *immédiat* et *différé* sont intéressants. Par contre, étant donné le caractère aléatoire de l'instant où sont réalisées l'évaluation de la condition et l'exécution de l'action, les deux autres modes seront-ils utilisés ? De plus, lors de l'exécution de l'action, les paramètres effectifs et les réponses des requêtes ne sont plus représentatifs de l'état de la BD.

Par contre, le système HIPAC propose un certain nombre de concepts intéressants que nous avons adoptés. En particulier, il a mis en évidence le fait que les événements peuvent avoir des origines multiples et que certains ne sont pas déclenchables par le système (Evénements abstraits). Il a aussi montré que les trois composantes d'une règle ne sont pas indépendantes. Le contexte dans lequel s'est déclenché l'événement peut être utilisé par la condition et l'action. Les calculs et les requêtes réalisés lors de l'évaluation de la condition peuvent être sauvegardés pour ne pas être réitérés lors de l'exécution de l'action.

6.3 *URDOS*

URDOS (Using Rules for a Dynamic Object System) est "un outil générique (portable) permettant d'introduire la composante active dans un SGBD hôte virtuel" [Tchounikine93]. C'est un modèle très récent qui a été étudié et conçu au laboratoire SIG/IRIT de Toulouse

L'objectif de généricité de l'outil a donné naissance à un modèle simple dont les concepts ont des définitions diffèrentes de celles du modèle HIPAC. Le *Schéma Actif* est la partie de URDOS qui a le plus retenu notre attention. Mais avant tout, voyons les définitions des termes E,C,A.

6.3.1 Le triplet <E,C,A>

URDOS dispose de deux types d'événements : les événements *Base de Données* et les événements *temporels*. Si les événements temporels sont semblables à ceux de HIPAC, les événements BD ont une définition très différente.

HIPAC et les systèmes que nous verrons par la suite proposent une modélisation de règles intégrée au système de base. Le mécanisme de déclenchement des événements est alors câblé dans le système. URDOS devant être portable, il n'était pas possible de coder le déclenchement des événements. Cette remarque et d'autres raisons [Tchounikine93] sont à l'origine d'une nouvelle définition de l'événement "BD".

URDOS déclenche un événement "BD" après une mise à jour de la base, lorsque le système fige la modification (au moment du "commit", qui permet de récupérer les identifiants des objets modifiés). URDOS ne déclenche pas un événement sur une opération spécifique, comme l'appel d'une méthode M, mais sur une opération générique qui est la modification de la base. L'outil permet pourtant d'exprimer une infinité d'événements en définissant un événement comme un état spécifique de la base. Les événements sont décrits par un triplet <I,S,CO> où :

- **I** est l'identificateur de l'événement,

- **S** est une requête définissant la source de l'événement. La source indique quels doivent être les objets modifiés pour déclencher cet événement. Elle obéit à la hiérarchie des objets. Si la source spécifie une classe C, elle englobe toutes les instances de cette classe, mais aussi toutes les instances de toutes les sous-classes de C.

- **CO** est la condition d'occurrence de l'événement. Elle spécifie l'état dans lequel doit se trouver la base (en particulier les objets modifiés) pour que l'événement soit déclenché.

Un exemple d'événement :

> E : I : "fin stock"
> S : classe produit

CO : l'attribut "quantité en stock" est en dessous du seuil.

L'événement "fin stock" est évalué à l'issue de toute modification d'une instance de la classe produit. Si la condition d'occurrence est satisfaite, l'événement est déclenché.

La condition est une requête booléenne exprimée dans le langage de requête du SGBD hôte ou le résultat d'une méthode booléenne. Toujours par souci de portabilité, l'action est une série de messages (appels de méthodes). L'objet mis à jour (déclencheur de l'événement) est une donnée commune à la condition et à l'action.

Une règle peut être active/armée ou inactive/désarmée. L'utilisateur peut aussi spécifier une priorité pour chacune d'elles. A cette priorité statique s'ajoute une priorité dynamique qui est calculée lorsque plusieurs événements sont déclenchés simultanément [Tchounikine93].

6.3.2 Le Schéma Actif

Les concepteurs d'Urdos ont constaté que l'activité d'un objet ne lui est pas intrinsèque, mais qu'elle dépend du contexte applicatif. Les règles sont alors représentées par des objets qui sont externes aux classes dont elles modélisent l'activité. Cette modélisation va aussi permettre d'écrire des règles multi-classes tout en libérant l'utilisateur du choix de la classe à laquelle est associée la règle.

URDOS représente l'activité des objets au sein d'un *Schéma Actif* qui est indépendant du schéma applicatif (Figure 6-10). Cette approche permet d'importer la définition des règles et des événements (le Schéma Actif) dans n'importe quel schéma d'application d'un SGBD hôte. Ensuite l'utilisateur pourra créer les règles qui modélisent l'activité de l'application.

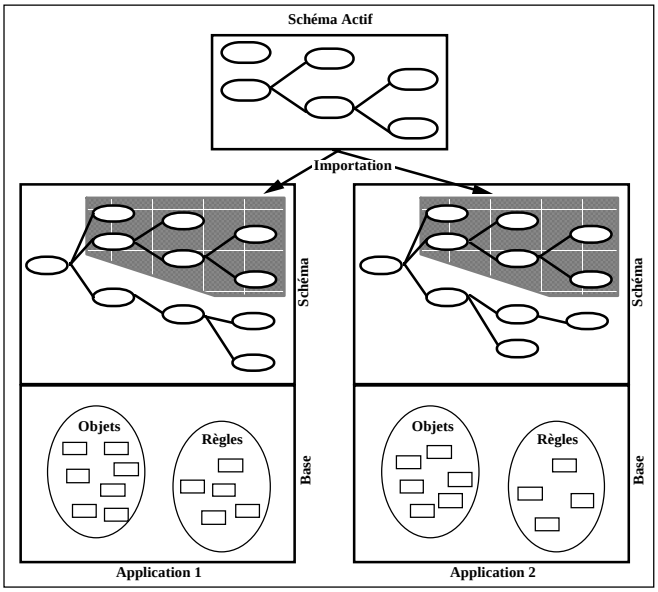


Figure 6-10 : Le schéma Actif et les Schémas applicatifs

Le concept du Schéma Actif présente un avantage majeur, qui est la réutilisation des objets. En effet, lors de la modélisation du comportement actif d'un objet, la structure interne de celui-ci n'est pas modifiée. Cet objet peut alors être réutilisé dans une autre base, indépendamment de son activité.

Le Schéma Actif comprend en particulier une classe *Règle* dont toutes les règles seront les instances. Les événements sont aussi modélisés par des objets, qui peuvent être utilisés dans plusieurs règles. Réifier[6] les événements, permet de les factoriser dans les règles et d'offrir une conception incrémentale de ces règles. En effet, l'utilisateur pourra définir un événement, ensuite les règles qui référencent cet événement. L'ensemble des types d'événements est représenté par une hiérarchie de classes (Figure 6-11).

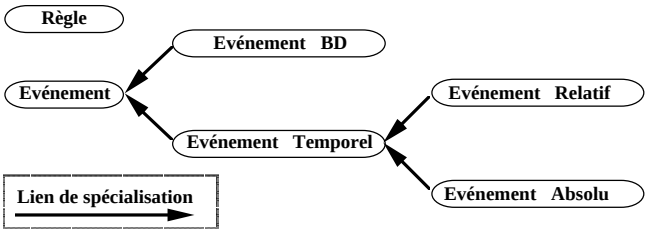


Figure 6-11 : Le schéma Actif

6.3.3 Bilan

Ce modèle est une illustration très simple du concept d'Objet Actif via des règles ECA : pas d'opérateur événementiel, ni de mode de couplage (pour l'instant) et des liens entre les trois composantes <E,C,A> réduits au minimum.

La définition particulière donnée à l'événement "BD" est surtout due au fait qu'URDOS est un outil générique. Cet outil se voulant portable sur tous les systèmes, le déclenchement des événements "BD" ne peut pas être associé aux méthodes, car il nécessiterait un mécanisme interne au système hôte.

Cependant la composante "Condition d'Occurrence d'événement" ("CO") nous paraît discutable. En effet, nous pensons qu'un événement doit correspondre uniquement à une opération sur la base de données, laissant aux règles le rôle de tester le contexte dans lequel s'est produit l'événement. Exprimer une condition dans l'événement ne permet que de factoriser, au niveau de celui-ci, une condition commune à plusieurs règles, ces dernières possédant aussi chacune une condition particulière.

[6] Réifier : action de modéliser un concept par un objet.

Le concept de schéma actif nous paraît intéressant et compatible avec la philosophie de SHOOD, car il permet l'expression de règles multi-classes et la construction de schéma applicatif indépendant de son activité.

6.4 D'AUTRES MODELES A BASE DE REGLES ECA

Dans cette partie nous ne faisons pas une description exhaustive des modèles, mais nous citons les particularités qui les différencient de HIPAC ou URDOS.

6.4.1 Le modèle Exact

Le modèle EXACT est issu de l'introduction du concept d'Objet Actif dans le SGBDOO ADAM [Diaz91] [Diaz94]. Comme dans HIPAC et URDOS, les règles et les événements sont représentés par des objets. Mais ici, les règles suivent le paradigme objet à la lettre, elles sont encapsulées dans les classes dont elles modélisent le comportement actif. Comme les classes sont hiérarchisées suivant une relation de spécialisation, les règles sont elles aussi hiérarchisées. Ainsi les règles des super-classes sont héritées dans les sous-classes.

Une règle est un objet dont deux des attributs indiquent l'événement déclencheur : le nom de la méthode déclenchante (ou opération) et le moment du déclenchement (avant ou après l'exécution de la méthode). Un autre attribut de la règle indique les objets de la classe active (où elle est encapsulée) qui font exception au comportement actif modélisé par la règle. Une règle est sélectionnée lorsqu'un des objets de la classe active reçoit l'événement (l'appel de méthode), sauf si cet objet y fait exception.

Une dernière particularité de EXACT concerne la distinction de trois familles de règles : les règles utilisateur, les règles d'intégrité, les règles de propagation. Les règles de la première famille, comme leur nom l'indique, sont créées par l'utilisateur tandis que les autres sont générées par le système. Les règles d'intégrité correspondent aux contraintes d'intégrité spécifiées déclarativement par l'utilisateur. Les règles de propagation concrétisent la sémantique des relations (liens d'héritage, liens référentiels ...) exprimée dans la base.

6.4.2 Les travaux de Medeiros

Les travaux de Medeiros[7] [Medeiros90] [Medeiros91] ont été implémentés sur le SGBDOO O2. Deux particularités le distinguent des autres : la définition de l'événement, la représentation des règles.

7 Ces travaux ne possédant pas de nom, nous les identifions par le nom de l'auteur servant de référence à la bibliographie. Cette identification n'indique pas que le modèle a été conçu par une seule personne, des co-auteurs figurant dans la bibliographie.

Deux types d'événements existent : les événements déclenchés par le système "d'envoi de messages" et les événements temporels. Le modèle possède l'opérateur événementiel de disjonction. L'expression d'un événement "envoi de messages" spécifie un filtrage sur sa source :

- **E=C** : désigne un message quelconque envoyé à n'importe quel objet de la classe C.
- **E=O** : désigne un message quelconque envoyé à un objet précis O.
- **E=(C,m)** : désigne un message pour la méthode m de la classe C.
- **E=(O,m)** : désigne un message pour la méthode m à destination de l'objet O.

La saisie d'une règle donne naissance à une règle "externe" et à un ensemble de règles "internes". La règle "externe" est l'image de la spécification donnée par l'utilisateur ; elle est stockée sous forme d'objets. Ensuite, suivant l'expression événementielle de la règle, la règle est décomposée en règles "internes". Cette décomposition suit les trois étapes suivantes :

- Si l'expression événementielle est une disjonction d'événements, pour chaque opérande événementiel, une nouvelle règle est créée par duplication.

- Ensuite pour chaque règle dérivée, si l'événement ne précise pas une méthode, la règle est dupliquée pour chaque méthode appartenant à l'objet (ou la classe) spécifié.

- La dernière étape concerne l'héritage des règles. Pour chacune des règles dérivées, si l'événement spécifie une classe C, cette règle est dupliquée dans toutes les sous-classes de C.

L'ensemble des règles dérivées constitue les règles internes. La condition et l'action d'une règle interne sont compilées en une méthode (transparente à l'utilisateur). Cette méthode est ensuite liée à la classe spécifiée au niveau de l'événement de la règle interne. Cette approche présente l’inconvénient de diminuer les performances du système, car la gestion de l’espace des règles internes est complexe.

6.4.3 Sentinel

SENTINEL [Chakravarthy94] est une extension du système Open OODB (Texas Instrument) avec des règles actives de type ECA. Le langage de spécification SNOOP [Anwer93] définit des événements de types primitifs (début et fin de méthode, transactionnel et temporels) et les événements composites formés par des opérateurs classiques tels que la conjonction, la séquence et la disjonction et des opérateurs sur les occurrences des événements (*Periodic Event Operators* et *Aperiodic Operators*).

Afin de constituer le contexte de déclenchement de l'événement, qui peut être composé de plusieurs occurrences d'événements, Snoop propose quatre façons de procéder :

- *recent* : prise en compte de l'événement le plus récent seulement.
- *chronicle* : Les occurrences d'événements sont prises en compte dans l'ordre chronologique.
- *continious* : toute occurrence d'événements est prise en compte comme le début potentiel d'une composition d'événements.
- *cumulative* : tous les paramètres des occurrences d'événements sont pris en compte pour construire un seul contexte d'un événement composite.

Afin d'éviter des mises à jour pendant l'évaluation de la condition, les événements générés pendant l'évaluation de la condition sont ignorés. Les règles et les événements étant modélisés par des objets, la création des règles peut être dynamique.

6.4.4 Samos

SAMOS, prototype en cours de développement à l'université de Zurich, offre la particularité de proposer un langage de spécification d'événements assez puissant [Gatziu92]. Il distingue les événements primitifs tels que les événements avant et après l'appel de méthode, temporels et externes, ainsi que les événements composites avec les constructeurs suivants : disjonction, séquence, conjonction, première occurrence, négation, etc.

Les règles peuvent être définies soit dans les classes (règles internes), soit en dehors (règles externes). Quant au modèle d'exécution, il est presque similaire à celui d'HIPAC.

6.5 LE SYSTEME NEOPUS

NEOPUS [Pachet92] est un système de représentation de connaissances alliant des objets "au sens classique"[8] et des règles de production d'ordre 1. La description qui suit présente une infime partie des concepts de NEOPUS, car seule une partie de la modélisation des règles de production nous intéresse.

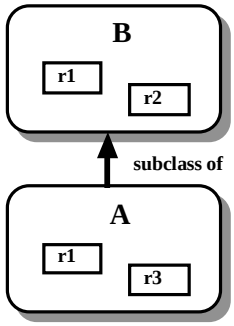
Les règles de production sont formées d'une composante "Prémisse" (condition) et d'une composante "Action" et sont identifiées par un nom. Elles sont compilées sous la forme de méthodes SMALLTALK [Goldberg83] et regroupées dans des bases selon le choix de l'utilisateur. Une base représente l'abstraction d'une certaine connaissance. Les bases sont des classes SMALLTALK.

[8] Les objets respectent l'encapsulation des données et communiquent par envoi de messages.

Le langage SMALLTALK respecte le principe d'encapsulation : les méthodes sont intégrées dans les classes. Les classes sont arrangées dans un graphe d'héritage. Les sous-classes héritent des propriétés de leurs super-classes, en particulier des méthodes. Ces dernières peuvent être surchargées au niveau des sous-classes.

Les bases étant des classes, elles sont aussi hiérarchisées dans un graphe. Les règles sont les méthodes de ces classes. Comme pour les méthodes, les règles des super-bases sont héritées dans les sous-bases et peuvent y être redéfinies.

L'héritage dans les bases de règles est utilisé par le moteur d'inférence. La stratégie de contrôle, nommée HBR (Héritage de Bases de Règles) est la suivante : "*Elle consiste à préférer les règles définies dans la base la plus basse*". Illustrons cette stratégie par un exemple repris de la thèse de Pachet [Pachet92].



Soit A une base de règles, sous-base de B (figure ci-dessus). B définit deux règles r1 et r2. A définit deux règles r1 et r3 (r1.A surcharge r1.B).
Le mécanisme d'héritage et de redéfinition implique que A contient en fait trois règles : r1 (définie dans A), r2 (définie dans B) et r3 (définie dans A)
En cas de conflit, la stratégie HBR va préférer systématiquement les règles de A par rapport à celles de B. Par exemple, si r1 et r2 sont déclenchables, r1.A sera choisie.

6.6 BILAN

L'état de l'art qui est présenté dans ce chapitre est loin d'être exhaustif, d'autres systèmes existent qui proposent des modèles à base de règles. Nous avons préféré présenter les systèmes qui ont réellement apporté des éléments originaux à notre modèle. Mais malgré ces choix, la plupart des systèmes présentent des similitudes :

- Presque tous les systèmes identifient les événements *BD* et les événements *temporels* ; seuls HIPAC et EXACT distinguent l'événement *Abstrait*. Chaque système donne une définition particulière de l'événement BD : Dans HIPAC, un événement correspond à une opération sur la base, pour URDOS, il exprime une condition sur l'état de la base, pour EXACT et les modèles de Medeiros et Beeri [Beeri91], il est associé à une méthode, et pour ODE [Gehani91], l'événement est implicite : c'est la modification de la base de données.

- La partie Condition est souvent une ou plusieurs requêtes sur la base, et l'action un programme à exécuter.

Tous les systèmes ont modélisé les règles par des objets, la plupart ont aussi réifié les événements. URDOS et NAOS [Collet94] se distinguent en implémentant les règles hors des

classes. Pour une règle spécifiée par l'utilisateur, le modèle de Medeiros crée une règle externe et un ensemble de règles internes. Dans EXACT, la définition d'une classe s'enrichit d'un attribut. Pour le modèle de Medeiros, des méthodes correspondant à la compilation de la condition et l'action sont ajoutées à la classe active, tandis que Beeri [Beeri91] modifie les méthodes existantes en leur ajoutant une série d'actions.

Pour le moment, l'opérateur événementiel de fermeture est propre à HIPAC. Par contre, les opérateurs de disjonction, de conjonction et de séquence ont été adoptés par d'autres systèmes, tels que SENTINEL, SAMOS, NAOS, etc. SAMOS et SENTINEL proposent en plus des opérateurs mettant en jeu les occurrences d'événements et les intervalles de temps.

Pour finir, le système NEOPUS nous a permis de faire un lien entre le Schéma Actif de URDOS et la hiérarchie de règles présentée dans des modèles comme EXACT, à travers la notion de base de règles.

7. LE MODELE ACTIF DE SHOOD

Rappelons tout d'abord l'objectif de notre travail qui est de proposer un système d'évolution qui doit permettre, entre autre, une expression déclarative de l'évolution. Tout au long de ce chapitre, les concepts introduits seront illustrés par des exemples d'évolution de classes et d'instances. Nous montrerons à travers ces exemples que les règles ECA constituent un outil permettant une expression déclarative de l'évolution [Bounaas94a] [Bounaas95c]. De plus, nous verrons dans le chapitre suivant que le mécanisme proposé sera réalisé par des règles actives ou règles ECA, définies par le modèle actif présenté dans ce chapitre [Bergues94].

Dans la suite, nous présentons tout d'abord les différents composants et caractéristiques des règles ECA. Nous donnons la définition d'un événement dans SHOOD, et la classification d'événements qui en découle. Nous présentons les composantes E, C et A d'une règle active, ainsi que les liens permettant le passage de valeurs entre ces trois composants. Ensuite, nous montrons que nous pouvons hiérarchiser le comportement actif des objets à travers la notion de base. Après la description conceptuelle du modèle actif, nous abordons sa représentation dans le modèle SHOOD. De ce fait, SHOOD devient un système actif. Nous terminons ce chapitre par la présentation du modèle d'exécution où nous décrivons la stratégie choisie pour le moteur d'exécution.

7.1 LES CONCEPTS DE BASE

7.1.1 L'événement

L'événement dans SHOOD représente une situation due à l'exécution d'une opération [Dayal88]. Cette opération englobe toute action effectuée sur la base, mais aussi celle assurant la communication avec des applications externes à SHOOD. Ces opérations peuvent être :

- soit des méthodes SHOOD, c'est le cas le plus courant. Dans ce cas, les événements dus à l'exécution d'une méthode sont déclenchés implicitement par le système. Comme l'exécution d'une méthode n'est pas instantanée, nous distinguons deux types d'événements méthodes ; l'un déclenché avant l'exécution de la méthode (Appel Méthode) et l'autre après l'exécution (Retour méthode),

- soit des fonctions Lisp, qui en général correspondent à des fonctions d'interface utilisateur ou d'application. Vu que leur exécution est contrôlée par l'interpréteur Le_Lisp, le déclenchement des événements associés est effectué par le programmeur. Ces événements sont identifiés comme des événements utilisateurs. Ils permettront de représenter, par exemple, des événements correspondant à une étape dans l'exécution d'une méthode ou dans l'action d'une règle.

Nous obtenons donc la classification suivante des événements :

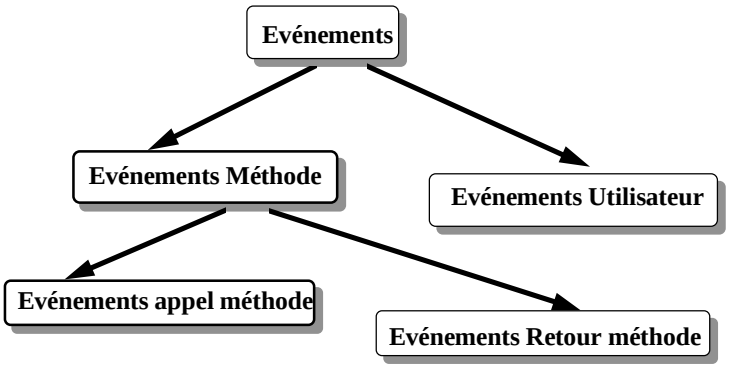


Figure 7-1 : Classification des événements

Un événement de type méthode est identifié par le nom de la méthode associé à l'événement. Mais afin d'assurer l'unicité des noms, le nom d'un événement est préfixé par son type (Appel ou Retour). Pour les événements utilisateur, le nom est fourni par le programmeur, et préfixé par son type (Utilisateur), ceci permettant la définition d'un événement utilisateur de même nom que celui d'une méthode.

Les méthodes dans SHOOD sont hiérarchisées : le comportement d'une méthode est plus spécifique que celui d'une super-méthode. De même, les événements sont hiérarchisés et le déclenchement d'un événement entraîne celui des super-événements. Par souci d'homogénéité, nous avons permis au programmeur de hiérarchiser les événements utilisateurs.

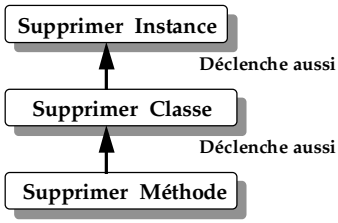


Figure 7-2 : Evénements hiérarchisés

Par exemple, soient les événements méthodes : suppression d'une instance, suppression d'une classe et suppression d'une méthode. Ces événements sont hiérarchisées (figure ci-contre), puisqu'une classe est aussi une instance (instance de "méta"), et qu'une méthode est aussi une classe (instance de "Méta-méthode", qui est sous-classe de "méta"). Donc, la suppression d'une méthode implique bien la suppression d'une classe et d'une instance.

Le contexte de déclenchement d'un événement désigne l'ensemble des objets sur lesquels une opération est exécutée. Il est représenté par un ensemble de paramètres appelé source. Cette source correspond aux paramètres d'entrée pour un appel de méthode et aux paramètres d'entrée et de sortie pour un retour de méthode. Pour les événements utilisateur, ils sont spécifiés par le programmeur. La source va permettre aux règles de spécifier un filtrage par rapport au contexte de déclenchement de l'événement. Elle va aussi permettre à la condition et

à l'action d'utiliser ce contexte. Comme les événements sont hiérarchisés, chaque sous-événement hérite des sources des super-événements.

Pour résumer, un événement est décrit par :

- son type (*Appel*, *Retour*, *Utilisateur*),
- son nom (*nom de méthode ou de fonction*)
- et par la source (*contexte de déclenchement*).

Exemple : *appel supprimer_instance (oid)* est l'événement qui est déclenché avant la suppression d'une instance, sa source est constituée d'un seul paramètre, l'identificateur de l'instance à supprimer (argument de la méthode supprimer_instance).

Le paragraphe suivant définit les différents composants d'une règle ECA et en particulier les expressions événementielles qui permettent l'expression d'un filtrage sur la source.

7.1.2 Les composants d'une règle ECA

7.1.2.1 L'expression événementielle

La composante **E** d'une règle ECA peut être une simple référence à un événement. Elle peut être plus complexe. Un filtrage sur chacun des paramètres de la source de cet événement peut y être spécifié. Ce filtrage peut être typé (filtrage type) et/ou énuméré (filtrage valeurs). Nous désignons par Référence Evénementielle (RE) un énoncé déclaratif spécifiant l'événement déclencheur d'une règle ainsi qu'un filtrage sur la source de celui-ci.

Soit une règle avec sa référence événementielle :

```
R :  E : Appel  Valuer_attribut  (instance  "Rectangle")
                                 (nom_attribut  "base")
     C : .....
     A : .....
```

La RE de la règle R spécifie un filtrage type sur le paramètre instance, qui doit être de type Rectangle et un filtrage valeur sur nom_attribut. Cette règle est déclenchée lors de la valuation (en création ou en modification) de la base d'un rectangle.

La composante **E** accepte aussi une disjonction de RE, qui exprime le fait qu'une règle peut être sélectionnée par le déclenchement de plusieurs événements. L'opérateur de disjonction permet aussi d'enrichir la notion de filtrage (voir exemple ci-dessous). Une règle est dite sélectionnable dès que l'un des événements qu'elle référence est déclenché. Et elle est dite sélectionnée si le filtrage sur la source de l'événement déclenché est satisfait.

Actuellement, la disjonction est le seul opérateur événementiel de SHOOD, et celui-ci est implicite (si plusieurs RE figurent au niveau de la composante **E**, il s'agit d'une disjonction). L'opérateur de séquence d'événements fait partie des extensions envisagées du modèle actif.

Par exemple, la règle suivante sera sélectionnée, si la suppression concerne soit un rectangle, soit un losange.

```
R :  E :    Appel  Supprimer_instance  (instance  "Rectangle")
            Appel  Supprimer_instance  (instance  "Losange")
     C :    ....
     A :    ....
```

Pour résumer, la composante **E** ou l'expression événementielle, permet à une règle d'être sélectionnable selon plusieurs événements et de spécifier un filtrage sur la source de ces événements.

7.1.2.2 L'action et la condition

La composante **C** (Condition) d'une règle ECA, est une expression dont l'évaluation retourne une valeur booléenne. Cette expression peut être un appel de méthode ou de fonction Lisp, tandis que la composante **A** (Action) est un ensemble d'opérations à exécuter, ces opérations étant des appels de méthodes ou de fonctions. Dans le cas d'une règle de vérification (cf §7.1.3), l'action est un message d'interruption de l'opération déclenchante.

D'autres informations sont associées aux règles, telles que les paramètres d'entrée et les variables locales. Les paramètres d'entrées permettent à l'action et à la condition d'accéder au contexte de déclenchement de l'événement (la source). Les variables locales permettent à l'action d'accéder aux résultats des opérations exécutées au niveau de la condition, ce qui évite la réitération de ces opérations dans l'action.

Le mode de couplage dans le modèle actif est immédiat (cf. Chapitre 6). L'évaluation de l'action suit immédiatement celle de la condition, et l'évaluation de la condition suit le déclenchement de l'événement. Cette définition sera remise en cause dans le cas où plusieurs règles sont sélectionnées (cf. §7.4.3). Du fait de l'absence actuelle des transactions dans le modèle SHOOD [Boulenger93] [Roche95], ce mode de couplage est unique, donc implicite.

7.1.3 Deux familles de règles ECA

A partir des objectifs cités (contrôle d'intégrité, évolution des objets et des schémas etc.), nous avons identifié deux types de règles : les **Règles de vérification** où l'action est un message d'erreur qui, si la condition est satisfaite, interrompt l'opération déclenchante ; et les **Règles de propagation** où l'action est un programme permettant généralement, lorsque la condition est satisfaite, de propager les effets de mise à jour à travers la base de connaissances.

Soient la règle de vérification R1 et la règle de propagation R2 :

```
R1 :E : Appel  Supprimer_classe (oid)
     C :  /* teste si la classe est utilisée par un domaine d'attribut*/
     utilisé_par? (oid "Domaine")
     A : Erreur (" la classe ne peut  être supprimée ").

R2 :E : Appel  Supprimer_classe (oid)
     C:  Vrai
     A : /* supprimer les instances de la classe*/
          Supprimer_les_instances  (classe)
```

Si la règle R2 est exécutée avant la règle R1, la condition de la règle R1 est toujours satisfaite. On supprimera alors les instances de la classe sans supprimer la classe. Par contre, si R1 est exécutée avant et si elle déclenche le mécanisme d'erreur, la règle R2 ne sera pas exécutée.

Cette distinction n'est pas un simple classement de règles, mais correspond bien à une stratégie du moteur d'exécution. Lors du déclenchement d'un événement, l'ensemble des règles sélectionnées sera divisé en deux sous-ensembles : les règles de vérification et les règles de propagation. Les règles de vérification seront exécutées avant les règles de propagation. Cette stratégie est due au fait que le modèle SHOOD ne dispose pas actuellement de la notion de transaction. Aucun mécanisme ne permet de défaire ce qui a été fait. Il est alors tout à fait normal d'exécuter en premier les règles de vérification, et si aucune contrainte n'a été violée, d'exécuter ensuite les règles de propagation. Cette stratégie évite l'exécution inutile des règles de propagation.

7.2 BASES DE REGLES POUR ORGANISER L'ACTIVITE

7.2.1 Non encapsulation des règles

Une règle multi-classe est une règle qui modélise le comportement d'objets issus de classes différentes. Comme pour les méthodes, les règles peuvent être multi-classes car il est souvent difficile de choisir la classe d'appartenance d'une règle.

On désigne par schéma applicatif, la modélisation de l'application dans le système SHOOD et par schéma actif l'ensemble des règles, événements et bases permettant de rendre actifs les objets du schéma applicatif. La modélisation du comportement d'un objet hors de sa classe permet une meilleure réutilisation des classes : les classes d'un schéma applicatif peuvent être réutilisées dans d'autres schémas actifs.

Les règles sont donc modélisées hors des classes des objets actifs. Mais afin de permettre une meilleure exploitation des règles, celles-ci seront regroupées dans des bases hiérarchisées.

7.2.2 Regroupement des règles

Afin de mieux organiser l'ensemble des règles, nous proposons de les regrouper dans des bases, que l'on pourra hiérarchiser. Le concept de "Bases" inspiré de NEOPUS [Pachet 92] permet de partitionner et de hiérarchiser un ensemble de règles. Aucun contrôle n'est effectué sur le regroupement des règles dans les bases. L'utilisateur est libre de choisir le thème associé à une base. L'un des critères consiste à regrouper les règles selon les objets actifs dont elles modélisent le comportement.

SHOOD disposant d'un mécanisme permettant de hiérarchiser les concepts, nous avons alors choisi de modéliser les bases de règles par des classes. A la notion de classes et sous-classes, nous faisons correspondre la notion de bases et sous-bases. La relation de spécialisation a pour sémantique l'héritage et la redéfinition des règles. Une base possède des règles qui lui sont propres et des règles héritées de ses super-bases : ainsi une sous-base possède un comportement plus spécifique que ses super-bases. Lors du choix de regroupement des règles, l'utilisateur va devoir tenir compte de cette hiérarchie. Le thème associé à une sous-base doit être plus spécifique que ceux associés aux super-bases de celle-ci (Figure 7-3)

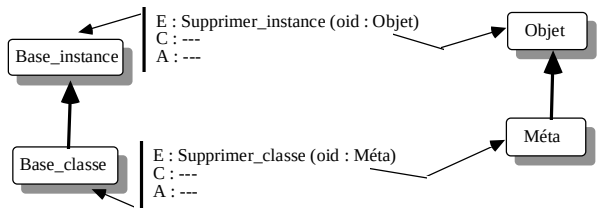


Figure 7-3 :Regroupement des règles dans des bases

Une classe étant une instance d'une méta-classe (*Méta* ou une de ses sous-classes), les règles traduisant la suppression d'une classe doivent être regroupées dans une base plus spécifique que celle regroupant les règles traduisant la suppression d'une instance (instance de *Objet* ou une de.ses sous-classes).

7.2.3 Héritage des règles

Une base B possède des règles qui lui sont propres et des règles héritées de ses super-bases si elles ne sont pas redéfinies par des règles propres à la base B. Nous parlons de redéfinition de règle dans le cas où une base possède une règle ayant le même nom qu'une règle appartenant à une super-base.

SHOOD autorise l'héritage multiple, ainsi une base peut posséder plusieurs super-bases directes. Afin de résoudre les problèmes de conflits d'héritage des règles, on applique le même procédé que pour les attributs, à savoir la notion de nom complet (cf. Chapitre 3). Nous identifions une règle en préfixant son nom par le nom de la base où elle est définie. De ce fait, quand une base hérite de deux règles de même nom, issues de deux bases différentes, elle garde les deux règles (noms complets différents). Si une ambiguïté apparaît, lors d'un héritage

multiple, entre deux règles dont l'une redéfinit l'autre, seule la plus spécifique (définie dans une base plus spécialisée) est héritée. Illustrons l'héritage multiple par l'exemple suivant :

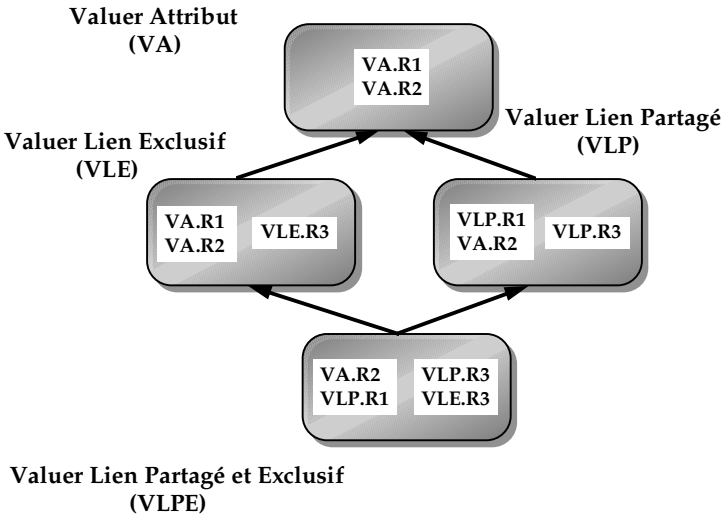


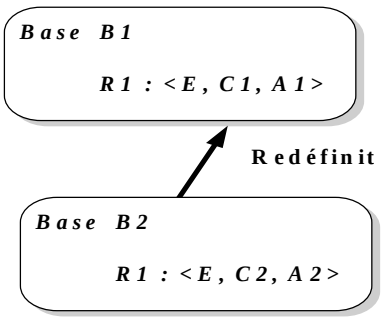
Figure 7-4 : Héritage des règles dans un graphe de bases

Dans SHOOD, un attribut est vu comme une relation sémantique entre deux objets. Un lien de dépendance (partagé ou exclusif) est modélisé par un attribut avec un comportement spécifique (cf. Chapitre 3). La base VA définit deux règles R1 et R2 pour gérer la valuation d'un attribut quelconque, la base Valuer Lien Partagé (VLP) redéfinit R1 (VLP*R1) et définit une règle R3 (VLP*R3) et VLE définit une règle R3 (VLE*R3). Dans la base qui gère simultanément les liens partagés et exclusifs (VPEL), on hérite de VLP*R1, plus spécifique que VLE*R1 ; on hérite des deux règles R3 de VLE et VLP (VLE*R3 et VLP*R3). Au cas où une règle R3 aurait été définie dans VLPE, les deux règles VLE*R3 et VLP*R3 n'auraient pas été héritées. Et si VLE avait surchargé R1, VLPE aurait hérité des deux règles VLE*R1 et VLP*R1 (aucune n'est plus spécifique que l'autre).

7.2.4 L'inhibition des règles

Dans le § 7.1.3, nous avons défini une forme d'inhibition entre les règles de vérification et les règles de propagation. Dans ce paragraphe, nous verrons une autre forme d'inhibition entre les règles de même famille. Une règle d'une base qui redéfinit une règle de même famille d'une super-base inhibe celle-ci dans le cas où elles sont sélectionnées simultanément et que leurs conditions sont satisfaites. Au lieu de dire qu'une règle en redéfinit une autre, nous dirons qu'elle l'inhibe ; nous parlerons de lien d'inhibition entre les deux règles.

Pour les règles de propagation, un lien d'inhibition représente l'inhibition de l'action de la règle inhibée. En d'autres termes, si la condition d'une règle inhibitrice n'est pas satisfaite, le moteur exécutera l'action de la règle inhibée si cette dernière a sa condition satisfaite (voir Figure 7-5). Pour les règles de vérification, le lien d'inhibition permet d'indiquer que le message d'erreur de la règle inhibitrice est prioritaire sur celui de la règle inhibée.



Dans l'exemple ci-contre, B2*R1 inhibe implicitement B1*R1. Si la condition C2 est satisfaite, alors A2 est exécutée. Et A1 ne sera exécutée que si l'évaluation de C2 est fausse et de C1 vraie.

Figure 7-5 : Inhibition des règles

Ces deux sémantiques entraînent le principe suivant : une règle de propagation ne peut inhiber que d'autres règles de propagation, et une règle de vérification ne peut inhiber que d'autres règles de vérification.

La redéfinition d'une règle établit un lien d'inhibition implicite ; on peut aussi inhiber explicitement des règles de noms différents. Un lien d'inhibition ne peut être inhibé qu'en modifiant le nom de la règle. A partir des liens d'inhibition, nous reformulons le principe d'héritage multiple des règles dans les bases : Une base hérite de l'ensemble des règles les plus spécifiques de ses super-bases, si elles ne sont pas inhibées par une autre règle au niveau de la base elle-même.

Exemple

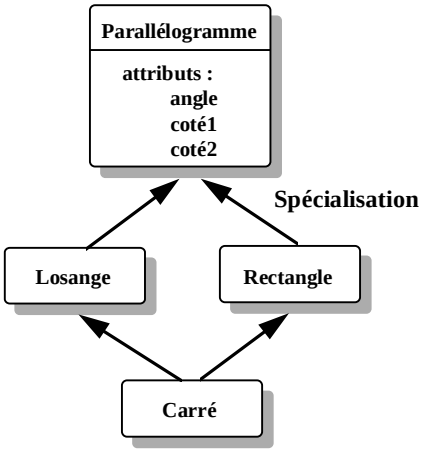


Figure 7-6 : Une hiérarchie de parallélogramme

Dans ce qui suit, nous décrivons un exemple de classification automatique d'un parallélogramme dans la hiérarchie ci-contre. Cette classification est représentée par un ensemble de règles définies dans une hiérarchie de bases.

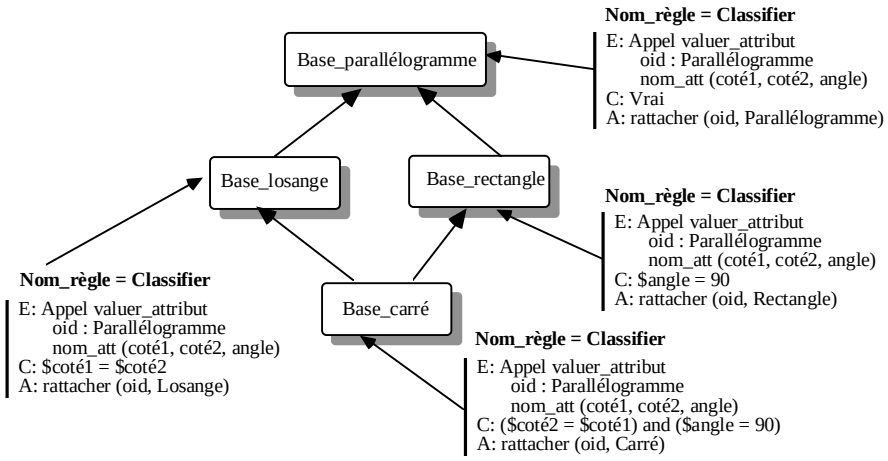


Figure 7-7 : Classification d'un parallélogramme

Les 4 règles ont la même référence événementielle, donc elles seront sélectionnées simultanément. De plus, ces 4 règles ont le même nom (Classifier) donc elles sont liées par des liens d'inhibition ; Base_parallélogramme*Classifier est la plus générale et Base_carré*Classifier la plus spécifique. La règle Base_carré*Classifier sera évaluée en premier ; si sa condition est satisfaite, le parallélogramme sera rattaché à la classe Carré et les autres règles seront inhibées. Dans le cas contraire, les règles Base_losange*Classifier et Base_rectangle*Classifier seront évaluées et si elles sont insatisfaites le parallélogramme sera forcément rattaché à la classe Parallélogramme. La règle Base_parallélogramme*Classifier ne possède pas de condition.

Après avoir défini les règles ECA et les événements dans SHOOD et avoir hiérarchisé le comportement actif des objets à travers un graphe de bases, nous allons maintenant décrire la modélisation des événements, des règles et des bases.

7.3 LA MODELISATION DU SCHEMA ACTIF

Nous désignons par schéma actif l'ensemble des règles, bases et événements permettant de décrire l'activité d'un schéma applicatif (cf. 7.2.1). Dans cette section, nous décrivons la modélisation des concepts du schéma actif. Pour cette modélisation, nous utilisons les concepts du modèle SHOOD, en particulier, le niveau méta.

7.3.1 Modélisation

7.3.1.1 Les événements

Les événements sont hiérarchisés ; ils sont modélisés par des classes, instances de la méta-classe *Méta_événement* (Figure 7-8). La racine des événements est la classe *Evénements* ; elle

possède trois sous-classes qui sont les racines respectives des graphes des événements de type appel méthode, retour méthode et utilisateur.

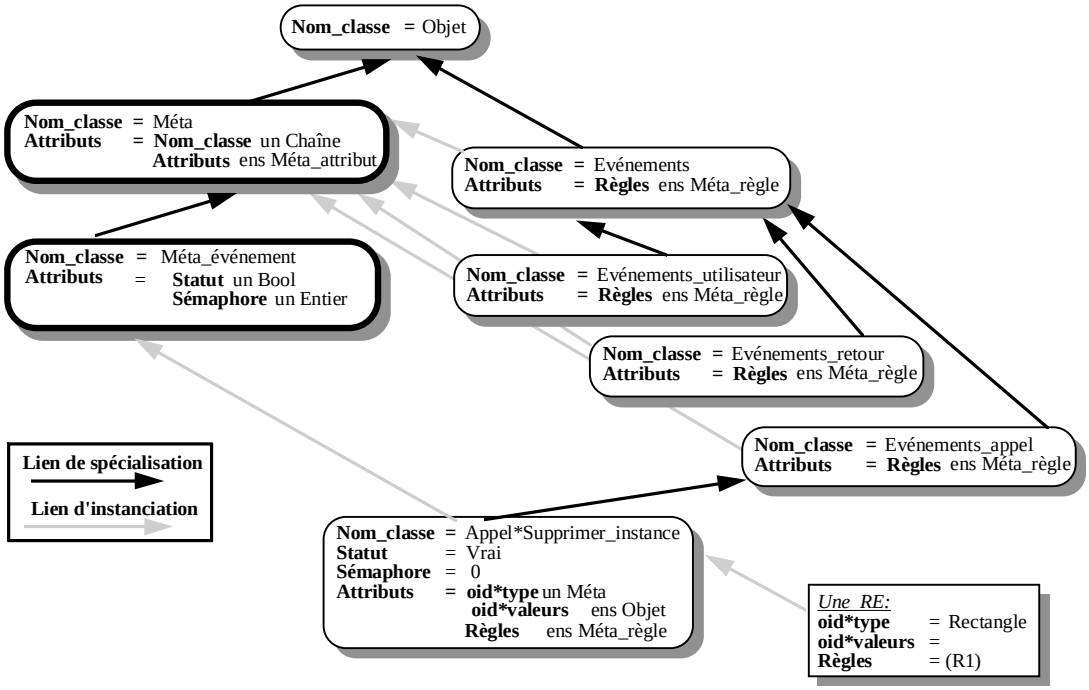


Figure 7-8 : Modélisation des événements

Les attributs d'une classe événement représentent la source de cet événement, et leurs instances sont des références événementielles (RE) (cf. § 7.1.2.1). Pour une RE, les valeurs des attributs désignent son filtrage. Comme deux filtrages (cf. § 7.1.2.1) sont possibles pour chaque paramètre de la source, à un paramètre correspond deux attributs, l'un pour renseigner le filtrage sur le type, l'autre pour renseigner le filtrage sur l'énumération des valeurs. C'est ainsi que dans la classe-événement Appel*supprimer_instance, on retrouve deux attributs (*oid*type* et *oid*valeurs*) pour la source *oid* (Figure 7-8). On retrouve aussi, un attribut "Règles" défini dans la classe *Evénements* et valué pour chacune des RE. Cet attribut permet au moteur d'exécution de connaître rapidement les règles à sélectionner lorsque le filtrage d'une RE est satisfait.

7.3.1.2 Les règles

Les règles sont modélisées par des classes instances de la méta-classe Méta-règle. Les attributs de cette méta-classe sont, entre autres, l'expression événementielle, qui n'est autre qu'un ensemble de RE, la condition, l'action, le statut de la règle (active ou inactive), les paramètres d'entrée, les variables locales et l'attribut *Inhibe*, qui indique les règles qui sont inhibées (implicitement et explicitement) (Figure 7-9).

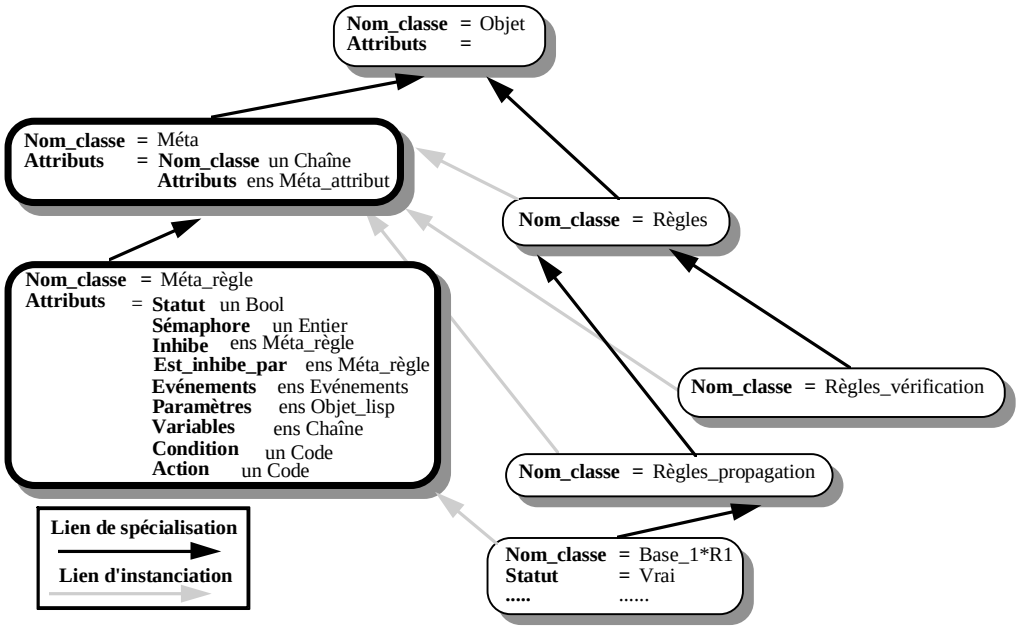


Figure 7-9 : Modélisation des règles

La modélisation des règles par des classes offre deux avantages : les instances d'une classe-règle permettent d'une part de sauvegarder le contexte d'exécution à travers les variables locales et les paramètres d'entrée, et d'autre part de garder une trace de l'exécution des règles dans la perspective d'expliquer le déroulement de l'exécution.

7.3.1.3 Les bases

Les bases sont modélisées par des classes ; elles sont instances d'une nouvelle méta-classe *Méta-Base*, sous-classe de *Méta*. Cette dernière possède un attribut *Règles* qui permet d'indiquer les règles définies au niveau de chaque base. Les classes-bases ne possèdent ni attributs propres, ni instances ; elles permettent seulement de hiérarchiser le comportement actif. La classe *Bases* est la racine du graphe des bases (Figure 7-10).

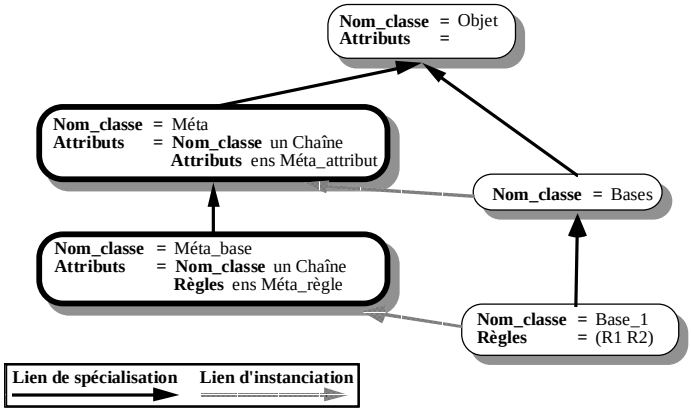


Figure 7-10 : Modélisation des bases de règles

7.3.2 Le noyau du système actif

Le graphe de classes et de méta-classes décrits ci-dessous (Figure 7-11) représent le système actif minimal permettant de représenter le comportement actif d'une application. Remarquons que **la modélisation de ce noyau n'a modifié aucun concept de SHOOD déjà existant**. Le comportement actif des objets est regroupé dans des bases, et non au niveau des classes ou des méthodes.

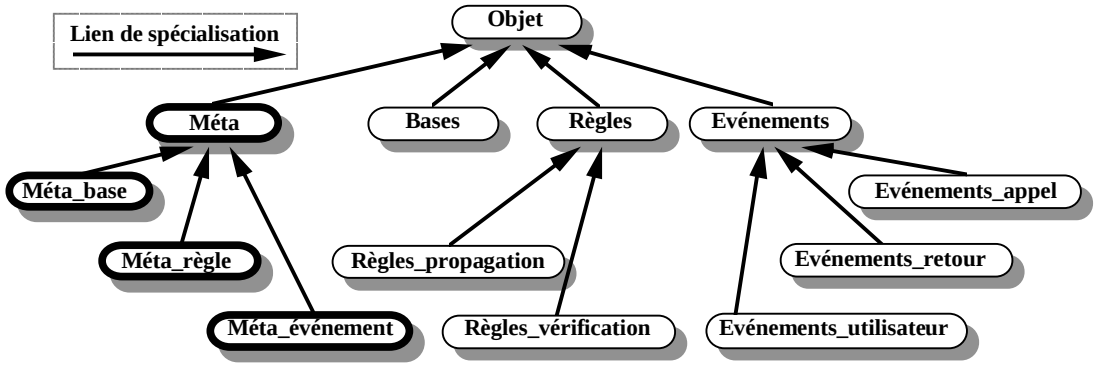


Figure 7-11 : Le noyau du Système Actif

Le Schéma Applicatif est alors complètement indépendant du Schéma Actif, la réciproque étant fausse. Par exemple, quand une méthode change de nom, les classes des événements qui lui sont associées doivent aussi changer de nom. Cette dépendance fait des événements des objets actifs sensibles aux opérations réalisées sur les objets du Schéma Applicatif. Ce comportement est réalisé par des règles ECA. En effet, l'évolution des règles, des événements et des bases est assurée par un ensemble de règles. De ce fait, le **système Actif devient évolutif**, car il peut s'auto-modifier.

Soit un exemple de règle permettant le changement de nom d'une méthode :

R : E : Retour Modifier_nom_classe (Classe Méta-méthode).
C : événements = extraire_événements_méthode (Classe).
A : MAPC ('modifier_nom_classe événements Classe).

Le schéma actif est sensible aux modifications des méthodes. La modification des connaissances modélisées par une méthode implique la modification des connaissances associées aux événements. Pour cette raison, nous introduisons un ensemble de règles pour gérer les vérifications et les propagations afin de maintenir la consistance du schéma actif. De même nous avons défini un ensemble de règles pour gérer l'évolution des règles et des bases de règles.

Gestion des événements

Un événement possède le même nom que la méthode correspondante. Si cette dernière change de nom, le nom de l'événement associé doit lui aussi changer. Si une méthode est créée et qu'elle possède des sous-méthodes dont les classes-événements existent (Figure 7-12), les événements appel et retour correspondants à la méthode créée doivent aussi être créés. La suppression d'une méthode entraîne celle des événements correspondants. Les deux graphes d'événements sont parallèles au graphe des méthodes. Aussi, si un lien de spécialisation est supprimé ou créé dans le graphe des méthodes, il doit aussi l'être dans les graphes des événements.

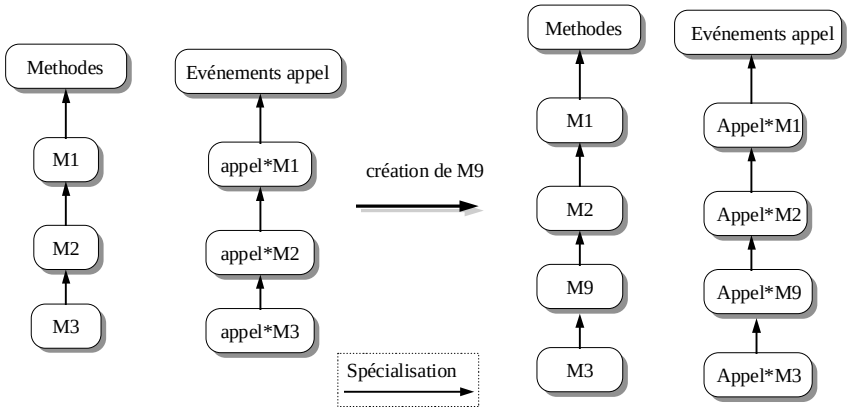


Figure 7-12 : Conséquence de la création d'une méthode

Le fait de créer la méthode M9 qui est super-méthode de M3 implique la création de la classe-événement "appel*M9" car la classe-événement "appel*M3" existe et que cette dernière doit hériter de la source de l'événement appel M9. Si la classe-événement "appel*M3" n'existait pas, la classe-événement "appel*M9" n'aurait pas été créée car elle ne possède pas de Réfénce Evénementielle et que sa source n'est pas héritée.

Comme les paramètres d'un événement sont définis à partir des arguments de la méthode, si un argument d'une méthode est modifié, la modification doit être répercutée sur les paramètres des événements correspondants. Les modifications prises en compte pour un argument sont le changement de nom, le changement de domaine, le changement de famille (entrée, sortie) et au niveau de la méthode même : la suppression ou la création d'un nouvel argument.

D'autres règles ECA

La Figure 7-13 présente le graphe de bases qui gère l'évolution du schéma actif. Lors de la suppression d'une base, la règle de vérification "confirmer_supp_base" demande confirmation

à l'utilisateur ; si celui-ci accepte, l'opération n'est pas interrompue et la règle de propagation "supprimer_regles_base" supprime alors les règles de la base.

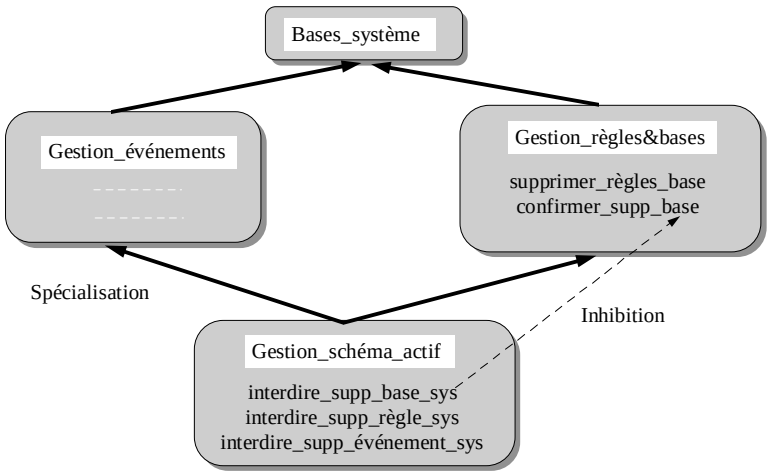


Figure 7-13 : Les bases gérant l'évolution du schéma actif

Par l'intermédiaire des règles de vérification de la base "gestion_schema_actif_sys", nous interdisons à l'utilisateur la suppression des règles système, des bases système et des événements utilisateurs. Nous remarquons que le message d'erreur de la règle "interdire_supp_base_sys" est prioritaire sur celui de la règle "confirmer_supp_base", la contrainte étant plus stricte.

7.4 LE MOTEUR D'EXECUTION

Le moteur d'exécution des règles est activé lors du déclenchement d'un événement. Son objectif est de sélectionner l'ensemble des règles qui référencent cet événement et de les exécuter. L'exécution d'une règle passe d'abord par l'évaluation de sa condition, puis par l'exécution de son action qui peut à son tour déclencher d'autres événements. Le moteur est donc récursif. Les règles qui sont sélectionnées à la suite d'un événement déclenché dans l'action d'une règle sont appelées des règles induites.

Cette section est découpée en trois phases : la phase de déclenchement des événements, la sélection des règles par filtrage et la phase d'exécution des règles. Dans cette dernière phase, nous traitons les différentes stratégies d'évaluation des règles de propagation et des règles induites.

7.4.1 Déclenchement des événements

Le déclenchement d'événements constitue une "pré-phase" du moteur d'exécution. Lorsqu'il s'agit d'événements méthodes ayant pour origine l'exécution d'une méthode ou plus généralement l'appel à une fonction générique, plusieurs événements peuvent être déclenchés.

En effet, l'appel à une fonction générique peut avoir pour conséquence l'exécution simultanée de plusieurs méthodes (coopération de méthodes) (cf. Chapitre 3), d'où le déclenchement simultanée de plusieurs événements. Dans ce cas, un problème d'ordonnancement d'événements apparaît lorsque deux méthodes doivent coopérer. En effet, si deux méthodes doivent coopérer, l'ordre de leur exécution dépend de l'ordre des traitements composant ces méthodes. Cet ordre peut être aléatoire ou défini par l'utilisateur à travers des contrôles de séquentialisation [Millasseau92]. Pour éviter un choix aléatoire de l'ordre des événements déclenchés, nous avons opté pour la stratégie qui consiste à déclencher simultanément les événements correspondants, c'est-à-dire qu'ils sont traités dans un même cycle par le moteur d'exécution. En ce qui concerne les événements utilisateurs, il ne peut y avoir d'événements simultanés, puisque leur déclenchement est réalisé par codification à l'intérieur d'un traitement précis.

De plus, les événements peuvent être hiérarchisés, et le déclenchement d'un événement entraîne celui de ses super-événements ; ainsi le moteur d'exécution est activé pour un ou plusieurs événements.

Lors du déclenchement d'un ou plusieurs événements, on identifie les classes événements correspondant aux événements déclenchés. Cet ensemble de classes est passé en paramètre au moteur d'exécution. Il est fourni aussi au moteur les éventuelles valeurs des paramètres de la source d'un événement (source effective). A partir de ces classes contenant les sources formelles et la source effective, le moteur pourra filtrer l'ensemble des règles sélectionnées.

Reprenons l'exemple de coopération de méthodes sur l'affichage des fenêtres (Chapitre 3). L'appel de la fonction générique Affiche_fenêtre, ayant en argument une fenêtre de type *FenêtreFond* et *FenêtreLogo*, entraîne l'exécution de *AfficheFenêtre*, *AfficheFenêtreFond* et *AfficheFenêtreLogo*. Le mécanisme de déclenchement est activé dans un premier temps avec les paramètres suivants :

- Evénements: Appel*AfficheFenêtreLogo et Appel*AfficheFenêtreFond ;
- source formelle : Fenêtre;
- source effective : F1.

Le mécanisme de déclenchement active alors le moteur avec les paramètres suivants :

- Evénements: Appel*AfficheFenêtreLogo, Appel*AfficheFenêtreTitre, Appel*AfficheFenêtre ;
- source formelle : Fenêtre;
- source effective : F1.

Dans un deuxième temps, le mécanisme de déclenchement est réactivé pour les événements de type *Retour*.

7.4.2 Sélection des règles

A partir des classes événements, le moteur récupère les instances de celles-ci. Ces instances sont les références événementielles (RE) qui définissent le filtrage sur la source et qui servent de lien avec les règles. Pour chaque RE, le moteur compare la source effective de l'événement déclenché avec les valeurs d'attributs de la RE. Si le filtrage est satisfait, les règles spécifiées au niveau de l'attribut *règles* des RE sont sélectionnées.

Soit l'instance suivante de l'événement *Appel*AfficheFenêtre* :

I_1 : Fenêtre*type = Fenêtre
Fenêtre*valeurs = (F1, F2)
Règles = (R1)

correspondant à la RE suivante de la règle R1 : appel AfficheFenêtre (Fenêtre (F1, F2)).

Et soit l'instance (ou RE) suivante de l'événement *Appel*AfficheFenêtreFond* :

I_2 : Fenêtre*type = FenêtreFond
Fenêtre*valeurs = (F2)
Règles = (R2)

F1 étant de type FenêtreFond, le mécanisme de filtrage récupère les deux instances I_1 et I_2. Il teste l'instance I_1 qui possède un filtrage : "Fenêtre*type et Fenêtre*valeurs" ; il compare celui-ci avec le paramètre formel "Fenêtre" qui a pour paramètre effectif F1. F1 étant de type FenêtreFond (donc de Fenêtre) et figurant dans la liste des valeurs possibles, la règle R1 est sélectionnée. Ensuite le mécanisme teste l'instance I_2 : F1 est bien du bon type mais comme il ne figure pas dans la liste des valeurs possibles, la règle R2 n'est pas sélectionnée.

7.4.3 Exécution des règles

Après la sélection des règles, le moteur divise l'ensemble des règles en deux catégories : les règles de vérification et les règles de propagation.

7.4.3.1 Les règles de vérification

Le moteur traite en premier les règles de vérification (Figure 7-14). Il commence par déterminer les règles les plus spécifiques (règles qui ne sont pas inhibées par d'autres règles de cet ensemble) ; il évalue ensuite les conditions de ces règles. Si la condition d'une règle est satisfaite, les autres règles de vérification et de propagation sont abandonnées et le message d'erreur de la règle est retourné à l'opération déclenchante, signalant ainsi l'interruption de l'opération. Si aucune règle de vérification n'a eu sa condition satisfaite, l'ensemble des règles de propagation est traité.

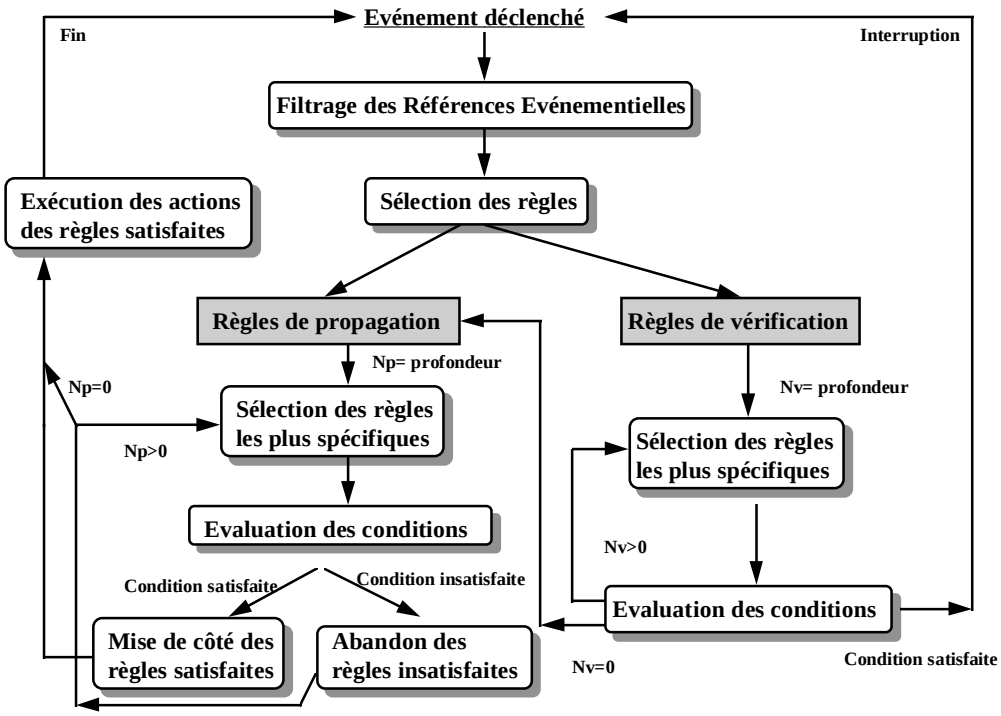


Figure 7-14 : Algorithme du moteur d'exécution

7.4.3.2 Les règles de propagation

L'exécution d'une règle de propagation se décompose en deux phases : l'évaluation de la condition et l'exécution de l'action. Suivant la stratégie choisie pour la séquentialisation de ces deux phases, les conséquences ne sont pas identiques. Deux stratégies sont alors possibles pour l'exécution des règles :

1. soit le moteur évalue toutes les conditions des règles sélectionnées, puis exécute leur action ;

2. soit il évalue pour chaque règle sélectionnée sa condition suivie de l'exécution de l'action.

La deuxième stratégie pose le problème du contexte d'évaluation de la condition. En effet, celui-ci n'est pas le même que celui du déclenchement de l'événement, car l'action peut modifier ce contexte. Pour cette raison, nous avons choisi la stratégie où on évalue d'abord l'ensemble des conditions, puis on exécute les actions des règles dont la condition est satisfaite.

Pour l'exécution d'un ensemble de règles de propagation, le moteur commence par déterminer les règles les plus spécifiques, et il évalue ensuite les conditions de ces règles. Si la condition d'une règle est satisfaite, la règle est mise de côté afin d'exécuter son action ultérieurement et les règles qu'elle inhibe sont retirées de l'ensemble des règles de propagation de départ. Sinon, la règle insatisfaite est abandonnée. L'algorithme boucle tant que l'ensemble des règles de

propagation est non vide. Ensuite, les actions des règles de propagation mises de côté sont exécutées. Les actions des règles sont exécutées les unes après les autres.

Lors de l'exécution de l'action d'une règle de propagation, celle-ci peut induire l'exécution de règles dites induites. Ce type de règles est sélectionné lorsqu'un événement est détecté dans l'action d'une règle. Trois stratégies sont possibles pour leur évaluation (Figure 7-15):

- Soit les règles induites sont exécutées après l'exécution des actions restantes (Stratégie1, Figure 7-15).
- Soit elles sont exécutées avant les autres actions (Stratégie2, Figure 7-15).
- Soit les conditions des règles induites sont évaluées avant les actions restantes et leurs actions le seront ultérieurement (Stratégie3, Figure 7-15).

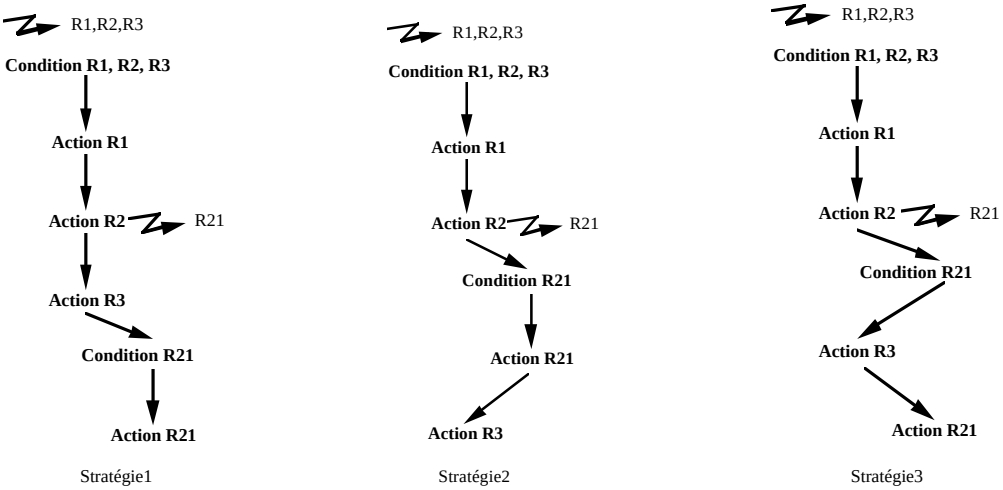


Figure 7-15 : Les stratégies d'exécution des règles induites

Nous retrouvons toujours le même problème du contexte de déclenchement de l'événement que précédemment. Dans la Figure 7-15, la première stratégie est rejetée car la condition de la règle R21 ne s'exécute pas dans le contexte de l'action de R2. Nous adoptons la deuxième stratégie car l'action s'exécute immédiatement (ce n'est pas le cas de la stratégie 3). En effet, nous pensons que si l'utilisateur déclenche une règle de propagation, c'est avec l'intention qu'elle soit évaluée dans le contexte de l'action. Par exemple, si la règle est déclenchée par l'appel d'une méthode, c'est que l'utilisateur souhaite qu'elle soit exécutée avant l'action de la règle déclenchante, sinon il aurait choisi l'événement retour méthode. Pour cela, les règles induites seront exécutées immédiatement, interrompant momentanément l'exécution des actions restantes.

Pour finir la description du modèle actif nous avons rajouté des commandes dynamiques pour contrôler le moteur d'exécution, telles que l'activation et la désactivation d'un événement ou d'une règle. Cela est souvent utile pour prévenir des boucles dans l'exécution des règles.

7.5 SYNTHESE

Nous avons proposé dans ce chapitre un modèle actif qui permet la génération et la mise en œuvre de règles actives. La syntaxe de ces règles est classique ; une règle s'énonce par trois composantes principales : l'événement, la condition et l'action. Ces règles ne sont pas encapsulées dans les classes, mais contenues dans des bases hiérarchisées. Leur exécution est assurée par un moteur offrant la possibilité de les activer et de les désactiver à tout moment, mais n'offrant malheureusement qu'un mode de couplage (immédiat). Les règles actives présentées dans ce chapitre permettent l'expression de l'évolution de schémas, mais l'inconvénient est que souvent le concepteur se voit obligé d'écrire plusieurs règles ECA pour exprimer une seule règle d'évolution. De plus, ce modèle de règles n'apporte de solutions pour la définition de plusieurs stratégies d'évolution. Le chapitre suivant va tenter d'apporter des solutions à ces inconvénients à travers un mécanisme de règles et de stratégies d'évolution.

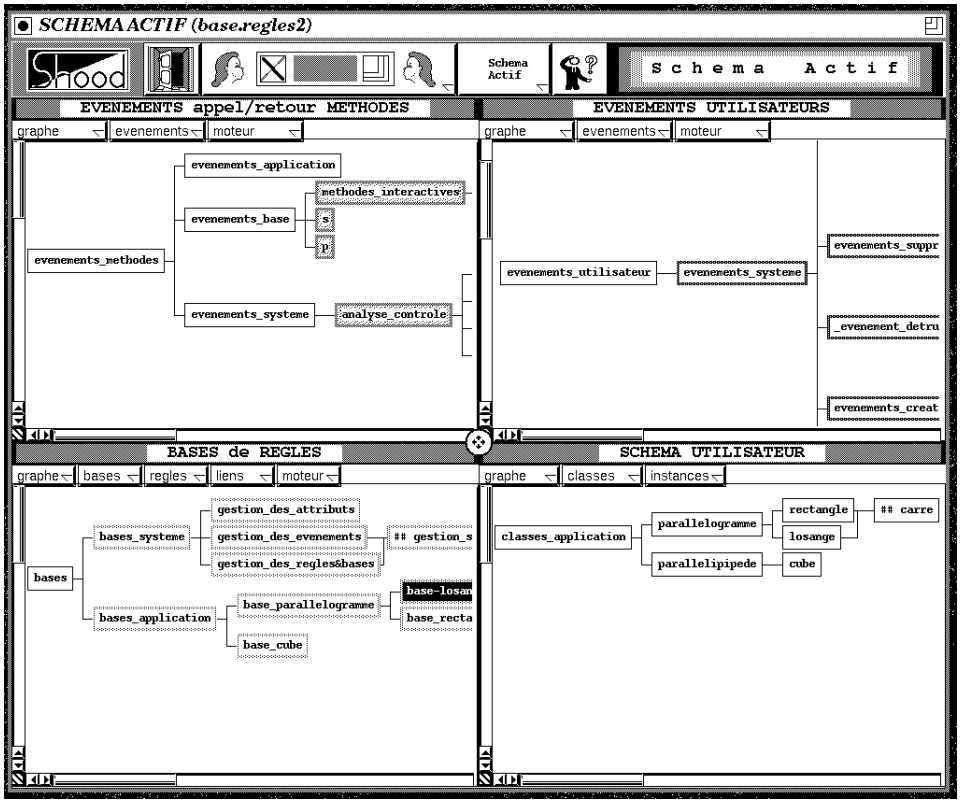
Parmi les extensions en cours du modèle actif, nous envisageons la modification du moteur d'exécution afin qu'il puisse accepter de nouveaux types de règles (Exemple : règles de cohérences, permettant de représenter des contraintes faibles), ainsi que l'introduction de nouvelles caractéristiques du modèle actif, comme la notion de priorité, de séquence d'événements et d'événement temporel [Clerc95].

Le modèle actif tel qu'il est présenté a été réalisé sur le prototype SHOOD (Version V1.14), sans modifier le code existant et cela grâce aux capacités d'extensibilité du modèle. Afin de mieux manipuler les concepts du système actif, celui-ci est exploitable à travers une interface homme-machine conviviale (Ecran 7-1 et Ecran 7-2).

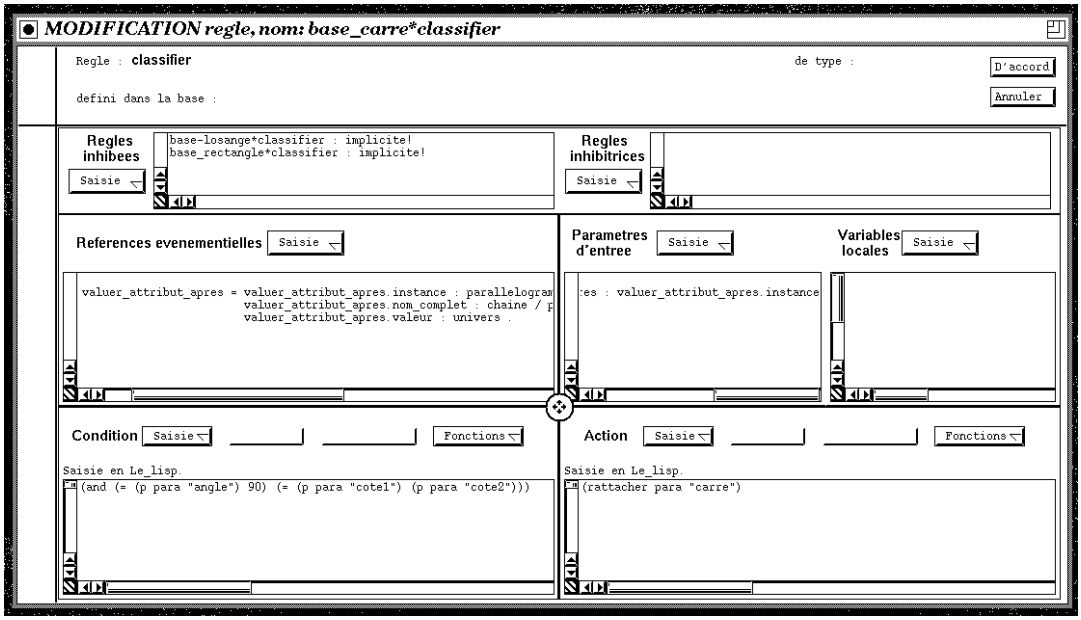
L'Ecran 7-1 présente la fenêtre principale du schéma actif. Cette fenêtre est composée de quatre sous fenêtres chacune gérant respectivement les événements méthodes, les événements utilisateurs, les bases de règles et le schéma utilisateur. Au niveau du graphe des événements, les événements "mouchetés" sont inactifs. Pour le graphe des événements méthode, un événement est à moitié moucheté, si seulement un des deux types d'événement (appel ou retour) est inactif.

L'Ecran 7-2 correspond à la fenêtre d'édition d'une règle active. Les quatre sous fenêtres permettent la saisie respectivement de l'expression événementielle, les paramètres et les variables, la condition et l'action. Une aide est fournie pour la saisie des informations telles que les règles inhibées, les filtrages, etc.

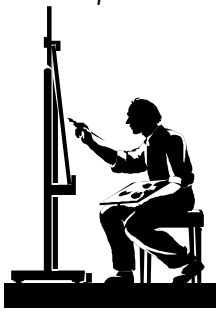
Les bases de l'Ecran 7-1 et la règle de l'Ecran 7-2 correspondent à l'exemple de classification des parallélogrammes (cf. Figure 7-7).



Ecran 7-1 : Gestion du modèle actif



Ecran 7-2 : Modification d'une règle



8. STRATEGIES ET REGLES D'EVOLUTION

Dans le chapitre 4, nous avons défini la stratégie d'évolution par défaut du système SHOOD, codé dans les opérations de manipulation. Dans ce chapitre, nous présentons un mécanisme qui permet la redéfinition de ces opérations, ainsi que la gestion de plusieurs stratégies. Ce mécanisme, le dernier de notre système d'évolution, a nécessité la définition des concepts d'opération, de règle et de stratégie, ainsi que la définition d'un langage de commandes permettant l'exploitation des règles et des stratégies d'évolution [Bounaas95d].

8.1 DEFINITIONS DES CONCEPTS

Soient C l'ensemble des classes et métaclasses de SHOOD, I l'ensemble des instances terminales représentant les données de la base de connaissances, sachant que toute instance de I est rattachée à une ou de plusieurs classes de C.

8.1.1 Définition d'une opération

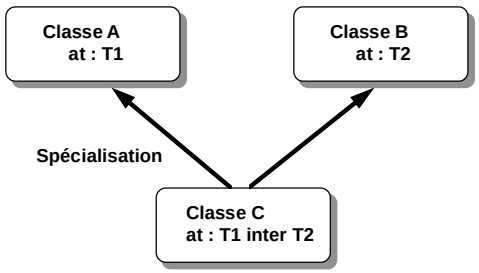
Une opération est une application sur l'ensemble C (de C vers C) représentant une évolution de classes, ou sur l'ensemble I représentant une évolution d'instances. L'ensemble des opérations est constitué de fonctions de création, de suppression et de modification de C et de I. Cet ensemble est noté O ; une opération o de O décrit la nature de l'évolution, c'est-à-dire une création, une suppression, etc. et sur quoi s'exécute o (attribut, classe, instance).

Pour qu'une exécution d'une opération de O n'introduise pas d'incohérence, il est nécessaire, dans certains cas, d'effectuer des contrôles ou vérifications pour rejeter ou accepter l'opération. Dans d'autres cas, le système doit réagir en conséquence à cette évolution par des propagations de mise à jour, toujours pour maintenir les informations cohérentes.

Ces vérifications et propagations devront avoir lieu respectivement avant et après l'exécution de l'opération qui sont illustrées par les exemples suivants :

- pour la suppression d'une classe, il faut effectuer des vérifications (par exemple : que la classe ne possède pas d'instances) et d'éventuelles propagations avant de supprimer la classe (par exemple : redéfinir le domaine des attributs ayant cette classe pour domaine),

- pour l'ajout d'un attribut dans une classe, les vérifications et les propagations peuvent avoir lieu après l'héritage de l'attribut ; ceci évite de simuler l'héritage pour détecter des incompatibilités de types.



Dans cet exemple, l'ajout de l'attribut *at* dans la classe B peut introduire une incompatibilité si l'intersection de *T2* et de *T1* est vide. Si le calcul *T1* inter *T2* est vide, l'ajout de l'attribut *at* sera défait.

Nous constatons par ces exemples qu'une opération doit être précédée et suivie d'une condition appelé vérification. De même, elle doit être précédée et suivie d'une action repésentant la propagation à effectuer. Nous proposons de fusionner syntaxiquement ces deux traitements, et comme les vérifications doivent être exécutées avant les propagations, une opération sera alors précédée d'un traitement appelé pré-opération et suivie d'un traitement appelé post-opération, chacun assurant la vérification puis la propagation (Figure 8-1).

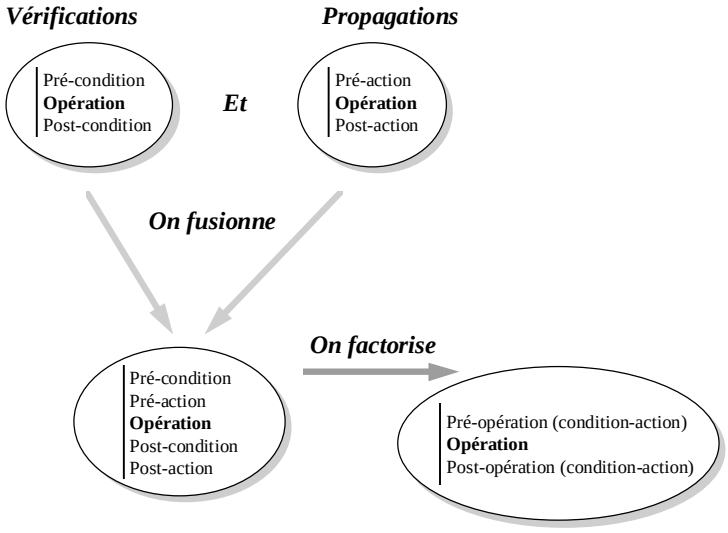


Figure 8-1 :Exécution d'une opération

8.1.2 Définition d'une pré-opération

Une pré-opération est un traitement effectué avant l'exécution de l'opération. Elle est constituée d'un couple *condition-action*, où *condition* permet de vérifier la possibilité d'exécution de l'opération et *action* permet de réaliser des propagations avant l'exécution de

l'opération. Si la condition d'une pré-opération est insatisfaite, l'opération est interrompue. Notons **Pré** l'ensemble des pré-opérations.

Exemple :

> Pré-op : Condition = Teste si une classe possède des instances
> Action = Tout attribut ayant pour domaine cette classe
> aura pour domaine une super-classe de la classe supprimée
> Op : Supprimer une classe

8.1.3 Définition d'une post-opération

Une post-opération est un traitement effectué après l'exécution de l'opération. De même que pour la pré-opération, une post-opération est constituée d'un couple *condition-action*. Si *condition* est insatisfaite l'exécution de l'opération et de la pré-opération sont défaites. Dans le cas contraire, *action* permet de propager les conséquences dues à l'exécution de l'opération. Notons **Post** l'ensemble des post-opérations.

Exemple :

> Op :Ajouter un attribut
> Post-op : Condition = Tester si une intersection d'attribut n'est pas vide
> Action = Répercuter l'ajout sur les instances.

8.1.4 Définition d'une règle d'évolution

Chaque cas d'évolution est exprimé par une règle d'évolution. Les règles d'évolution de schémas et d'instances vont conduire le programmeur ou le concepteur à préciser les répercussions caractérisant une évolution et les conditions qui la contraignent.

Une règle d'évolution r est un triplet < pré-o, o, post-o > tel que : pré-o ∈ Pré, o ∈ O et post-o ∈ Post. La règle *r* décrit de manière déclarative une sémantique d'évolution d'un concept. En d'autres termes, elle exprime les réactions du système à l'exécution de l'opération *o*. L'exécution de *o* entraîne l'évaluation de la pré-opération *pré-o* : si sa condition est satisfaite, alors son action est exécutée, ainsi que l'opération *o*. La condition de *post-o* est évaluée après l'exécution de *o*, et si celle-ci n'est pas vérifiée, les exécutions de *o* et de *pré-o* sont défaites. Dans le cas contraire (aucune incohérence n'a été détectée), l'action de *post-o* est exécutée pour propager les conséquences de l'exécution de *o*. Notons **R** l'ensemble des règles d'évolution.

Exemple :

r : règle d'évolution traduisant une sémantique (partielle) de la suppression de classe
o : Supprimer_classe
Pré-o : Condition = Teste si une classe possède des instances
Action = Tout attribut ayant pour domaine cette classe
aura pour domaine une super-classe de la classe supprimée
Post-o : Condition = aucune
Action = Envoyer un message (" la suppression de la classe est terminée").

Si on demande la suppression de classe (appel de o), le système vérifiera d'abord la condition de la pré-opération. Si la classe ne possède pas d'instances, la classe pourra être supprimée. Mais avant la suppression, les domaines d'attribut référençant cette classe sont modifiés par une super-classe commune. Après la suppression, un message de fin de suppression (post-opération) est envoyé.

8.1.5 Définition d'une stratégie

Ce concept a été introduit pour permettre la gestion de plusieurs stratégies d'évolution. Une stratégie représente l'ensemble des sémantiques d'évolution d'une application ou des concepts d'un modèle de données. Elle est modélisée par l'ensemble des règles d'évolution traduisant cette stratégie d'évolution. Nous offrons, par ce concept, la possibilité de faire coexister plusieurs schémas d'évolution dont un seul sera pris en compte par le système à un instant donné. Nous offrons de même la possibilité de réutiliser un schéma d'évolution pour en créer d'autres.

Nous définissons une stratégie comme un ensemble de règles d'évolution de R. Notons S l'ensemble des stratégies. Toute stratégie $s_i \in S$ est composée d'un sous-ensemble R_i de R.

Soient deux règles d'évolution (r1, r2) telles que :

r1 = <o1, pré1, post1> définit une sémantique de l'opération o1 avec $r1 \in R1$,

et

r2 = <o1, pré2, post2> définit une autre sémantique de l'opération o1 avec $r2 \in R$.

r1 et r2 définissent deux règles d'évolution "disjointes" (elles ne peuvent pas appartenir à une même stratégie) ; lors de l'exécution de o1, on doit évaluer soit r1, soit r2. Nous proposons alors de définir une deuxième stratégie qui redéfinit r1 par r2. La définition d'une stratégie devient alors :

soient s1 et $s2 \in S$, avec s1 et s2 composées respectivement de R1 et R2, et s2 redéfinit s1 :

$r1 \in R1$ et $\exists\ r2 \in R2$/ r2 redéfinit r1 ==> $R2 = R1-\{r1\} \cup \{r2\}$.

Le programmeur choisit toujours une et une seule stratégie de S et lors de l'exécution d'une opération, la règle correspondant à la stratégie choisie sera évaluée.

Exemple :

soient r1 et r2 deux règles d'évolution :

> r1 : o= supprimer une classe
> pré-o = vérifier que la classe ne possède pas d'instances

> r2 : o= supprimer une classe
> pré-o = supprimer les instances de cette classe

Comme ces deux règles sont disjointes, elles pourront appartenir à deux stratégies différentes. En phase d'exécution, seulement une des deux règles doit être évaluée.

Ne possédant pas de moyen de vérifier la disjonction de deux règles, il est impossible de vérifier la cohérence d'une stratégie. En effet, en présence de deux règles d'évolution pour une même opération, il n'est pas possible de dire si deux règles sont contradictoires ou complémentaires. De même, la complétude d'une stratégie ne peut être vérifiée car le nombre de règles nécessaires pour exprimer la sémantique complète d'une opération d'évolution n'est pas connu. Actuellement, la vérification de la consistence des règles d'évolution est à la charge de celui qui les spécifient.

Par la suite, nous présentons l'utilisation de ces concepts à travers un langage de commandes, permettant de décrire l'évolution d'une classe ou d'une instance.

8.2 REGLES D'EVOLUTION

Comme présenté dans le paragraphe précédent, une règle d'évolution est définie par trois composants : l'opération, la pré-opération et la post-opération. Avant de donner la syntaxe permettant de créer une règle d'évolution, nous décrivons celle de ses composants.

- **opération** : une opération est décrite par son nom, ainsi que ses paramètres. Ces paramètres permettent l'expression d'un filtrage sur les paramètres formels de **o**. Ce filtrage peut être réalisé sur le type et/ou sur les valeurs appelés respectivement filtrage_type et filtrage-valeur. L'ensemble des opérations est constitué des opérations du support d'évolution (cf. Chapitre 4).

Exemple :

> Opération : Supprimer_classe (oid : Méta_a_clef)

Cette opération concerne la suppression d'une classe qui peut posséder des clefs. Un filtrage sur le type de l'argument de la méthode Supprimer_classe est spécifié.

- **pré-opération** : un couple Condition-Action (C, A) ; si l'évaluation de C (expression booléenne) est fausse, l'opération **o** est interrompue, sinon A est exécutée. A est une séquence

d'opérations, qui peuvent être des opérations du support d'évolution ou des opérations sur la base (requêtes, appels de méthodes ...).

Exemple :

```
Opération : Supprimer_classe (oid : Méta_a_clef)
Pré-opération :     ( Condition : (not (p oid "instances")))
                      Action : (Message "Suppression de la classe autorisée")
```

Si la classe à supprimer possède des instances, la suppression sera rejetée. La condition de la pré-opération vérifie que l'opposé de la projection de l'attribut instances sur la classe est vrai.

- **post-opération** : un couple Condition-Action (C, A) ; si l'évaluation de C est fausse, l'exécution de **o** est défaite, sinon A est exécutée pour propager les mises à jour dues à l'exécution de **o.**

Exemple :

```
Opération : Valuer_attribut (oid: Losange ; nom-att : Chaîne, ("côté1", "côté2"))
Post-opération :    (Condition : ( = (P oid “côté1”)  (P oid “côté2”) )
                    Action : (valuer_attribut oid "surface"
                                                (* (P oid “côté1”)  (P oid “côté2”)))
```

("côté1", "côté2") constitue un filtrage_valeur sur l'attribut *nom-att* de type Chaîne. Lorsqu'un des deux côtés d'un losange est modifié, la condition vérifie leur égalité. Dans le cas où cette dernière est satisfaite, la modification est propagée en recalculant la surface du losange. Dans l'autre cas (condition fausse), la modification est défaite.

Pour permettre un passage de valeurs entre les différentes conditions et actions des pré et post opérations, nous définissons au niveau d'une règle d'évolution une liste de noms de variables que le programmeur pourra affecter et consulter. Dans l'exemple ci-dessus, on peut utiliser des variables, pour optimiser l'accès aux valeurs de l'instance.

Exemple :

```
Opération : Valuer_attribut (oid: Losange ; nom-att : Chaîne, ("côté1", "côté2"))
variables : côté1, côté2
Post-opération :    ( Condition : (PROGN  (SETQ  côté1  (P oid “côté1”))
                                          (SETQ  côté2  (P oid “côté2”))
                                          ( =  côté1 côté2) )
                    ( Action : (valuer_attribut oid "surface" (* côté1  côté2) ) )
```

La syntaxe de la création d'une règle d'évolution est la suivante, celle de la modification est identique :

Syntaxe :

*(**creer_regle_evolution** nom_règle variables opération pré-opération post-opération)*

où :

variables = (var_1,..., var_N)

opération = nom_opération ($para_1$: $type_1$, (val_1,..., val_N) ; $para_N$: $type_N$, (val_1,..., val_N)).

*pré-opération = post-opération = (**C** : expression_booléenne, **A** : Expression).*

nom_règle représente une chaîne de caractères permettant d'identifier la règle,

variables est une liste de noms de variables utilisées dans le code des conditions et des actions,

nom_opération correspond au nom d'une méthode d'évolution (cf. Chapitre 4),

$para_I$ $type_I$ et (val_I) désignent respectivement les noms des arguments des opérations, un filtrage sur le type de $para_I$ et un filtrage sur les valeurs de $para_I$. $type_I$ et (val_I) sont optionnels,

Expression_booléenne et Expression représentent des expressions Lisp. *Expression_booléenne* doit rendre une valeur booléenne (NIL pour désigner faux). Les conditions et les actions sont optionnelles.

La syntaxe présentée ci-dessus est identique pour une règle d'évolution de classes et d'instances ; le type des objets concernés (métaclasse, classe, instance) fait la différence.

Exemples

Soit la règle d'évolution de classes concernant l'ajout d'un attribut :

Exemple :

```
nom_règle = règle_ajout_attribut1
opération = ajouter_attribut (classe : Meta ; def_att : Objet_lisp ; res: Bool)
Pré-opération = ()
Post-opération = ( C : res
                   A : (MAPC (LAMBDA (oid)
                             (valuer_attribut oid (CAR def_att) NIL)
                             )
                             (ex_toutes_instances classe)
                             ))
          /* MAPC est une fonction Lisp permettant d'itérer l'appel d'une fonction sur
          une liste d'arguments */
```

def_att est un schéma décrivant la définition d'un attribut. Par exemple, pour ajouter un attribut *âge* de type entier la définition serait : ("age" (type (un "entier"))). Et *res* correspond à un argument en sortie de la méthode *ajouter_attribut* qui contiendrait le résultat de l'exécution. L'opération *ajouter_attribut* ajoute l'attribut dans la classe et le fait hériter dans les sous-classes. Si la méthode

rend *vrai* comme résultat, la condition de la post-opération est satisfaite. On value, par exemple, les attributs des instances de classe à NIL, sinon l'ajout de l'attribut est défait (échec de l'exécution).

Soit la règle suivante traduisant un cas de suppression de classe :

Exemple :

```
nom_règle = règle_suppression_classe1
opération = supprimer_classe (oid : Méta)
Pré-opération = (C : (not (utilise_par oid meta_attribut "domaine"))
                 A : (pre_declarer (ex_super_comm oid)  (ex_nom_attributs_propres
                     oid)))
Post-opération = ( C : (), A : (Message " la suppression de classe est terminée")).
```

Lors de la suppression d'une classe, si cette classe est utilisée par un domaine d'attribut, la suppression est interrompue, sinon nous pré-déclarons les attributs dans une super-classe commune et ceci avant la suppression de la classe.

L'exemple ci-dessus présente la stratégie par défaut pour la suppression d'une classe. Le programmeur peut mettre en œuvre une deuxième stratégie, par exemple celle d'Orion, en écrivant la règle ci-dessous. Cette règle consiste à supprimer les attributs propres et à modifier, par la première super-classe de la classe à supprimer, le domaine de l'attribut qui a pour domaine la classe à supprimer.

Exemple :

```
nom_règle = règle_suppression_classe2
opération = supprimer_classe (classe : Méta)
Pré-opération =      (C : (),
                      A : (supprimer_att_propres classe)
                          (modifier_dom_atts classe)
Post-opération = ( C : (), A : (Message " la suppression de classe est terminée").
```

Lors de la suppression d'une classe qui est utilisée par un domaine d'attribut, on modifie le domaine de l'attribut par la première super-classe de la classe à supprimer (action réalisée dans la fonction modifier_dom_atts, qu'on ne détaillera pas ici). Les attributs propres sont supprimés (action effectuée dans la fonction supprimer_att_propres).

Mais parmi ces deux stratégies, une seule doit être évaluée lors de la suppression d'une classe.

8.3 STRATEGIES D'EVOLUTION

Une des propriétés du système d'évolution est la possibilité d'exprimer des stratégies multiples d'évolution. Une stratégie est définie comme un ensemble de règles d'évolution dans lequel on peut redéfinir ou ajouter des règles.

La syntaxe pour définir une stratégie est la suivante :

Syntaxe : ***(definir-strategie*** *nom-strategie nom_super_strategie)*

Lors de la définition d'une stratégie, il faut spécifier son nom (*nom_strategie*) et celui de la stratégie qu'elle redéfinit (*nom_super_strategie*). Au niveau de cette syntaxe on ne précise pas les règles redéfinies ou ajoutées. Cette information sera spécifiée au niveau de la création de la règle.

Le support d'évolution définit une stratégie d'évolution par défaut décrite à travers les opérations de manipulation (cf. Chapitre 4). Pour cette raison, nous définissons une stratégie virtuelle S0 représentant la stratégie par défaut.

A partir de cette redéfinition de stratégie, on obtient une hiérarchie de stratégies, où chaque stratégie peut redéfinir ou ajouter des règles d'évolution.

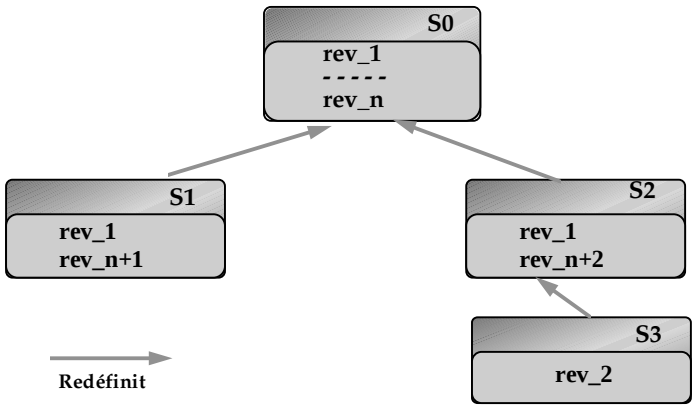


Figure 8-2 : Hiérarchie de stratégies

Dans la stratégie S0 (Figure 8-2) sont définies les règles d'évolution rev_i (i=1..n) ; S1 et S2 redéfinissent la règle rev_1 ; S3 redéfinit rev_2 ; S1 et S2 ajoutent respectivement les règles rev_n+1 et rev_n+2.

Le déclenchement d'une règle **r** est réalisé lors de l'invocation de l'opération correspondante. Le système évalue la règle **r** de la stratégie qui est active (**s**). Si la règle **r** n'est pas présente dans la stratégie **s**, le système effectue une recherche dans les stratégies que **s** redéfinit. Dans l'exemple de la Figure 8-2, lors du déclenchement de la règle rev_1, si S3 est la stratégie active, le système évalue la règle rev_1 de la stratégie S2.

Pour cela, on offre à l'utilisateur la possibilité d'activer et de désactiver dynamiquement une stratégie par les commandes suivantes :

Syntaxe : *activer (nom-stratégie)*
désactiver (nom-stratégie)

Afin d'éviter les problèmes de résolution de conflits de nom de règles, une stratégie ne peut redéfinir qu'une seule stratégie (redéfinition simple) et une seule stratégie peut être active à la fois.

Pour prendre en compte les stratégies, la syntaxe de la création d'une règle d'évolution est redéfinie :

Syntaxe : *(**creer_regle_evolution** nom-strategie nom-regle operation pre-operation post-operation)*
où *nom-strategie* est le nom de la stratégie où sera définie ou redéfinie *nom-regle*.

8.4 EXEMPLES

8.4.1 Exemple : suppression d'une classe

Nous citons ci-après trois stratégies possibles pour la suppression d'une classe :

- Les stratégies possibles par rapport aux attributs propres de la classe :
    1. les attributs sont pré-déclarés dans une super-classe commune et propagés dans les sous-classes s'ils ne sont pas affinés,
    2. la suppression de la classe est interdite,
    3. les attributs sont supprimées.
- Les stratégies possibles par rapport aux instances :
    1. les instances sont retirées (rattachées aux super-classes),
    2. la suppression de la classe est interdite,
    3. les instances sont supprimées.

La stratégie ***1*** pour les instances et les attributs est la stratégie par défaut (S0), réalisée dans l'opération de suppression de classe. Pour mettre en œuvre les autres stratégies, nous définissons deux stratégies (S1, S2) où nous créons deux règles correspondant à la gestion respective des attributs et des instances :

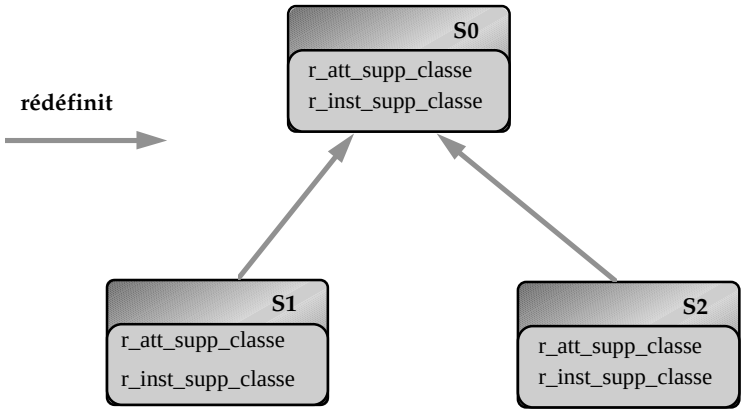


Figure 8-3 : Stratégies de l'exemple

Les règles d'évolution pour la gestion des attributs :

```
nom_règle = r_att_suppression_classe
nom-stratégie = S1
opération = supprimer_classe (oid : Meta)
Pré-opération = (C : /*la classe ne possède pas d'attributs propres */
                    (ex_att_propres oid),
                    A : ())
```

```
nom_règle = r_att_suppression_classe
nom-stratégie = S2
opération = supprimer_classe (classe : Méta)
Pré-opération = (C : (),
                    A : /* Supprimer les attributs propres */
                        (MAPC 'supprimer_attribut (ex_att_propres oid)))
```

Et pour la gestion des instances, nous définissons deux règles respectivement dans les stratégies S1 et S2 :

```
nom_règle = r_inst_suppression_classe
nom-stratégie = S1
opération = supprimer_classe (oid : Méta)
Pré-opération = (C : /* la classe ne possède pas d'instances */),
                    (P oid "instances")
                    A : ())
```

```
nom_règle = r_inst_suppression_classe
nom-stratégie = S2
opération = supprimer_classe (oid : Méta)
Pré-opération = (C : (),
                  A : (MAPC 'supprimer_instance (ex_instances oid)))
```

Suivant la stratégie désirée, l'utilisateur activera la stratégie correspondante (S1 ou S2). Si l'utilisateur active la stratégie S1, la suppression d'une classe sera refusée si cette classe

possède des instances ou des attributs propres. L'utilisateur ne pourra pas sélectionner la première stratégie pour la gestion des instances et la deuxième pour la gestion des attributs. Pour pallier ce problème, il devra définir autant de stratégies que de combinaisons désirées.

8.4.2 Exemple : représentation de contraintes

Dans cet exemple, on désire représenter les contraintes de conception d'une clé plate. En définissant les informations nécessaires pour notre exemple, la clé est représentée par au moins quatre attributs (Dc1, Dc2, Lb1 et Lb2).

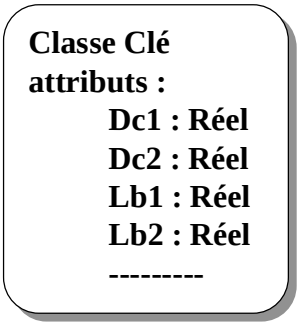


Figure 8-4 : Représentation minimum de la clé

- Dc1 et Dc2 représentent les diamètres des cylindres extérieurs situés aux extrémités de la clé.
- Lb1 et Lb2 représentent les largeurs des boîtes qui sont supprimées des cylindres (Figure 8-5).

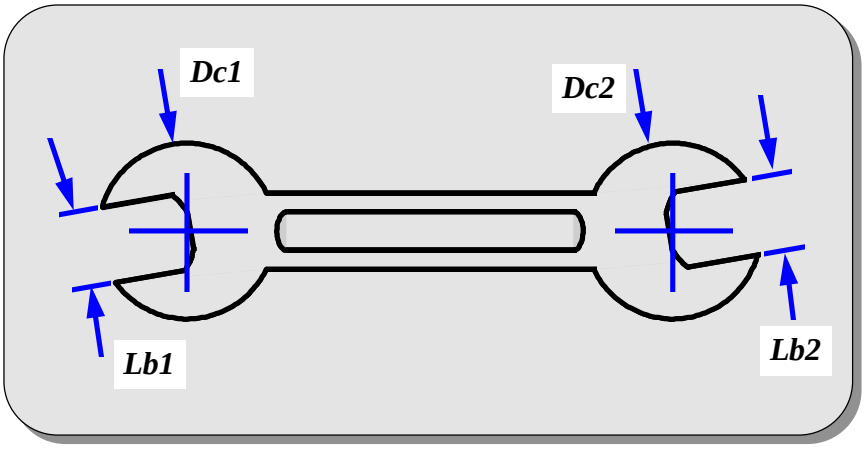


Figure 8-5 : Exemple de la clé

Dans cet exemple, nous voulons représenter deux contraintes simples:

- **Contrainte 1** : La largeur de la première boîte doit être supérieure de 1 à celle de la deuxième boîte : ***Lb1=Lb2+1***
- **Contrainte 2** : La largeur d'une boîte (b1 ou b2) doit être plus petite que le diamètre du cylindre correspondant (c1 ou c2) : ***Lbn < Dcn / 1,5. (n=1, 2)***

Soient deux façons (ou stratégies) d'exprimer ces deux contraintes :

- Contrainte 1 : Si Lb1 (ou Lb2) est modifié alors
    1. vérifier que Lb2 est égal à Lb1+1
    2. modifier la valeur de Lb2 par Lb1+1 (ou Lb1 par Lb2-1 si c'est Lb2 qui est modifié)
- Contrainte 2 : Si Lbn (ou Dcn) est modifié alors
    1. vérifier que Dcn est égal à 1,5xLbn
    2. modifier la valeur de Dcn par 1,5xLbn (ou Lbn par Dcn/1,5 si c'est Dcn qui est modifié)

La première stratégie tente uniquement de vérifier que la valeur modifiée est cohérente, tandis que la deuxième stratégie va propager la modification pour faire respecter la contrainte. Pour cet exemple, il faudra définir deux stratégies : S1 qui représente les contraintes de vérification et S2 qui représente les contraintes de propagation. L'utilisateur choisira S1 ou S2 suivant les besoins de son application.

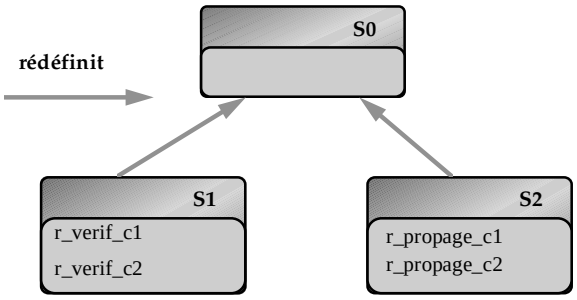


Figure 8-6 : Stratégies pour les contraintes

Soient r_verif_c1 et r_propage_c1, les règles concernant la contrainte 1 :

```
nom_règle = r_verif_c1
nom-stratégie = S1
opération = valuer_attribut (oid : Clef; att : Chaine, ("Lb1", "Lb2"))
Pré-opération = (C : /* On vérifie si la contrainte1 est respectée*/),
                (= (+ (P oid "Lb1") 1) , (P oid "Lb2"))
                A : ())
```

```
nom_règle = r_propage_c1
nom-stratégie = S2
opération = valuer_attribut (oid : Clef; att : Chaine, ("Lb1", "Lb2"))
Post-opération = (C : ()),
                 A : /* On vérifie que contrainte1 n'est pas respectée*/
                     (IF (<> (+ (P oid "Lb1") 1) , (P oid "Lb2"))
                         (IF (= att "Lb1") (valuer_attribut oid "Lb2" (+ Lb1 1))
                                           (valuer_attribut oid "Lb1" (- Lb2 1))
                         ))))
```

De même, nous retrouvons deux autres règles, que nous ne présenterons pas, (r_verif_c2 et r_propage_c2) pour la contrainte 2. L'utilisateur peut activer S1 ou S2 selon qu'il désire s'assurer que les valeurs de la clé sont correctes ou qu'il désire automatiser sa conception.

8.5 REALISATION

Nous avons vu dans les exemples du chapitre 7 que les règles ECA du modèle actif permettent d'exprimer l'évolution d'un concept, mais souvent il faut écrire plusieurs règles ECA pour une seule règle d'évolution. Pour cela nous avons proposé un langage d'expression de règles d'évolution [Bounaas95d].

Pour rendre ce langage exécutable, il faut d'une part permettre le déclenchement et l'exécution des règles d'évolution et d'autre part gérer les stratégies d'évolution. Pour la mise en œuvre des règles d'évolution, nous établissons une correspondance entre une règle d'évolution et des règles ECA. La création d'une règle d'évolution génère les règles ECA correspondantes. Un lien est établi entre une règle d'évolution et ses règles ECA, qui permettra la modification de ces dernières. Quant à l'exécution, celle-ci est assurée par celles des règles ECA. Pour les stratégies d'évolution, nous utilisons le concept de base de règles. A une stratégie, nous associons une base de règles ; la redéfinition d'une stratégie est représentée par la spécialisation de la base de règles associée. Dans les paragraphes qui suivent, nous détaillons la mise en œuvre des stratégies et des règles d'évolution.

8.5.1 Règles d'évolution

La correspondance entre une règle d'évolution et plusieurs règle ECA se fait facilement. En effet, une opération correspond à une référence événementielle où l'événement est celui déclenché à l'appel (pour représenter les pré-opérations) ou au retour (pour représenter les post-opérations) d'une méthode d'évolution (qui représente l'opération). A chaque pré-opération et post-opération, on fait correspondre une règle de vérification et une règle de propagation mettant en œuvre respectivement la condition et l'action. Donc, à toute règle sont associées au plus deux règles de vérification et deux règles de propagation.

Traduction d'une règle d'évolution en règles actives

Soit REV = (nom_rev, op_rev, var_rev, pre_op_rev, post_op_rev), où :

nom_rev : le nom de la règle d'évolution
op_rev = (nom_op, para_op) : le nom de l'opération et les paramètres associés
var_rev : ensemble des variables de REV
pre_op = (condition1, action1) : la pré-opération
post_op = (condition2, action2) : la post-opération

Soient, REAV1 et REAV2 deux règles actives de vérification, et REAP1 et REAP2 deux règles actives de propagation.

Avec : REAxy $_{(x = V \text{ ou } P,\ y = 1 \text{ ou } 2)}$ = (nom_rea, type_rea, re_rea, var_rea, cond_rea, act_rea)

- nom_rea : le nom de la règle active
- type_rea : le type de la règle (verfication ou propagation)
- re_rea : la référence événementielle
- var_rea : les variables locales
- cond_rea : la condition
- act_rea : l'action

Et :

REAV1 =

- nom_rea = nom_rev.pre.verification
- type_rea : verfication
- re_rea : Appel nom_op (para_op)
- var_rea : var_rev
- cond_rea : (NOT Condition1) /* Interruption si la Condition1 est fausse */
- act_rea : ("Incohérence dans la pré_opération de nom_rev")

REAV2 =

- nom_rea = nom_rev.post.verification
- type_rea : verfication
- re_rea : Retour nom_op (para_op)
- var_rea : var_rev
- cond_rea : (NOT Condition2)
- act_rea : ("Incohérence dans la post_opération de nom_rev")

REAP1 =

- nom_rea = nom_rev.pre.propagation
- type_rea : propagation
- re_rea : Appel nom_op (para_op)
- var_rea : var_rev
- cond_rea : Vrai
- act_rea : Action1

REAP2 =

- nom_rea = nom_rev.post.propagation
- type_rea : propagation
- re_rea : Retour nom_op (para_op)
- var_rea : var_rev
- cond_rea : Vrai
- act_rea : Action2

Les paramètres spécifiés au niveau de l'opération correspondent aux filtrages types et valeurs de l'événement (cf. Chapitre 7). Pour les règles de vérification, la chaîne de caractères spécifiée au niveau de l'action représente le message d'interruption.

La traduction d'une règle d'évolution est réalisée par une fonction Le_Lisp qui a pour paramètre d'entrée, la règle d'évolution à traduire et en sortie, elle génère les quatre règles ECA ci-dessous.

Les règles ECA générées pour une règle d'évolution seront définies dans la même base de règles. Nous verrons plus loin que toutes les règles ECA d'une même stratégie seront définies dans la même base (cf. 8.5.2).

Exemple

Si nous traduisons la règle d'évolution sur la suppression de classe :

```
nom_règle = règle_suppression_classe1
opération = supprimer_classe (oid : Méta)
Pré-opération = (C : (not (utilise_par oid meta_attribut "domaine"))
                 A : (pre_declarer (ex_super_comm oid)  (ex_nom_attributs_propres
                    oid)))
Post-opération = ( C : (), A : (Message " la suppression de classe est terminée")).
```

On obtient les trois règles ECA suivantes :

```
Nom_règle = règle_suppression_classe1.pré.vérification
Référence-événementielle = Appel supprimer_classe  (classe   Meta)
Type = Vérification
Condition = (not (not (utilise_par oid meta_attribut "domaine"))
Action = "Incohérence dans règle_suppression_classe1"
```

```
Nom_règle = règle_suppression_classe1.pré.propagation
Référence-événementielle = Appel supprimer_classe
Type = Propagation
Condition = Vrai
Action = (pre_declarer (ex_super_comm oid)  (ex_nom_attributs_propres oid))
```

```
Nom_règle = règle_suppression_classe1.post.propagation
Référence-événementielle = Retour supprimer_classe  (classe   Meta)
Type = Propagation
Condition = Vrai
Action = (Message " la suppression de classe est terminée")
```

Pour cet exemple, il n'y a que trois règles ECA car la post-opération ne possède pas de condition.

Modélisation d'une règle d'évolution

Afin de garder les références d'une règle d'évolution vers ses règles ECA et de permettre le passage de valeurs des variables entre les règles ECA, nous modélisons une règle d'évolution

par une classe qui référencera ses règles ECA et qui aura pour attributs les variables. Cette classe sera sous-classe de la classe *Regle_evolution*, laquelle définit quatre attributs ayant pour domaine la classe *Regles*. De cette façon, toute instance d'une règle d'évolution pourra référencer les instances des règles ECA (voir Figure 8-7) . En effet, lors de la détection de l'événement (donc de l'opération d'évolution), une instance de la règle d'évolution est créée pour référencer les instances des règles ECA et contenir les valeurs des variables. Ces variables sont consultées et mise à jour par les règles ECA.

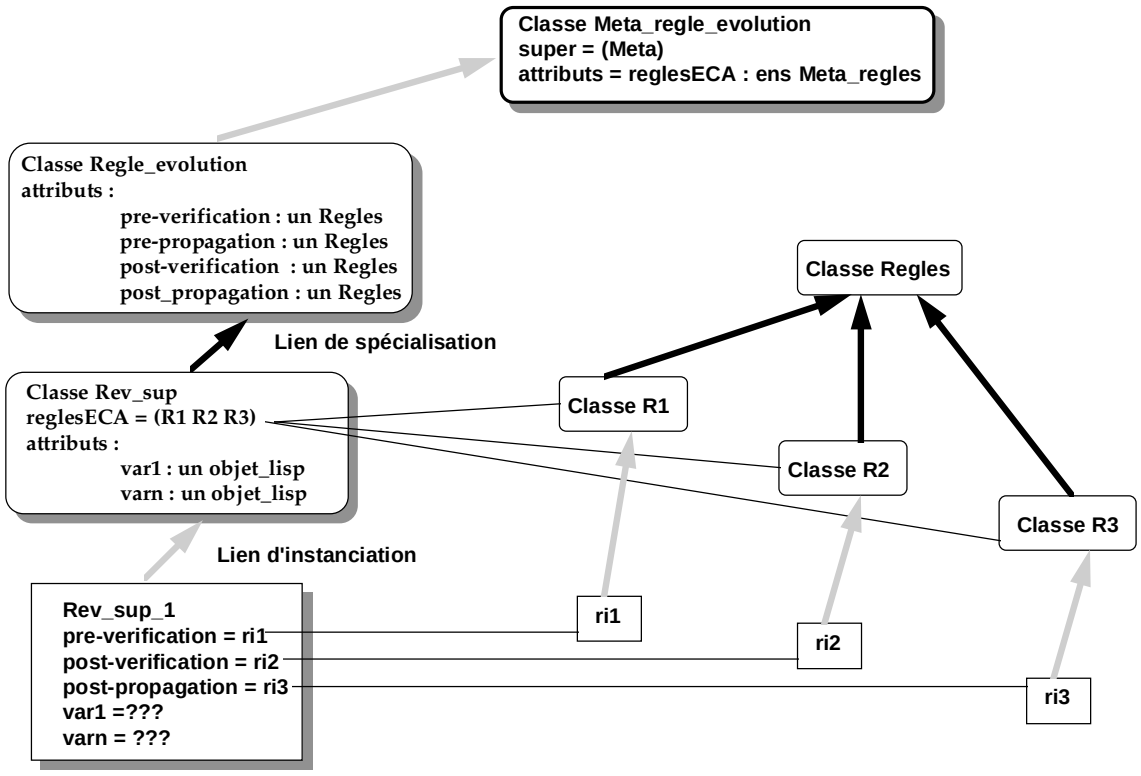
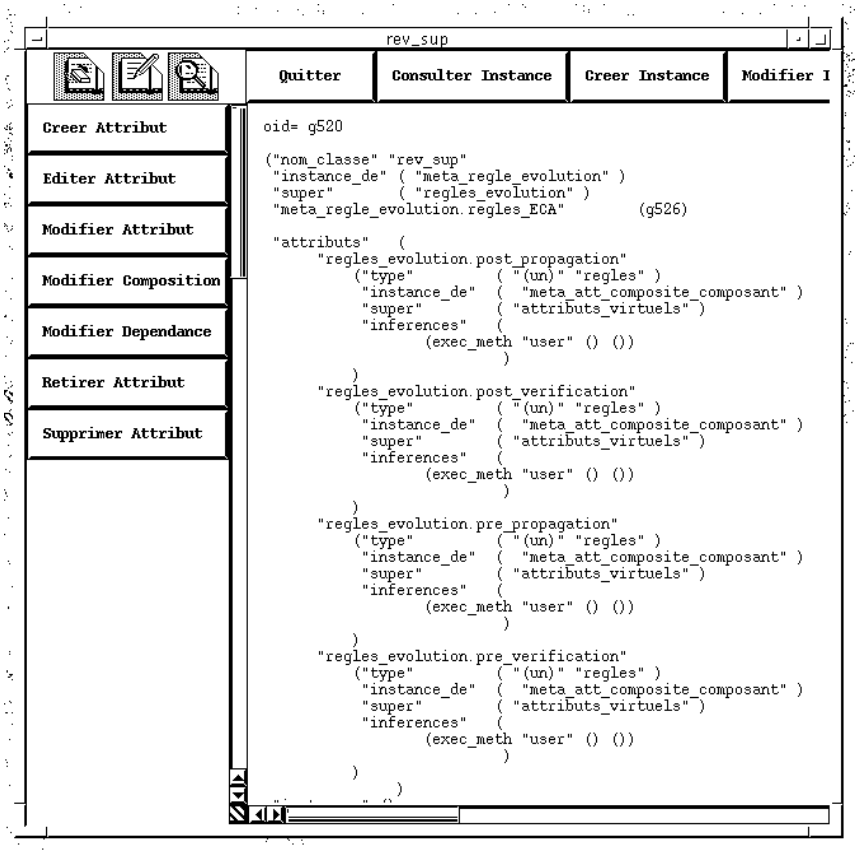


Figure 8-7 : Modélisation d'une règle d'évolution

Pour permettre l'accès aux bonnes variables par les règles ECA, on insère un code dans ces règles pour accéder à l'instance de la règle d'évolution correspondante.

Code à insérer dans la condition et l'action d'une règle ECA *rea* :

> Soit *rev* la règle d'évolution qui référence *rea*.
>
> Pour chaque variable *var* de *rea* :
>
> générer_variable (*var*)
>
> affecter_variable (*var*, (P *rev* "*var*")) /* P : fonction de projection */



Ecran 8-1 : La classe représentant une règle d'évolution

Exécution d'une règle d'évolution

Pour l'exécution d'une règle d'évolution, nous utilisons le moteur d'exécution des règles ECA moyennant une modification pour prendre en compte les variables d'une règle d'évolution. L'exécution de cette dernière se déroule ainsi :

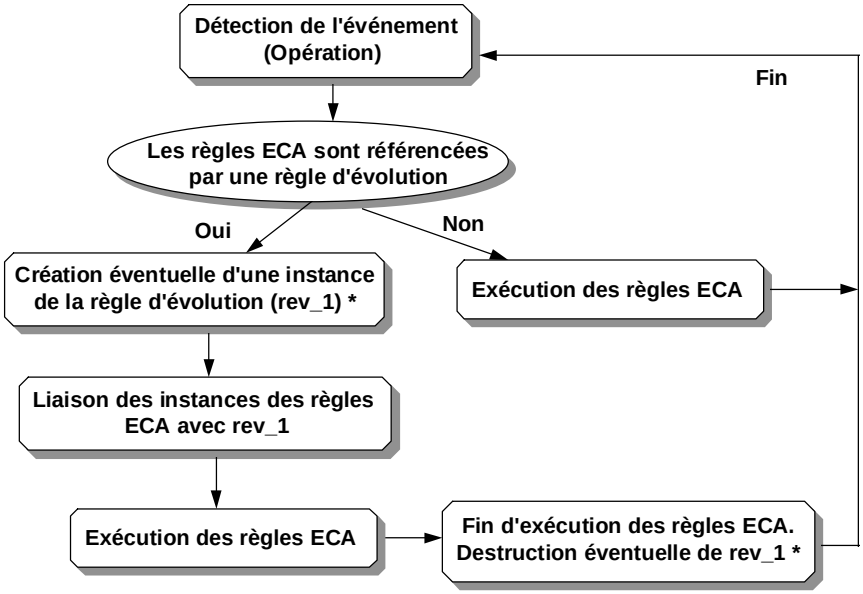


Figure 8-8 : Exécution d'une règle d'évolution

La détection de l'exécution d'une opération est assurée par le moteur d'exécution du modèle actif. Lors de la détection de l'événement correspondant, si une des règles sélectionnées est une règle référencée par une règle d'évolution, le moteur crée une règle d'évolution et établit une liaison de référence avec la règle active en cours d'exécution. Lorsque toutes les règles ECA d'une règle d'évolution sont exécutées, le moteur détruit la règle d'évolution correspondante. Pour les étapes *, l'instance de la règle d'évolution rev_1 ne sera pas créée si elle existe déjà et elle ne sera pas détruite tant que toutes les règles ECA correspondantes ne sont pas évaluées.

8.5.2 Stratégies d'évolution

Modélisation d'une stratégie

La modélisation des stratégies est réalisée en utilisant le concept de "Bases" du modèle actif (cf. Chapitre 7). A une stratégie correspond une base qui contient l'ensemble des règles ECA générées par toute les règles d'évolution définies ou redéfinies de cette stratégie. Le lien de redéfinition des stratégies correspond au lien de spécialisation des bases (Figure 8-9).

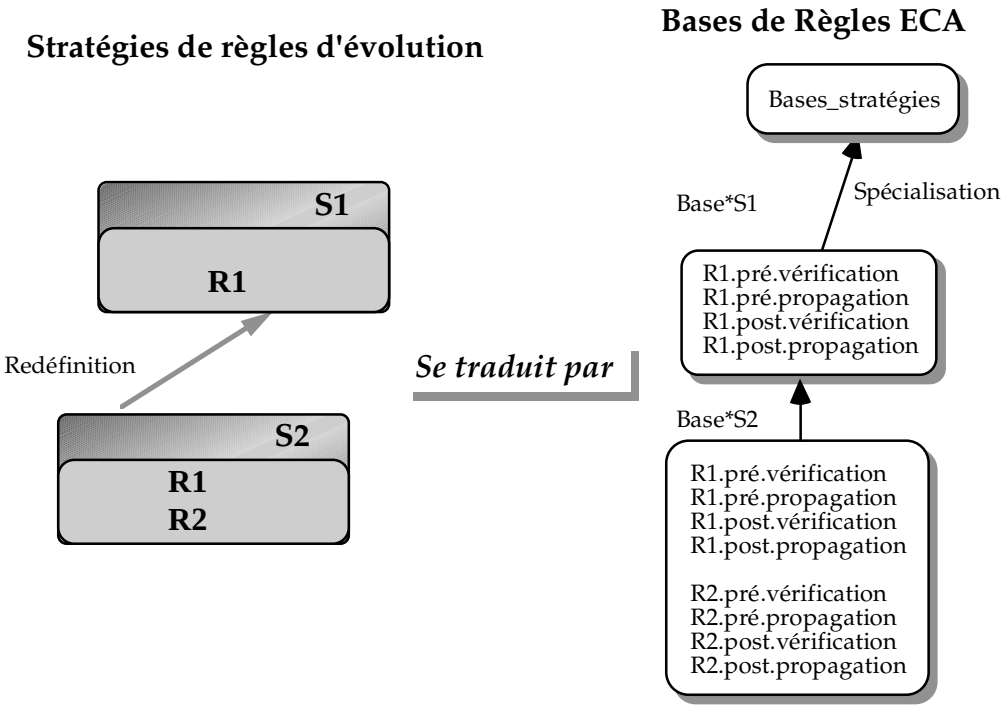


Figure 8-9 : Correspondance stratégie-base

Afin d'établir une liaison, d'une part, entre une stratégie et une base de règles, et d'autre part, entre une stratégie et ses règles d'évolution, nous modélisons une stratégie par une classe qui référencera l'ensemble de ses règles d'évolution, ainsi que la base de règle correspondante (Figure 8-10 et Ecran 8-2).

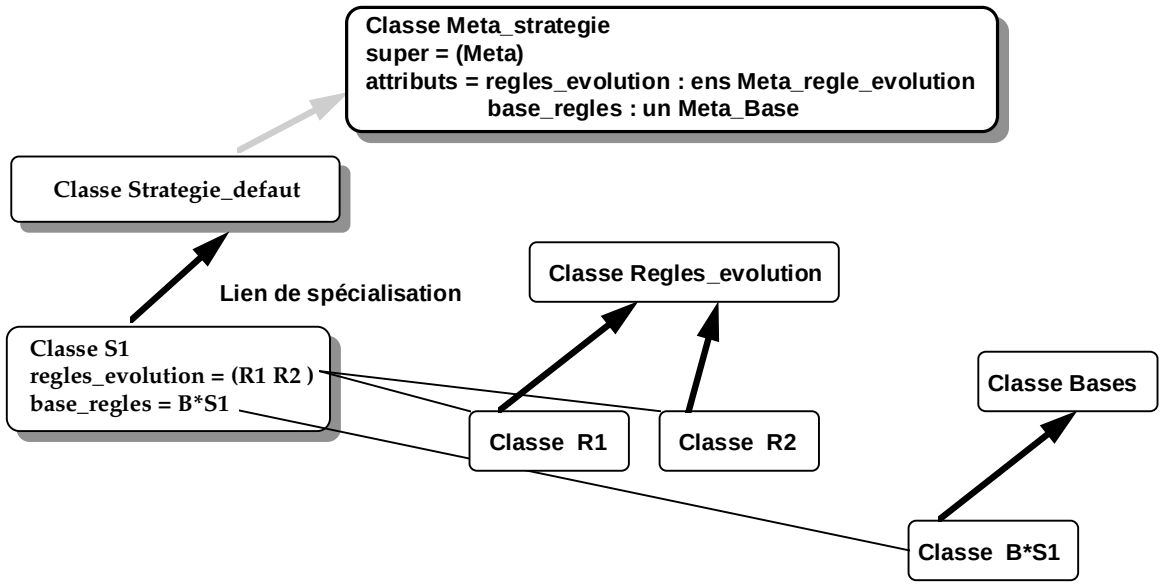
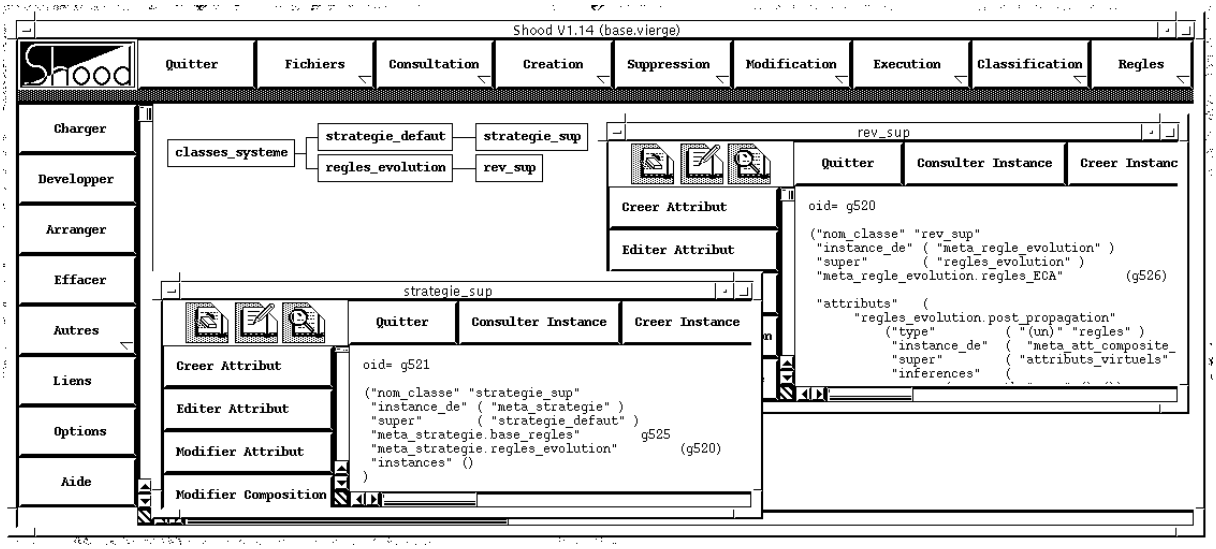


Figure 8-10 : Modélisation d'une stratégie



Ecran 8-2 : Modélisation des règles et des strategies

Algorithmes d'activation et de désactivation d'une stratégie

Lorsque l'utilisateur active une stratégie donnée (**s**), dans un premier temps on désactive les autres stratégies et on active toutes les règles de **s.** Puis, on active les règles des super-stratégies directes et indirectes, sauf celles que **s** redéfinit. L'activation (resp. désactivation) d'une stratégie correspond à l'activation (resp. désactivation) des règles d'évolution définies ou redéfinies dans cette stratégie. Et l'activation (resp. désactivation) d'une règle d'évolution correspond à l'activation (resp. désactivation) des règles ECA associées.

Activer_stratégie (s_i)

Soit S_j la stratégie courante. /* stratégie active */

Soit S_K l'ensemble de toutes les stratégies que s_i redéfinit. /* super-stratégies */

Soit R_I l'ensemble des règles (définies et redéfinies) de s_i.

Soit R_L l'ensemble des règles redéfinies par s_i.

Désactiver_stratégie (s_j).

Pour tout sk ∈ S_K :

Soit R_k l'ensemble des règles de s_k.

Pour tout $r_k \in R_k$: /* Activer les stratégies S_K*/

Activer-règle (r_k, s_k).

Pour tout $r_l \in R_L$: /* désactiver les règles R_L dans S_K*/

Désactiver_règle (r_l, s_k).

Pour tout $r_i \in R_I$:

Activer-règle (r_i, s_i). /* Activer la stratégie si*/

Stratégie-courante : = s_i.

Fin-activer_stratégie

Désactiver_stratégie (s_i)

Soit S_K l'ensemble de toutes les stratégies que s_i redéfinit.

Soit R_I l'ensemble des règles de la stratégie s_i

Pour tout sk ∈ S_K :

Soit R_k l'ensemble des règles de s_k.

Pour tout $r_k \in R_k$: /* Désactiver les stratégies S_K*/

Désactiver-règle (r_k, s_k).

Pour tout $r_i \in R_I$:

Désactiver_règle (r_i, s_i). /* Désactiver la stratégie si */

Stratégie_courante := S0 /* Stratégie par défaut */

Fin-désactiver_stratégie

Activer-règle (r, s) /* r représente une règle d'évolution */

Soit RI l'ensemble des règles de la base base*s /* Au maximum, il en existe 4*/

Pour tout $r_i \in R_I$: /* Activer les règles ECA suivantes de base*s */

(activer_regle r_i) /* Appel à la fonction Lisp pour activer une règle ECA */

Fin-activer-règle

Désactiver_règle (r, s)

Soit RI l'ensemble des règles de la base base*s /* Au maximum, il en existe 4*/

Pour tout $r_i \in R_I$: /* Désactiver les règles ECA suivantes de base*s */

(désactiver_regle r_i) /* Appel à la fonction Lisp pour désactiver une règle ECA */

Fin-désactiver-règle

Dans les fonctions Activer_règle et Désactiver_règle, on fait appel aux fonctions activer_regle et désactiver_regle du moteur d'exécution des règles ECA. Tous les algorithmes présentés ci-dessus sont implémentés par des fonctions Lisp utilisant l'interface programmable du système actif.

Les coûts des algorithmes d'activation et de désactivation sont en fonction du nombre moyen de règles d'évolution dans une stratégie et de la profondeur du graphe de stratégies. Quant aux algorithmes de traduction des règles et des stratégies d'évolution, ils sont assez coûteux mais leur exécution a lieu uniquement en création ce qui ne degrade pas les performances d'exécution.

L'évolution des règles et des stratégies d'évolution est assurée par une technique de correction ; elle est réalisée d'une part par des opérations de maniuplation et d'autre part par des règles ECA.

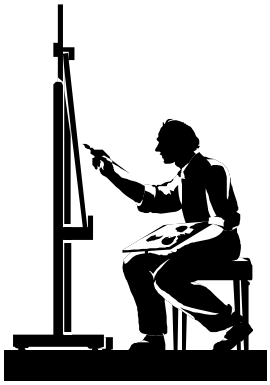
8.6 CONCLUSION

L'évolution dans SHOOD concerne l'évolution des classes et des instances. Elle est mise en œuvre par un ensemble d'opérations constituant le support d'évolution (cf. Chapitre 4). Ce dernier peut être étendu par des stratégies et des règles d'évolution. Les règles d'évolution permettent d'exprimer, de manière déclarative, la sémantique de l'évolution d'une classe ou d'une instance. Quant aux stratégies (ensemble de règles d'évolution), elles permettent de définir plusieurs sémantiques d'évolution différentes et d'en activer une seule. L'ensemble des stratégies est organisé dans un arbre de spécialisation (héritage simple) où la spécialisation a pour sémantique l'héritage ou la redéfinition des règles d'évolution. Cette organisation des stratégies offre une meilleure réutilisation des règles d'évolution.

Parmi les inconvénients de ce mécanisme de règles et de stratégies, le passage d'une stratégie à une autre peut introduire des comportements non désirés. En effet, en changeant de stratégie courante, l'utilisateur peut hériter de règles d'évolution non souhaités ; et cela peut survenir si le programmeur ne possède pas une vue globale des stratégies. Une interface graphique devrait être développée pour permettre la gestion de la hiérarchie des stratégies et la représentation de chaque règle d'évolution par un commentaire exprimant sa sémantique.

Une extension de ce travail est de définir une bibliothèque de stratégies d'évolution. Il existe plusieurs stratégies d'évolution connues, généralement implémentées dans les systèmes tels que ENCORE [Skarra87], ORION [Banerjee87] et GEMSTONE [Penney87]. On pourrait répertorier ces stratégies pour qu'elles soient utilisables directement, et bien sûr redéfinissables par le programmeur. De cette manière, on pourrait simuler la stratégie d'ENCORE (expérience réalisée dans le cadre de la thèse de [Ahmed-Nacer94]).

Une lacune est due au fait que la sémantique des règles est exprimée dans les conditions et dans les actions de manière procédurale. Cette lacune ne permet pas de vérifier que deux règles sont incompatibles, c'est-à-dire qu'elles ne peuvent s'exécuter simultanément. Une perspective, mais à plus long terme, serait alors de vérifier la cohérence d'une stratégie.



9. EXPERIMENTATION

L'expérimentation du système d'évolution se situe dans le cadre de la conception d'un système à base de connaissances pour des applications du bâtiment [Culet93]. L'objectif est de permettre la mise au point technique et économique d'un projet de construction par une mise en commun des propositions émises par les différents corps de métiers pour répondre au descriptif fourni par le maître d'oeuvre. La base de connaissances, appelée DUA (Dossier Unique Actualisé), se présente alors comme un document de dialogue, de validation et de stockage d'informations concernant une opération immobilière.

Notre démarche pour le développement du DUA consiste à mener en parallèle la modélisation des connaissances structurelles et comportementales communes ou particulières aux différents intervenants et le développement d'un prototype permettant une validation du système [Culet94]. En effet, la modélisation et le prototypage se font en parallèle ce qui nécessite, d'une part, une transformation sans perte entre le modèle et le système de représentation, et d'autre part, de bénéficier d'un outil informatique suffisamment évolutif pour que le prototypage puisse entrer dans le cycle de développement du système à base de connaissances.

Cette application est décrite dans cette thèse pour présenter une utilisation de la dynamique de SHOOD. L'évolution du système SHOOD a permis, d'une part, la réalisation d'une maquette par prototypage incrémental [Dieng90], et d'autre part, le contrôle de l'intégrité des descriptifs et la gestion des variantes de l'opération immobilière [Bounaas95b]. Cette application utilise principalement le support d'évolution de SHOOD, ainsi que les règles actives.

9.1 LE CADRE DE RECHERCHE

Le projet concerne la spécification, l'élaboration et l'expérimentation d'un modèle de représentation et d'échange de données techniques et économiques entre PME d'un groupement dans le domaine du bâtiment. Son objectif est de permettre la mise au point technique et économique d'un projet de construction.

D'un point de vue organisationnel, l'enjeu porte sur la mise en accord entre partenaires différents, à la fois sur des objectifs communs et sur les interactions entre activités

[Bounaas95b]. En permetant d'établir, de chiffrer et de gérer la description technique et la production d'un ouvrage, le prototype visé a pour objectif d'organiser et codifier les nombreux allers et retours entre les différents corps de métier, le groupement de PME, les conducteurs de travaux et les acteurs extérieurs tels que la maîtrise d'œuvre et d'ouvrage.

9.1.1 Le Dossier Unique Actualisé

La base de connaissances, appelée DUA (Dossier Unique Actualisé), se présente comme un document de dialogue, de validation et de stockage d'informations concernant une opération immobilière. Il rassemble les propositions techniques et économiques faites par chaque entreprise pour son lot propre et coordonnées par le groupement, à partir du descriptif de l'ouvrage établi par le maître d'oeuvre. Un lot regroupe un ensemble (ou sous-ensemble) d'ouvrages relatifs à un métier (ex : lots "menuiserie intérieure", "menuiserie extérieure", "plomberie", "charpente", "cloisons", ...). Ces données font l'objet de négociations, jusqu'à l'obtention d'un accord avec le commanditaire en matière de descriptif et de coût. Ce dossier doit donc être en mesure d'accueillir toutes sortes de variantes en terme de prestations. Support d'accords successifs, il intervient tout à la fois en qualité d'interface entre le montage d'une opération (apport en terme de facilité commerciale) et la gestion du chantier (disposer d'un document conforme à la proposition) et comme mémoire des modifications du projet (document de référence) pendant toute la période de vie de l'ouvrage pour les entreprises.

C'est un document "unique" dans le sens où les informations qu'il contient sont mises en cohérence et centralisées. Il est "actualisé" dès lors que les variantes au descriptif de base proposées par les entreprises se substituent à la description initiale, une fois celles-ci validées par la maîtrise d'oeuvre. Ces variantes de prestations sont proposées par les entreprises lors de la phase de négociation d'une affaire, dans le but de réduire le coût d'un ouvrage. D'autres variantes peuvent être également réalisées sur le chantier pour prendre en compte de nouvelles demandes de prestations. Une variante peut induire une incohérence globale qui doit pouvoir être détectée.

9.2 LA DEMARCHE

La démarche engagée de conception de la base de connaissances est itérative et incrémentale : "itérative" dans le sens où la démarche se décompose en étapes successives par lesquelles on passe incessamment, "incrémentale" pour signifier que la base de connaissances s'enrichit de nouvelles informations à chaque tour [Culet94].

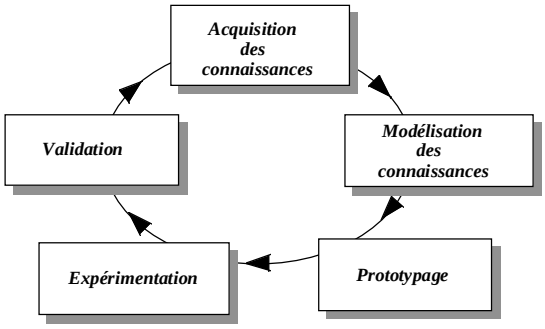


Figure 9-1 : Une démarche itérative et incrémentale

- *Acquisition de connaissances*. Les deux sources essentielles de connaissances sont les documents existants et la description orale des pratiques professionnelles faites par les experts du domaine considéré. Il est demandé aux entrepreneurs d'exposer les différentes solutions techniques de leur métier et leur manière d'aborder un dossier. A ce stade, il s'agit de comprendre la manière selon laquelle se structure l'activité de chaque corps de métier ; un travail d'autant plus délicat que les experts ont rarement une conscience explicite des principes qui organisent leurs pratiques professionnelles [Dieng90] [Schreiber91].

- *Modélisation des connaissances*. Nous avons adopté le modèle SHOOD pour réaliser cette modélisation. Nous nous sommes donc attachés à structurer les concepts identifiés lors de l'étape précédente et expliciter leur relations. Nous avons défini les concepts génériques propres au domaine de la construction (ouvrage, produit, ...) et les concepts spécifiques à chaque corps de métier (fenêtre, poignée...). Le modèle doit refléter l'organisation des connaissances adoptée par les experts et doit rendre compte de leur utilisation.

- *Prototypage*. Le système SHOOD autorise une approche par prototypage incrémental qui permet de voir systématiquement les résultats de l'analyse dès qu'ils sont établis. Cela permet de faciliter la phase ultérieure de validation .

- *Expérimentation*. Les experts utilisent le prototype sur des cas simples, puis plus complexes. Cette étape permet d'établir une communication plus rapide entre les différents intervenants du projet. Les experts apprennent aussi à mieux juger de la pertinence des informations qu'ils divulguent. Cette étape est source de motivation, puisqu'ils peuvent voir concrètement se dessiner l'esquisse du prototype en cours de définition.

- *Validation*. Cette étape fait le bilan des connaissances acquises et valide leur bien-fondé. Un constat d'erreur dans la représentation des connaissances implique d'entrer dans une nouvelle boucle sur les mêmes thèmes de travail ; si, par contre, la représentation est satisfaisante, de nouveaux thèmes sont abordés. Cette étape est souvent à l'origine de la définition de nouvelles demandes : suggestions en matière d'ergonomie, élargissement des fonctionnalités, etc.

Compte tenu de la quantité d'informations représentant le DUA, une structuration de ces informations est nécessaire pour manipuler cette base de connaissances. La mise en œuvre de cette dernière doit tenir compte des différentes contraintes assurant la cohérence du DUA.

9.3 LA MODELISATION

9.3.1 Description structurelle

Le concept élémentaire manipulé est celui d'*ouvrage*. Il correspond à la plus petite entité pour laquelle une entreprise propose un prix (exemple : une baignoire totalement équipée et mise en oeuvre sur le chantier). Les renseignements qui lui sont associés sont commerciaux (prix, quantités) ou techniques (produits utilisés, particularités de pose, localisation). Il peut exister plusieurs occurrences d'un même ouvrage (exemple : porte d'entrée et porte d'intérieur).

Un ouvrage est composé de différents *produits*, au sens de produit manufacturé (exemple : l'ouvrage porte est composé de plusieurs produits : un ouvrant, un dormant, une serrure, une poignée, etc.) ayant eux-mêmes des caractéristiques techniques.

Pour satisfaire le besoin d'organisation des ouvrages, il faut pouvoir les regrouper autour d'entités plus vastes (les ouvertures sont l'ensemble des portes, des fenêtres et des porte-fenêtres) qui peuvent être considérées comme des ouvrages généraux (hiérarchie à plusieurs niveaux). En terme de concepts objet, un ouvrage est représenté par une classe, qui peut admettre des sous-classes. Les attributs associés permettent de décrire les ouvrages avec en particulier les produits qui les composent (Figure 9-4, sous-graphe ouvrage) et de citer les variantes proposées.

Les produits composants un ouvrage sont des produits au sens générique (un type d'ouvrant, un modèle de serrure). Ils peuvent donc être partagés par plusieurs ouvrages (exemple : les fenêtres et porte-fenêtres possèdent le même type de poignée) et ils dépendent existentiellement du ou des ouvrages qu'ils composent. Les produits sont modélisés par des classes dont les attributs représentent leurs caractéristiques jugées pertinentes, c'est à dire indispensables pour fixer le prix des ouvrages qu'ils composent et pour leur réalisation. L'organisation de l'ensemble des produits en un graphe de spécialisation permet de factoriser les caractéristiques communes. Une instance d'une classe produit représente un type de produit (un type d'ouvrant, un modèle de serrure) (Figure 9-4, sous-graphe produit).

Certaines caractéristiques de produits sont calculées en fonction d'autres caractéristiques de produits entrant dans la composition du même ouvrage ou d'autres ouvrages. Ainsi, le cadre dormant des ouvertures doit avoir un profil approprié pour un recouvrement sur un doublage de cloison d'une épaisseur donnée. En associant une inférence à l'attribut "épaisseur" du dormant, celui-ci est automatiquement calculé une fois l'épaisseur du doublage connu. D'autre

part, des contraintes sont également associées aux caractéristiques de produits, celles-ci permettant de vérifier la cohérence des valeurs fournies. Par exemple pour les cloisons, l'épaisseur des parements, la nature et l'espacement des montants sont contraints par la hauteur des pièces.

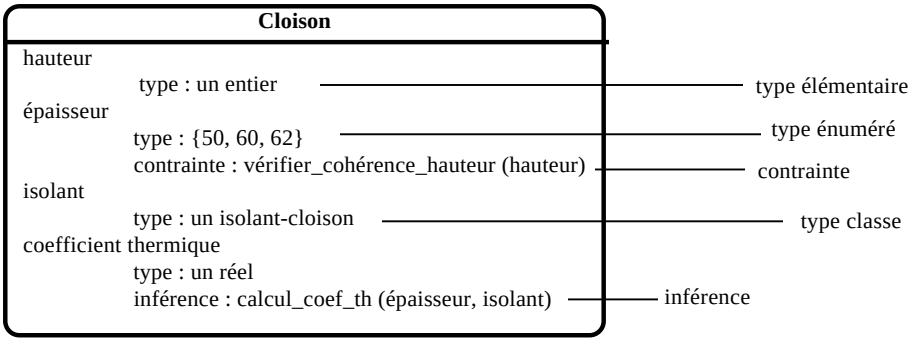


Figure 9-2 : Contraintes et Inférences

Dans l'exemple ci-dessus, la contrainte est intra-attribut, sa vérification est gérée par les opérations de création et de modification. Pour les contraintes inter-classes, elles sont exprimées et vérifiées par des règles ECA. Par exemple, la couleur de la baignoire doit être la même que celle du lavabo.

Type de règle : Vérification

Evénement : Appel Valuer_attribut
Oid : Accessoires
nom_att : Baignoire, Lavabo

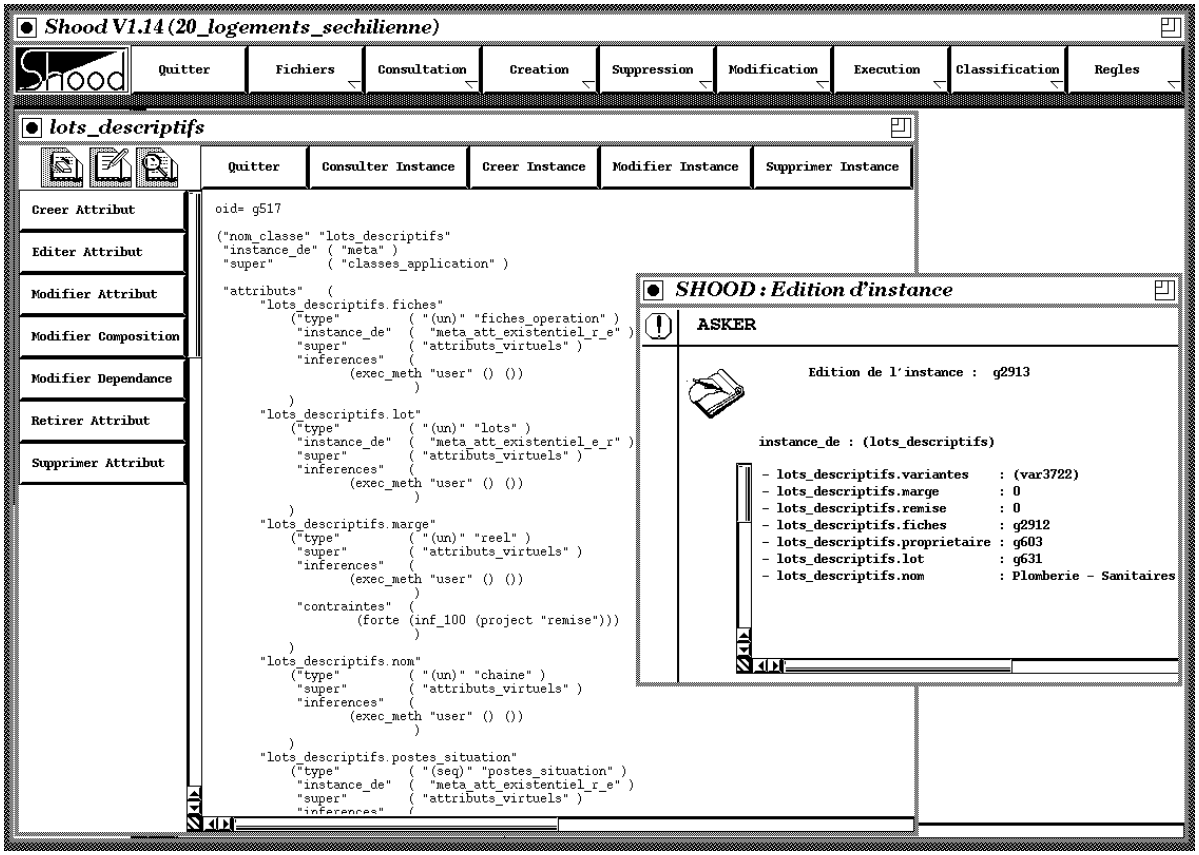
Condition : (= (Project oid "Baignoire>couleur" (Project oid "Lavabo>couleur")

Action : La couleur de la baignoire n'est pas la même que celle du lavabo

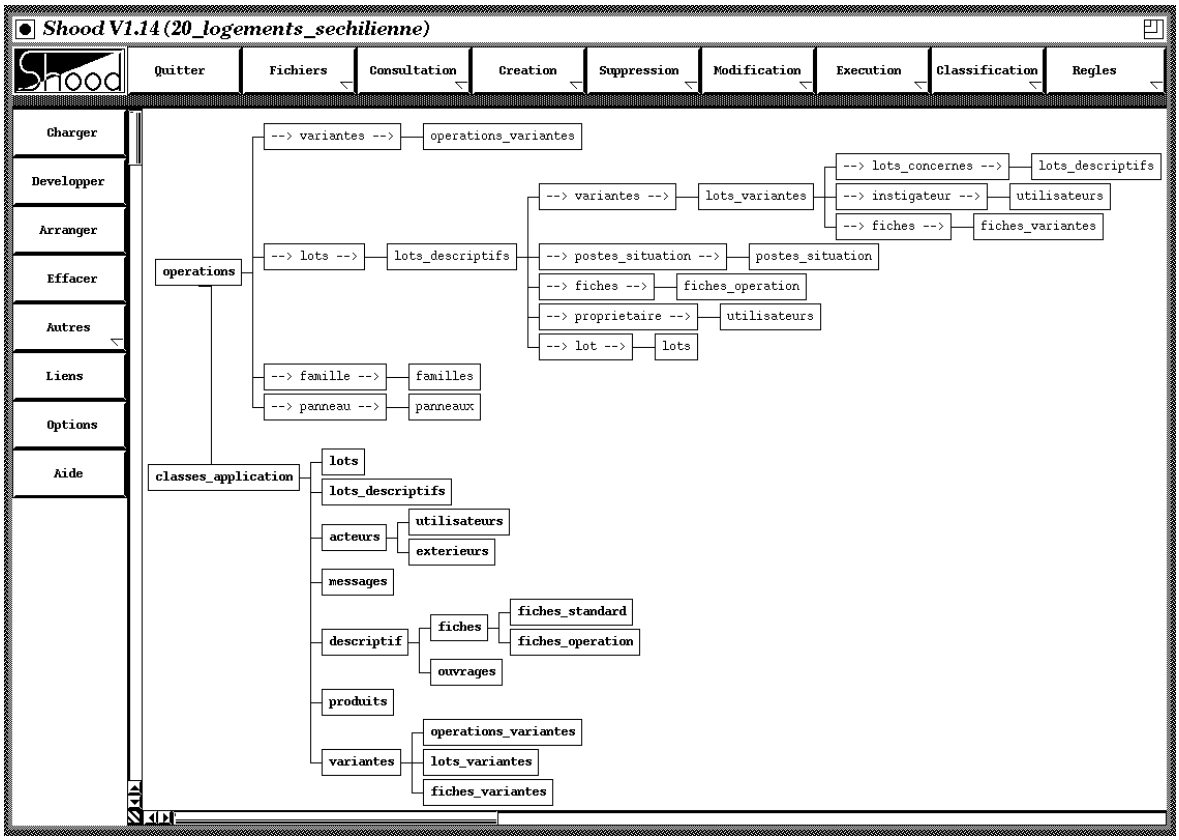
Figure 9-3 : Contrainte inter-classes

L'ensemble des ouvrages est décrit lot par lot par un document structuré hiérarchiquement (voir Ecran 9-4). Ce document est composé d'un ensemble de paragraphes. Un paragraphe est défini par une instance de la classe *Fiche* dont les attributs sont : des généralités, une référence vers les ouvrages décrits et le prix total correspondant à l'ensemble des ouvrages cités. L'ensemble des paragraphes est organisé en un graphe de composition, ceci donnant la possibilité de factoriser certaines informations. De plus un paragraphe est dépendant existentiellement de ses sous-paragraphes. Cette dépendance permettra, lors de la destruction d'un paragraphe, une destruction automatique des sous-paragraphes.

Ces paragraphes sont créés dans un premier temps pour construire une bibliothèque de paragraphes standards qui pourront être utilisées (par duplication) pour construire des paragraphes pour une opération immobilière.



Ecran 9-1 : Modélisation d'un lot descriptif



Ecran 9-2 : Modélisation d'un lot descriptif et d'une opération

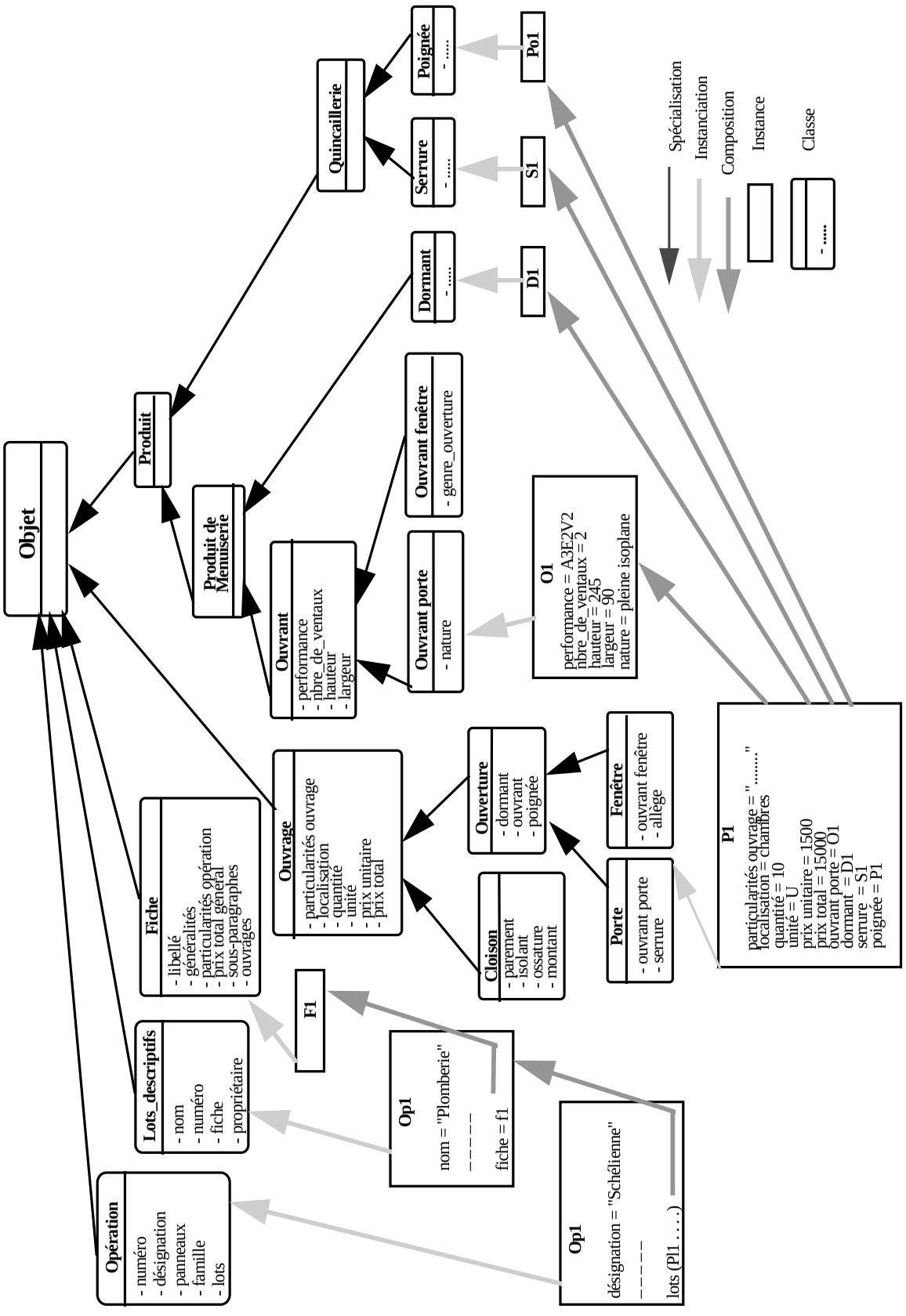


Figure 9-4 : La modélisation ouvrage/produit

Un lot (instance de la classe *Lots_descriptifs*) est alors décrit par une référence vers la racine des fiches et des informations générales telles que le propriétaire, le nom du lot, etc. De même, une opération (instance de la classe *Operation*) est décrite par une référence vers l'ensemble des lots descriptifs et des informations sur l'opération telles que la désignation, la famille et la fiche signalétique (instance de la classe *Panneaux*) de l'opération. Cette base de connaissances est utilisée par N+1 acteurs, N acteurs représentant les différents propriétaires des lots et un représentant du groupement responsable de l'opération.

9.3.2 Les variantes

Des variantes de prestations, proposées par les entreprises lors de la phase de négociation ou devant être réalisées ultérieurement sur le chantier, peuvent avoir des conséquences sur la définition d'autres prestations [Bounaas95b]. Il n'est pas question ici de réaliser automatiquement les mises à jour, conséquences de la prise en compte d'une variante sur les lots connexes, car chaque entreprise reste maîtresse de ses choix de produits. En terme d'usage, nous avons donc opté pour le déclenchement d'alarmes afin d'alerter les lots concernés par l'introduction d'une variante. Ces alarmes sont déclenchées dès qu'une variante d'un lot est créée.

Une variante d'un lot donné correspond à la modification, l'ajout ou la suppression d'un ouvrage ou d'un paragraphe du descriptif de ce lot. Cette variante est traduite par la duplication des paragraphes modifiés ou ajoutés. Pour cela, nous avons défini un autre type de paragraphe contenant en plus un commentaire sur la variante et un indicateur précisant si cette variante correspond à la suppression d'un paragraphe ou non. Une variante peut être vue comme une version d'instance d'un paragraphe [Agrawal90] [Katz87] [Bounaas91]. Les paragraphes variantes sont instances de la classe *Fiche_variantes* (Figure 9-9).

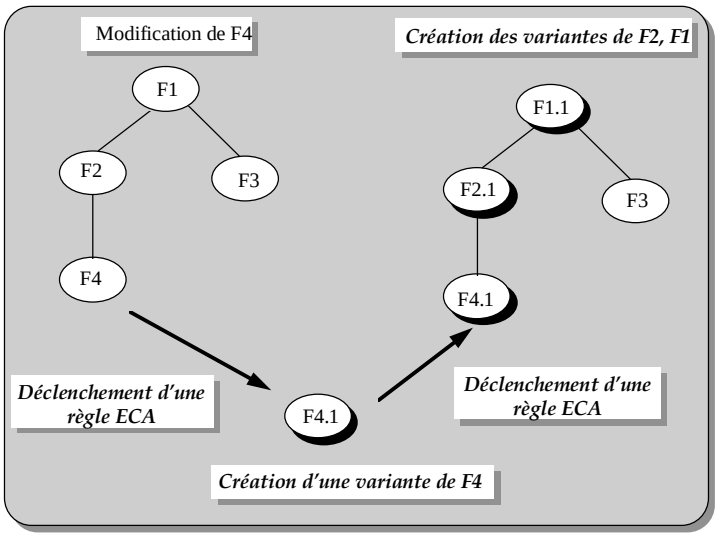


Figure 9-5 : Schéma de déclenchement des règles ECA

L'ensemble des paragraphes variantes constitue une hiérarchie d'instances de la classe *Fiche_variantes (*voir Figure 9-5). La création de cette hiérarchie de paragraphes est rendue automatique par l'utilisation des règles ECA. En effet, lorsqu'un paragraphe d'un lot est modifié et que l'utilisateur se trouve dans une phase de création de variante, le système déclenche une règle pour dupliquer le paragraphe altéré et donc créer une instance de la classe *Fiche_variante* (Figure 9-6).

Type de règle : Propagation

Evénement : Retour Modifier_instance
Source : oid : Fiche

Condition : (= Mode_Variante 'T)

Action : (dupliquer_fiche oid oid_variante)

Figure 9-6 : Création d'une variante de paragraphe

Comme les paragraphes sont organisés dans une hiérarchie, la duplication d'une variante de paragraphe entraîne la duplication des paragraphes les plus généraux, si ces derniers n'ont pas déjà été dupliqués. Cette duplication est réalisée par une règle ECA (Figure 9-7). Lorsque l'utilisateur valide l'édition de la variante d'un lot, la hiérarchie de variantes de paragraphes est totalement construite.

```
Type de règle : Propagation

Evenement :  Utilisateur dupliquer_fiche_apres
             Source : oid : Fiche
                      oid_variante : Fiche_variante
Variables : pere_paragraphe
Condition : (SETQ pere_paragraphe (ex_pere_fiche oid))

Action : (SETQ pere_variante (existe_variante pere))
         (IFN pere_variante
              (dupliquer_fiche pere pere_variante)
              )
         (inserer_fils pere_variante oid_variante)
```

Figure 9-7 : Construction de la hiérarchie de variantes

Cette règle se déclenche après l'exécution de la fonction dupliquer_fiche. Cette règle vérifie qu'il ne s'agit pas de la duplication de la racine de la hiérarchie (Condition), puis, dans le cas où il s'agit d'un autre paragraphe, elle vérifie l'existence d'une variante du paragraphe au dessus, le duplique et l'insère dans la hiérarchie.

Le principe de structuration d'un descriptif de lot est conservé pour les variantes. Une variante de lot (instance de la classe *Lot_variante*) (Figure 9-9) référence la racine de la hiérarchie des fiches variantes et contient, entre autres, des informations sur l'instigateur, le

statut (en étude ou validée) de cette variante et les éventuels lots concernés en cas de variante infuente (variante qui induit la création d'autres variantes).

Une fois que la variante est créée, le système, en utilisant les règles ECA, envoie un message d'alerte vers les propriétaires des lots concernés afin qu'ils génèrent à leur tour une variante induite par la variante en cours. Pour ce faire, nous avons utilisé les règles ECA qui permettent d'exprimer que la création d'une variante (Evénement) entraîne l'envoi de messages aux lots concernés par ce changement, en indiquant les ajustements à réaliser (Action).

Type de règle : Propagation

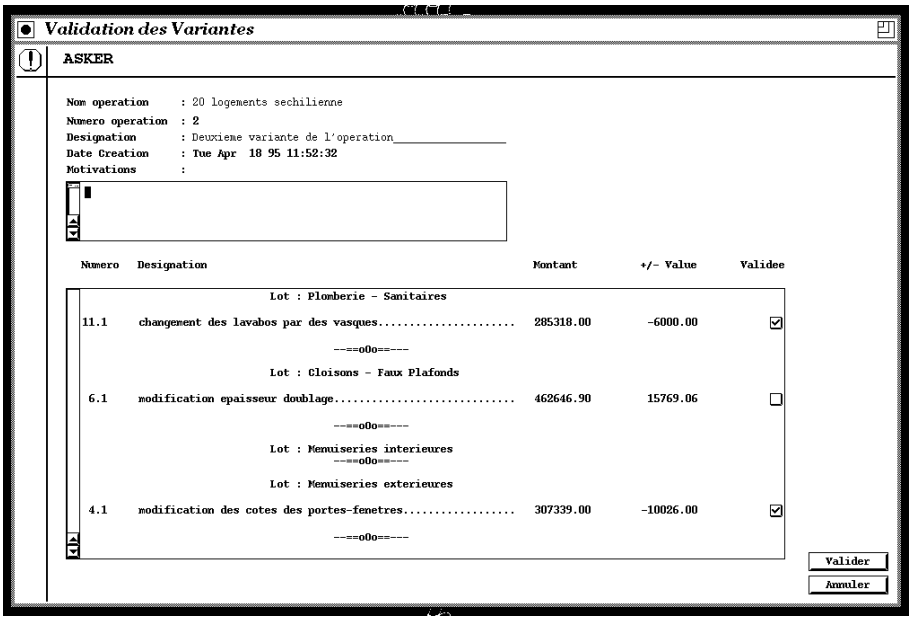
Evénement : Utilisateur Créer_variante_après
Source :ouvrage : Ouvrage

Condition : (= (Project ouvrage "désignation") "Ventilation Mécanique Controlée")

Action : Alerter_lot ("menuiserie_exterieure" "incorporation de grille")
Alerter_lot ("couverture" " chapeaux sortie en toiture")

Figure 9-8 : Exemple d'alarme

Lors du passage d'un chauffage à eau chaude à un chauffage électrique (création d'une variante), il faut installer une ventilation mécanique contrôlée (Condition). Ceci entraîne l'alerte des lots menuiserie_extérieure et couverture pour incorporer respectivement des grilles et des chapeaux de sortie en toiture (Action).



Ecran 9-3 : Validation d'une variante

Lorsqu'un ensemble de variantes lots est choisie pour être validé (Ecran 9-3), nous construisons une variante de l'opération en créant une instance de la classe *Operation_variante*, qui référencera les différentes variantes de lot choisies. Les références

des différentes variantes de l'opération sont sauvegardées dans une instance de la classe *Operation*. Ceci permettra de connaître l'historique des différentes variantes générées pour une opération donnée.

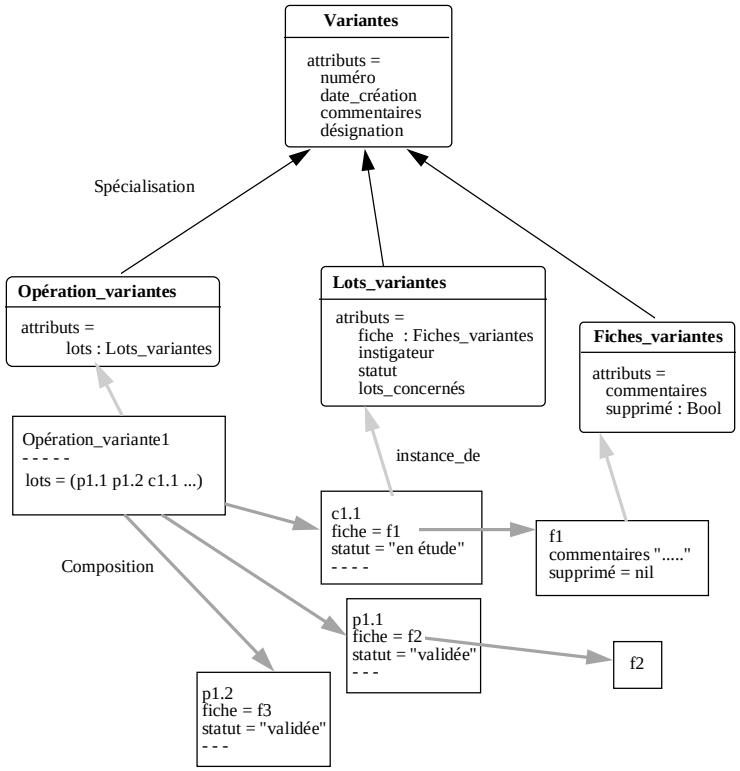


Figure 9-9 : Description des variantes

Les variantes ne sont pas les seuls cas d'utilisations des règles actives : des alarmes doivent être définies pour la recherche des "doublons". En effet certaines prestations peuvent apparaître dans deux lots distincts. Par exemple, le produit "vitres" peut apparaître à la fois dans le lot menuiserie (pour une porte vitrée, par exemple) et dans le lot vitrerie. Pour cela, lors de la description d'un ouvrage dans un lot donné (Evénement), on vérifie que celui-ci n'est pas présent dans un autre lot (Condition) ; dans le cas d'un doublon on alerte l'entreprise générale (Action).

Type de règle : Propagation

Evénement : Appel Creer_instance
Source : oid : Ouvrage

Variables : selection

Condition : (setq selection (selectionner_fiche (Project oid "designation"))

Action : (alerter_lot "Entreprise_generale"
"doublons"
(Project oid "designation")
(Project selection "designation"))

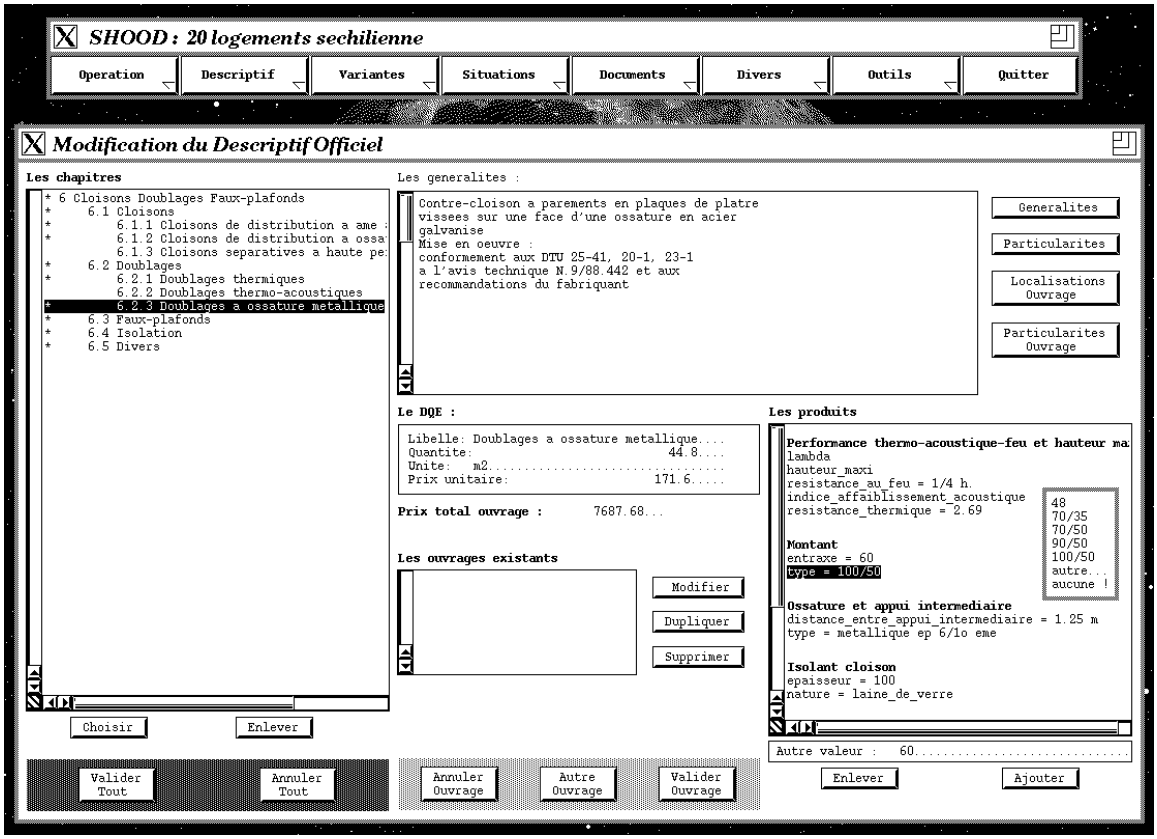
Figure 9-10 : Recherche de doublons

A chaque fois qu'un ouvrage est créé, on vérifie l'existence d'un doublon, en appelant une fonction de recherche dans les paragraphes (instance de Fiche) de tout le DUA.

9.4 LA MAQUETTE

Après chaque étape de modélisation, les résultats sont introduits dans le système SHOOD pour vérifications et mise en cohérence de la modélisation. Cette dernière est expérimentée par les experts qui se retrouvent en position d'utilisateurs.

Bien qu'il existe une interface graphique de haut niveau pour les concepteurs de la base de connaissances, celle-ci reste très liée aux concepts offerts par SHOOD. Pour ce projet, les concepts objet doivent être totalement occultés, car hors des pratiques d'usage des acteurs du BTP. A cette fin, une nouvelle interface graphique adaptée au monde de la construction a dû être construite. Cette dernière permet de manipuler non plus des classes, des instances ou des méthodes, mais tout simplement des ouvrages, des produits, … comme le montre l'Ecran 9-4.



Ecran 9-4: Edition du descriptif d'un lot

Cette interface comprend :

- Une interface de saisie des informations concernant l'opération : la fiche signalétique, la sélection des lots de l'opération et l'affectation des entreprises aux lots.

- Une grille de saisie des données techniques et commerciales d'un lot donné, avec des facilités de saisie telles que la proposition des ouvrages standards les plus courants et la proposition d'un ensemble de valeurs possibles pour les caractéristiques de produits.

- La consultation des Descriptifs Quantitatifs et Estimatifs (quantité, prix unitaire des ouvrages) lot par lot ou pour toute l'opération.

- La gestion des situations de travaux ou état d'avancement de l'opération, ainsi que la gestion des variantes de l'opération (création, suppression d'une variante ....).

- Un module de messagerie entre utilisateurs de la base de connaissances, leur permettant de communiquer et d'être averti de tout changement de descriptif de l'opération. Cette messagerie est géré par des boîtes aux lettres.

En offrant une interface graphique dédiée, nous expérimentons la base de connaissances au fur et à mesure de sa conception et nous facilitons ainsi la phase de validation.

9.5 CONCLUSION

L'application présentée dans ce chapitre a pour cadre la conception d'une base de connaissances pour le bâtiment. Le principal objectif est de permettre la mise au point technique et économique d'un projet de construction où interviennent plusieurs corps de métiers.

Nous nous sommes attachés à définir une démarche de conception itérative et incrémentale de la base de connaissances. Le cycle de conception basé sur les 5 étapes successives d'acquisition, de modélisation, de prototypage, d'expérimentation et de validation a d'ores et déjà été parcouru plusieurs fois. Cette démarche présente l'avantage d'intégrer le prototypage dans le cycle de développement, ce qui nous semble indispensable dans un contexte applicatif où des experts de métier, et donc de savoir et de compétences diverses, doivent coopérer.

L'approche par prototypage incrémental est rendue possible par l'utilisation:

- du modèle de représentation de connaissances SHOOD, évolutif et extensible

- du système SHOOD permettant de manipuler les différentes représentations

Une interface graphique spécialement conçue pour cette application facilite les étapes d'expérimentation, mais aussi de validation et d'acquisition dans un nouveau cycle de conception.

La modélisation obtenue repose sur deux composantes. La composante structurelle décrit les connaissances déclaratives communes et spécifiques au différents experts. La composante comportementale décrit les interactions entre les différents lots, lors de l'apparition de variantes ou de doublons, durant les étapes de négociation technique et commerciale. Ces deux composantes sont indépendantes, de manière à pouvoir réutiliser facilement la description structurelle lors de la poursuite de nos travaux.

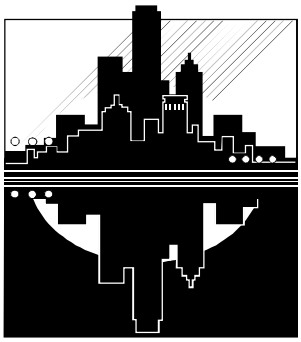
La suite de l'étude porte sur deux thèmes :

- Le premier a pour objectif la validation du DUA par d'autres experts de lots différents. En effet la base de connaissances actuelle a été construite à partir de trois lots (menuiserie, plomberie et cloison-doublage).

- Le deuxième thème vise l'utilisation et l'enrichissement du DUA pour et lors du suivi de chantier. Une première approche nous a révélé que si la description structurelle du bâtiment reste semblable, la description comportementale semble fortement dépendante du contexte applicatif. Les alarmes nécessaires au suivi de chantier ne sont pas identiques à celles ayant permis le maintien de la cohérence du dossier lors des phases de négociation. De plus, de nouvelles fonctionnalités devront être introduites, par exemple pour exploiter les localisations (emplacement des ouvrages) afin de faciliter le suivi de chantier.

Le caractère évolutif du modèle SHOOD lui confère le pouvoir de prototyper "rapidement" un système à base de connaissances. En effet, le schéma établi dans la phase de modélisation est directement traduit dans le langage de représentation et peut être modifié dynamiquement. Par conséquent, il est possible de faire évoluer à tout moment le prototype par enrichissement et remise en cause des connaissances acquises. De ce fait, le processus d'acquisition peut être cyclique et incrémental, avec un prototypage s'effectuant parallèlement au processus de modélisation.

Cette application du DUA a permis d'expérimenter, d'une part, le prototype SHOOD sur un cas "réel" (295 classes et 1200 instances), et d'autre part, de tester l'évolution des structures et données, ainsi que son extensibilité, puisqu'il a fallu enrichir les fonctionnalités de base (par exemple, de nouveaux constructeurs comme séquence et liste).



10. CONCLUSION

Cette étude tente d'apporter des solutions aux problèmes d'extension et de réutilisation des mécanismes d'évolution dans les bases de connaissances. Nous proposons un système d'évolution permettant la définition et la mise en œuvre de l'évolution. Cette mise en œuvre est réalisée par un ensemble de mécanismes tels que la classification et les règles actives, ainsi que par un ensemble d'opérations de manipulation : le support d'évolution. Nous offrons au concepteur d'applications, par la proposition d'un mécanisme de règles et de stratégies d'évolution, le moyen de programmer des contraintes d'évolution de structures ou de données de manière déclarative. Le concepteur peut définir des stratégies pour regrouper ces contraintes. Il peut, suivant ses besoins, faire cohabiter plusieurs stratégies, mais une seule sera active à un moment donné.

Le système d'évolution, tel qu'il est présenté, est caractérisé principalement par les propriétés suivantes :

Convivialité : Possibilité d'exprimer la sémantique de l'évolution à plusieurs niveaux d'abstraction. Cette expression peut être exprimée par des règles ECA ou des règles d'évolution.

Extensibilité : On peut étendre facilement l'architecture du système d'évolution, soit par des stratégies d'évolution, soit par des mécanismes d'évolution qui peuvent être mis en œuvre en utilisant le support d'évolution.

Multi-Stratégie : Les règles d'évolution seront définies dans des stratégies qui peuvent être spécialisées par modification ou ajout de règles d'évolution. Une stratégie peut s'appliquer à une classe ou à un certain type de classes (instance d'une métaclasse particulière), de même qu'on peut définir une stratégie d'évolution pour les instances d'une classe (se résumant à un ensemble de contraintes, par exemple). L'ensemble des stratégies est organisé dans une hiérarchie simple avec héritage et/ou redéfinition des règles contenues dans les stratégies. Cette organisation offre une meilleure réutilisabilité des règles d'évolution.

Uniformité : Toute évolution s'exprime de manière uniforme pour les différents concepts du modèle (classe, métaclasse, instance). En effet, dans ce système d'évolution, les règles

d'évolution des classes et des instances ne sont pas séparées, car c'est la sémantique des opérations qui les différencie.

Certaines des propriétés caractérisant un système d'évolution ne sont cependant pas complètement assurées. En effet, les propriétés de cohérence et de complétude ne sont assurées que partiellement, car il n'est pas possible actuellement d'affirmer qu'une stratégie est cohérente et complète. Pour permettre cela, il faudrait pouvoir détecter que deux règles d'évolution sont incompatibles et que les règles d'évolution d'une stratégie expriment bien tous les cas d'évolution.

La mise en œuvre de ce système a été facilitée par la réalisation d'un mécanisme de règles actives. En fait, le système d'évolution proposé dans cette thèse peut être vu comme une contribution à l'utilisation des règles actives pour l'évolution de schémas et d'instances [Bounaas95c].En ce sens, il contribue à une utilisation plus générale des règles actives, limitées jusqu'à présent au contrôle de l'intégrité.

BILAN

Ce travail nous a conduit à étudier dans un premier temps les différentes approches de gestion de l'évolution de schémas et d'instances dans les bases de données et les bases de connaissances. Pour l'évolution de schémas, nous avons opté pour une approche par correction où les classes sont immédiatement modifiées (sans création de versions) et où toutes les instances sont converties immédiatement après la modification des classes. Quant à l'évolution des instances, elle est régie, d'une part, par un ensemble de contraintes et de calculs inférentiels intra-classe, et d'autre part, par un mécanisme de classification d'objets évolutifs.

Pour permettre la manipulation des concepts du modèle SHOOD, nous avons construit un ensemble d'opérations respectant les invariants de ce modèle. Cet ensemble d'opérations est appelé : **Support d'évolution**. L'ensemble des opérations sur les classes a pour but de modifier la définition des classes, de répercuter cette modification sur les sous-classes et éventuellement d'assurer l'impact sur les instances des classes modifiées. Pour les instances, les opérations de manipulation permettent de créer, supprimer et modifier les valeurs d'une instance donnée. Du fait de l'existence de la multi-instanciation, nous autorisons le changement de classes d'appartenance d'une instance.

Au-dessus de ce support d'évolution, nous avons proposé un mécanisme de **Classification d'instances** [Bounaas94b] [Liotard93], qui permet de trouver les classes d'appartenance d'une instance incomplète et/ou incohérente en fonction des informations qu'elle détient. Ce

mécanisme utilise les inférences et les contraintes définies au niveau de la hiérarchie de classes ; il fournit en résultat l'ensemble des classes possibles pour l'instance à classifier.

Non-extensibilité des mécanismes d'évolution

A ce stade de l'étude, l'évolution des classes et des instances présente principalement les inconvénients suivants :

- Pour l'évolution de schémas, l'évolution est figée et peut difficilement être modifiée pour l'adapter à une application. En effet, les opérations de mise à jour de schémas (suppression de classe, ajout d'attribut, etc.) sont "câblées" dans le code du système, ce qui ne facilite pas son extension. De plus, une seule stratégie, est proposée pour toutes les classes ; il n'est pas possible de disposer de plusieurs stratégies et de choisir celle qui est la mieux adaptée aux besoins d'une application.

- Pour l'évolution des instances, les contraintes et les inférences intégrées dans le modèle de données sont insuffisantes. Elles ne permettent pas, par exemple, l'expression de contraintes inter-classes.

L'extension du mécanisme de classification pour prendre en compte les objets composites n'est pas aisée, et pour cause : le mécanisme est aussi "câblé".

Expression déclarative de l'évolution

Pour remédier au manque d'extensibilité des mécanismes actuels, nous avons proposé un modèle à base de **règles actives** qui permet l'expression déclarative des vérifications et des propagations nécessaires lors d'une évolution de classe ou d'instance [Bergues94] [Bounaas94a] [Bounaas95a]. Une règle active s'énonce par trois composantes principales : l'événement, la condition et l'action. La détection d'un événement entraîne l'évaluation de la condition ; si celle-ci est satisfaite, l'action est exécutée.

Nous avons identifié deux familles de règles : les règles de vérification, qui contrôlent les contraintes d'intégrité et les règles de propagation qui maintiennent la cohérence des connaissances en propageant les modifications réalisées. Ces règles ne sont pas encapsulées dans les classes d'appartenance des objets dont elles modélisent le comportement actif. Par contre, ces règles peuvent être regroupées dans des bases hiérarchisées. Chaque base hérite et/ou redéfinie les règles définies dans les super-bases ; la redéfinition d'une règle représente un lien d'inhibition.

L'exécution des règles est assurée par un moteur d'exécution offrant la possibilité de les activer et de les désactiver à tout moment, mais n'offrant malheureusement qu'un mode de couplage (immédiat), lacune due à l'absence actuelle des transactions [Boulenger93]

[Roche95]. L'évolution des concepts de règle, base et événement est assurée par les règles actives elles-mêmes. Le modèle actif s'auto-modifie donc lui-même.

Expression des règles d'évolution

L'utilisation des règles actives pour l'évolution a montré la nécessité d'une meilleure structuration de celles-ci. En effet, il faut souvent écrire plusieurs règles ECA pour exprimer une seule règle d'évolution. Pour cela, nous avons proposé un mécanisme permettant la définition et l'exploitation des **Règles d'évolution** [Bounaas95d].

Une règle d'évolution permet d'exprimer des vérifications et des propagations, qui seront exécutées avant et après une opération faisant évoluer un schéma ou une instance. Elle est représentée par trois composantes : la pré-opération, l'opération et la post-opération. La pré-opération et la post-opération représentent des traitements exécutés respectivement avant et après l'exécution de l'opération. Chaque traitement est défini par un couple condition-action. La sémantique d'une règle d'évolution est la suivante : l'exécution d'une opération *o* entraîne l'exécution de sa pré-opération ; si sa condition est vraie, son action et l'opération *o* sont exécutées, sinon *o* est interrompue. L'exécution de *o* est suivie de celle de la post-opération ; si la condition de la post-opération est vraie, son action est exécutée, sinon l'exécution de l'opération et de la pré-opération sont défaites.

Stratégies multiples de l'évolution

Nous avons enfin tenté de résoudre le problème des stratégies en proposant une notion de **Stratégie d'évolution**, regroupant un ensemble de règles d'évolution [Bounaas95d]. Le concepteur a la possibilité de réutiliser une stratégie, en redéfinissant et/ou en héritant ses règles d'évolution. La redéfinition consiste à redéfinir les pré et post opérations de la règle d'évolution ; la validité de cette redéfinition est à la charge du concepteur. De même, nous ne vérifions pas que deux règles incompatibles, ne pouvant pas s'exécuter simultanément, ne sont pas définies dans la même stratégie. La notion de stratégie proposée permet une meilleure réutilisation des règles d'évolution.

La réalisation du mécanisme de règles et de stratégies d'évolution est basée sur la notion de règles ECA. Ce mécanisme est principalement mis en œuvre en établissant une correspondance entre règles d'évolution et règles ECA, ainsi qu'une correspondance entre stratégies d'évolution et bases de règles. En effet, nous associons à une règle d'évolution des règles ECA, et à une stratégie une base de règles, contenant les règles ECA traduisant les règles d'évolution de cette stratégie. La mise en œuvre de ce mécanisme n'a pas nécessité la réalisation d'un moteur d'exécution spécifique, car elle est basée sur l'utilisation du moteur existant pour les règles ECA.

Réalisation et Expérimentation

Dans le cadre de cette thèse, nous avons conçu un système d'évolution permettant l'évolution des concepts du modèle SHOOD, et l'exploitation de plusieurs stratégies d'évolution (Figure 10-1). Ce système est défini par un ensemble de concepts (opération, règle, stratégie), et des mécanismes d'évolution permettant la définition et l'exploitation de ces concepts.
Le système d'évolution de SHOOD est mise en œuvre par un ensemble de mécanismes sous-jacents (classification, règles actives, règles et stratégies d'évolution), qui ont été implanté sur le prototype actuel [Shood93]. Le prototype, développé sur stations de travail Unix, a été transféré à quatre laboratoires universitaires où il est utilisé pour la conception d'applications CAO et multimédia

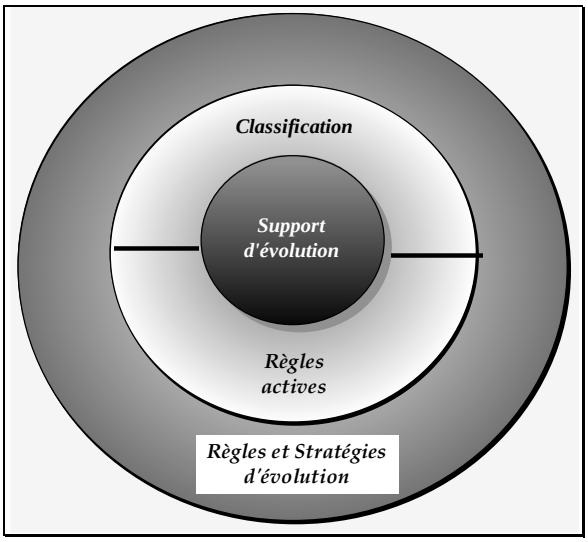


Figure 10-1 : Le Système d'évolution

L'évolution dans le système SHOOD a été testée lors d'une expérimentation avec un partenaire industriel, où l'objectif était de développer une base de connaissances dans le domaine du bâtiment [Culet94] [Bounaas95b]. Le développement de cette base de connaissances a suivi une démarche incrémentale et cyclique, grâce à l'évolutivité du système.

PERSPECTIVES

Nous envisageons des prolongements de ce travail, dans le domaine des règles actives et dans celui de l'évolution, en particulier sur l'étude des stratégies d'évolution, qui sont :

- *enrichir les fonctionnalités du modèle actif actuel*. Pour cela, nous envisageons la modification du moteur d'exécution afin qu'il puisse accepter de nouveaux types de règles (exemple : règles de cohérence permettant de représenter des contraintes faibles). Nous envisageons aussi l'introduction de nouvelles caractéristiques dans le modèle actif, comme la notion de priorité, de séquence d'événements et d'événement temporel. Une autre perspective peut être envisagée ; celle d'enrichir la sémantique de la spécialisation d'une règle en offrant

la possibilité d'hériter ou de redéfinir uniquement la condition ou l'action d'une règle. L'enrichissement du mécanisme de règles actives permettra son utilisation dans d'autres applications, comme la spécification d'un modèle de tâches pour la mise en œuvre des applications de conception [Nguyen93] [Clerc95].

- *étendre le système d'évolution pour prendre en compte les aspects comportementaux*. Il serait intéressant d'étudier la cohérence des méthodes ou des règles lorsqu'un schéma est modifié. Nous pourrions soit détecter les méthodes et les règles à remettre en cause, soit les rendre utilisables lors d'une modification de schémas.

- *adapter les opérations du support d'évolution avec l'introduction de la persistance.* Il faudra remettre en cause certains choix en raison de l'introduction des transactions et pour des problèmes de performances [Roche95]. Par exemple, l'adaptation des instances à une modification de schéma peut être revue ; comme les instances seront sauvegardées en mémoire secondaire, nous pourrions utiliser d'autres méthodes de propagations, comme différer leur modification jusqu'à leur accès effectif.

- *étendre le modèle de stratégies*. La notion de stratégie proposée dans cette thèse permet une meilleure réutilisabilité des mécanismes d'évolution, et offre la possibilité de construire des bibliothèques de stratégies. Pour cette raison, l'extension du modèle de stratégies constituerait une perspective intéressante. Pour faciliter l'utilisation et la réutilisation des stratégies, il faudrait permettre l'activation de plusieurs stratégies en même temps et autoriser la redéfinition multiple des stratégies. Ces deux perspectives nécessitent la résolution des problèmes de conflits de règles. Une autre perspective, à long terme, est de formaliser les règles d'évolution afin de vérifier la cohérence d'une règle et donc d'une stratégie. Pour formaliser une règle d'évolution, nous pourrions nous inspirer des langages de spécification et de vérification de systèmes réactifs [Nguyen 95].

D'autres perspectives peuvent être envisagées pour le système d'évolution comme l'extension du mécanisme de classification pour classifier les objets composites et la conception d'un mécanisme de gestion de versions de classes [Bounaas91]. Ces mécanismes sont nécessaires dans le domaine de la CAO. La classification d'objets composites aura pour tâche la classification des composants d'un objet en cours de classification. Quant au mécanisme de gestion de versions de classes, il aura pour but de garder l'historique des modifications d'un graphe de classes.

11. BIBLIOGRAPHIE

[Ait-Hssain93] A. Ait-Hssain, C. Djeraba, B. Descotes-Genon. *Production Information system design*. In Proc. Industrial engineering and production management. Mons (B) Juin 1993.

[Adele93] Adèle, *Configuration Management.* User Manual, Verilog SA, 1993

[Agrawal90] R. Agrawal. *Object versionning in Ode.* Proc. of the 16th vldb, Brisbane (Australia), Août 1990.

[Ahmed-Nacer94] M. Ahmed-Nacer. *Un modèle de gestion et d'évolution de schéma*. Thèse INPG, Grenoble, Juillet 94

[Anwer93] E.Anwar, L. Maugis, and S. Chakravarthy. *A New Perspective on rule Support for Object Oriented Databases*, ACM SIGMOD Conference on Management of Data, Washington. Mai 93.

[Atkinson89] M. Atkinson, & al. *The object oriented database manifesto*. In Proc. of Int. Conf. On Deductive and O-O Databases DOOD89, Kyoto (Japan), Décembre 1989.

[Bancilhon89] F. Bancilhon, et al. *The O2 Book.* GIP Altaïr, 1989.

[Banerjee87] J.Banerjee, W. Kim, H. Korth. *Semantic and Implementation of schema evolution in Object-Oriented Databases*. Proc. ACM/SIGMOD Conf. On Management of Data, San Francisco, Mai 1987.

[Barbedette91] G. Barbedette. *Schema modifications in the LispO2 persistent object oriented language*. In Proc. of ECOOP91, Genève, 1991.

[Beeri91] C. Beeri, T. Milo. *A Model for Active Object Oriented Database*. Proc. 17th International Conference VLDB, Barcelona, Septembre 1991.

[Benatallah94] B. Benatallah. *Evolution de schéma et systèmes à objets* : une synthèse. Xième Congrès INFORSID 94. Aix en Provence, Mai 1994.

[Benzaken93] V. Benzaken, A. Doucet, P.Y. Policella. *Définition et gestion de contraintes d'intégrité dans le langage Thémis*. In actes 9émes Journées BDA. Toulouse (France), Septembre 1993

[Bergues94] P. Bergues. *Les objets actifs dans un système de représentation de connaissances.* Mémoire CNAM, Versailles, Mars 1994.

[Björnerstedt89] A. Björnerstedt, C. Hulten. *Version control in an object oriented architecture.* Ed. Kim, Lochovsky, Addison Wesley, 1989.

[Bobrow77] D.G. Bobrow, T. Winograd. *An overview of KRL, a Knowledge Representation Language*. Cognitive science, vol1, n. 1, 1977.

[Bobrow88] D.G. Bobrow, L.G. Demichel, & al. *Common Lisp Object System Specification.* X3J13 Document 88-002R, Juin 1988

[Bouaziz95] T. Bouaziz. *Spécification et contrôle d'intégrité dans les bases de données objet : approche utilisant les règles actives*. Thèse INSA de Lyon, Janvier 1995.

[Boulenger93] J. Boulenger. *Introduction à la persistance dans Shood*. Rapport DEA INPG, Septembre 1993.

[Bounaas91] F. Bounaas. *Evolution des types et des classes dans un langage à objets. Application au système Guide*. Rapport de DEA, INPG, Grenoble 1991.

[Bounaas94a] F. Bounaas, P. Bergues. *L'activité : Un outil pour une évolution déclarative*. Xième Congrès INFORSID 94. Aix en Provence, Mai 1994. Sélectionné pour la Revue Ingéniere des Systèmes d'Information, à paraître.

[Bounaas94b] F. Bounaas. *Classification d'objets évolutifs*. IIème Rencontres des Jeunes Chercheurs en Intelligence Artificielle. Marseille, Septembre 1994.

[Bounaas95a] F. Bounaas. *Using rules to support evolution*. Basque International Workshop on Information Technology. BIWIT 95, IEEE Press, Spain, Juillet 1995.

[Bounaas95b] F. Bounaas, A. Culet, J.L. Guffond, G. Leconte. *Le dossier unique actualisé Conception d'un outil d'échange de données et de mise en accord entre métiers du bâtiment*. Edition Communication Construction, Recherches N.67, Avril 1995.

[Bounaas95c] F. Bounaas. *Using ECA rules for object and schema evolution in an object-oriented system.* In Proc. of the 17th TOOLS USA 95, USA, Juillet 1995.

[Bounaas95d] F. Bounaas. *Un outil d'évolution d'objets multi-stratégies.* Actes. AFCET '95, Colloque Technologie Objet, Toulouse, Octobre 1995.

[Bounaas95e] F. Bounaas, G.T. Nguyen. *Evolution in object-oriented systems: an approach using rules.* Poster au congrès OO-ER 95. Australie, Décembre 1995.

[Bouzeghoub91] M. Bouzeghoub, E. Métais. *Semantic modeling of object oriented databases*. In Proc. of the 17th VLDB, Spain, Septembre 1991.

[Bouzeghoub94] M. Bouzeghoub, E. Simon. *Integrity Specification and enforcement in databases*. Revue ISI, Hermès, 2(2) 1994

[Capponi94] C. Capponi. *Exploitation des types dans un modèle de représentation des connaissances par objets*. Dans 9ème Congrès RFIA Paris, 1994.

[Carre89] B.Carre. *Méthodologie orientée objet pour la représentation des connaissances*. Thèse de l'Université des sciences et techniques de Lille Flandres Artois, Janvier 1989.

[Carre90] B. Carré, L. Dekker, J.M. Geib. *Multiple and evolutive representation in the ROME language*. In. Proc. Tools2, Paris, 1990.

[Casais91] E. Casais. *Managing Evolution in Object Oriented Environments : An algorithmic Approch*. PhD Thesis, University of Geneva, 1991.

[Chakravarthy94] S. Chakravarthy, V. Krishnaprasad, E. Anwar S.K. Kim. *Composite events for active databases: semantics, contexts and detection*. Technical report, University of Florida, Gainsville (USA), Février 1994

[Cheval90] J.L. Cheval. *Modélisation de l'évolution des objets*. Rapport Aristote RAP005, Grenoble, Février 1990.

[Clamen92] S.M. Clamen. *Type evolution and instance adaptation*. Technical Report CMU-CS-92-133, Carnegie Mellon University, 1992.

[Clerc95] E. Clerc. *Contribution à l'introduction de règles actives et de tâches dans le SRC SHOOD* Mémoire Ingénieur IIE CNAM. Paris; Juin 1995.

[Cointe87] P. Cointe. *Metaclasses are first classes : The ObjVlisp Model*. Proceedings OOPSLA 1987.

[Collet94] C. Collet, P. Habraken, T. Coupaye, M. Adiba. *Active rules for the software engineering platform GOODSTEP*. In Proc. Of the 2nd International Workshop on Database and Software Engineering. Sorrento, Italy, Mai 1994.

[Culet93] A. Culet, J.F. Millasseau, J.L. Guffond, G. Leconte. *Modélisation et échange de données techniques dans un groupement de PME*. 4e Congrès International de Génie des Systèmes Industriels, GSI4 Marseille, Décembre 1993

[Culet94] A. Culet, F. Bounaas. *Développement d'une base de connaissances pour le bâtiment*. Représentations Par Objets RPO94, Paris, Juin 1994.

[Dayal88] U. Dayal, A. P. Buchmann, D R. McCarthy. *Rules Are Objects Too: A Knowledge Model For An Active OODBMS*. In Proc. Advances in Object-Oriented Database System, Septembre 1988.

[Dayal90] U. Dayal, M. Hsu, P. Gray. *Organizing long running activities with triggers and transactions*. In Proc. Int. Conf. on Management of Data, ACM SIGMOD, Atlantic City,USA, Mai 1990..

[Defude93] B. Defude, H. Martin. *A precondition and postcondition mechanism to enforce integrity in an object oriented database*. In Int. Symposium on Computer and Information Sciences, Istanbul, Novembre 1993.

[Delcourt92] C. Delcourt. Evolution de schémas dans un système de bases de données orienté-objet. Thèse Université d'Orsay. Paris, Novembre 1992.

[Delobel91] C. Delobel, C. Lecluse, P. Richard. *Bases de Données : des systèmes relationnels aux systèmes à objets*. InterEditions 1991

[Diaz91] O. Diaz, N. Paton, P. Gray. *Rule management in object oriented databases: a uniform approach*. In Proc., 17th International Conference VLDB, Barcelona, Septembre 1991.

[Diaz94] O. Diaz, A. Jaime. *Exact : an EXtensible approach to ACTive object-oriented databases*. Workshop on active DBMS Allemagne, Mars 1994.

[Diaz95] O. Diaz. *Dimensions of active databases systems*. Actes Bases de Données Avancées, Nancy, Août 1995.

[Dicky94] H. Dicky, C. Dony, M. Huchard, T. Libourel. *ARES, un algorithme d'ajout avec restructuration dans les hiérarchies de classes*. Actes LMO, Langages et Modèles à Objets, Grenoble, Octobre 1994.

[Dieng90] R. Dieng. *Méthodes et outils d'acquisition des connaissances*. Rapport de Recherche INRIA 1990

[Djeraba93a] C. Djeraba. *Composite objects and dependency relationship in engineering*. In Proc. AIENG, Toulouse 1993.

[Djeraba93b] C. Djeraba. *Quelques liens sémantiques dans un Système à Base de Connaissances*. Thèse Université Claude Bernard, Lyon, Décembre 1993.

[Djeraba93c] C. Djeraba, A. Ait-Hssain, B. Descotes-Genon. *Composition and dependency relationship in production information system design* . In Proc. DEXA, Prague, Septembre 1993.

[Djeraba93d] C. Djeraba, G.T. Nguyen, D. Rieu. *Objets composites et liens de dépendances dans un système de à base de connaissances.* Actes INFORSID, Lille, Mai 1993.

[Djeraba94] C. Djeraba. *Objets composites dans un modèle à objets*. Actes Bases de Données Avancées, Clermont Ferrand, Septembre 1994.

[Ducournau88] R. Ducournau. *YAFOOL : Manuel de référence*. Sema-Matra, 1988.

[Dugerdil88] P. Dugerdil. *Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage ObjLog*. Thèse de doctorat université d'Aix-Marseille, 1988

[Escamilla90] J. Escamilla, P.Jean. *Relationships in an object knowledge representation model*. Proc. 2nd International Conf. Tools for Artificial Intelligence Washington (USA) Novembre 1990.

[Escamilla93] J. Escamilla. *SHOOD un modèle méta-circulaire de représentation de connaissances*. Thèse INPG, Grenoble, Octobre 1993.

[Euzenat87] J. Euzenat. *Maintenance de la vérité dans les représentations centrées-objet*. Actes 6ème RFIA, Antibes (France), 1987.

[Favier90] V. Favier, D. Rieu. *Manipulation d'objets dynamiques dans les bases de connaissances.* In Proc. MICAD '90, Paris, Hermès Ed, Février 1990.

[Gatziu92] S. Gatzui, K.R. Ditttrich. *SAMOS: an active object oriented database system*. IEEE Quartly Bulletin on data engineering, special issue on active database, Vol 15. 1992

[Gehani91] N Gehani and H. V. Jagadish. *Ode as an Active database: Constraints and Triggers.* Proceedings 7th VLDB. September 1991.

[Giambiasi91] N. Giambiasi, C. Oussalah. *Les langages à objets.* Revue Génie logiciel et Systèmes Experts. N. 22, Mars 1991.

[Girard95] P. Girard. *Gestion Hypothétique d'objets complexes*. Thèse UJF, Grenoble, Octobre 1995.

[Goldberg83] A. Goldberg, D. Robson. *Smalltalk-80 The language and its implementation.* Addison-Wesley, 1983.

[Haton91] J.P.Haton, N.Bouzid, F.Charpillet, M.C.Haton & al. *Le raisonnement en intelligence artificielle*, InterEditions, IIA, 1991.

[Katz87] R.H. Katz, E. Chang. *Managing change in a Computer-Aided Design Databases*. In Proc of the 13th VLDB Conference, Brighton 1987.

[Kim88] W. Kim, H. Chou. *Versions of schema for object-oriented databases*. Proc. 14th VLDB Conference, USA 1988.

[Kim89] W. Kim, E. Bertino, J.F. Garza. *Composite objects revisited.* In Proc of ACM SIGMOD 1989.

[Kotz88] R.H. Kotz, K. Dittrich, and J. Mulle. *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*. In Proceedings of the EDBT '88, 1988.

[Krakowiak87] S. Krakowiak & al. *Modèle d'objets et langage du système Guide*, Rapport N. 2, LGI-BULL, Grenoble 1987.

[Lerner90] B.S. Lerner, A.N. Habermann. *Beyond schema evolution to database reorganisation*. In Proc. ECOOP '90, Octobre 1990.

[Libourel92] T. Libourel. *Introduction des relations pour exprimer l'évolutivité dans un système d'objets*. Thèse Université de Montpellier II, Mai 1992.

[Lieberherr89] K.J. Lieberherr, I. Holland. *Formulations ans benefits of the law of Demeter*. SIGPLAN Notices ACM , vol24, N. 3, Mars 1989.

[Liotard93] M.P. Liotard. *Mécanisme de Classification pour un système de représentation de connaissances*. Mémoire d'ingénieur CNAM, Grenoble, Mars 1993.

[MacGregor91] R Macgregor., M.H.Burstein. *Using a Description Classifier to Enhance Knowledge Representation.* IEEE Expert, Juin 1991.

[Marino91] O. Marino. *Classification d'objets composites dans un système de représentation de connaissances multi-points de vue*. 8ème conf. RFIA, Lyon, Novembre 1991.

[Marino93] O. Marino. *Raisonnement classificatoire dans une représentation à objets multi-points de vue*. Thèse UJF, Grenoble, Octobre 1993.

[Masini89] G. Masini, A. Napoli, D. Colnet, D. Leonard, K. Tombre. *Les langages à objets*. InterEditions, IIA, 1989.

[Medeiros90] C.B. Medeiros, P. Pfeffer. *A mechanism for managing rules in an object oriented database*. GIP Altaïr report, Janvier 1990.

[Medeiros91] C.B. Medeiros, P. Pfeffer. *Object integrity Using Rules.* In Proc. ECOOP, Genève, 1991.

[Meyer88] B. Meyer. *Object oriented software construction.* Prentice Hall, New York, 1988.

[Millasseau92] J.F Millasseau. *Les méthodes sont aussi des objets*. Rapport de DEA, INPG, Grenoble 1992.

[Monk93] R. Monk , I. Sommerville. *Schema evolution in OODBs Using Class Versioning*. SIGMOD RECORD Vol 22, No 3, Septembre 1993.

[Napoli92] A.Napoli, *Représentation à objets et raisonnement par classification en intelligence artificielle*. Thèse d'Etat, Nancy, 1992.

[Nguyen89a] G.T. Nguyen, D Rieu. *Schema Evolution in Object Oriented Database Systems*. Data and Knowledge Engineering, North Holland, 1989.

[Nguyen89b] G.T. Nguyen, D. Rieu. *Schema change propagation in object oriented databases*. Proc. 11th World Computer Congress, IFIP Congress 89, San Francisco, Août 1989.

[Nguyen91] G.T Nguyen, D.Rieu. *Databases issues in object oriented design.* In Proc TOOLS 91, Paris, Mars 1991.

[Nguyen92a] G.T Nguyen, D.Rieu. *Multiple Object Representations*. In Proc 20th ACM Computer Science Conf. Kansas City. Mars 1992.

[Nguyen92b] G.T Nguyen, D.Rieu. *SHOOD : a design object model*. Proc. 2nd Intl. Conf. Artificial Intelligence in Design, Carnegie Mellon Univ, Pittsburgh, Juin 1992.

[Nguyen93] G.T. Nguyen. *Shood : plate-forme pour la conception assistée*. Revue Ingénierie des Systèmes d'Information, Hermès, Vol. 1, N.3, 1993.

[Nguyen94] G.T. Nguyen, F. Vernadat. *Cooperative information systems in integrated manufacturing environments.* In Proc. 2nd International Conf. On Cooperative Information Systems, CoopIS 94, Toronto, Mai 1994.

[Nguyen95] G.T. Nguyen. *A reactive object model for concurrent engineering platforms*. Soumis à publication 1995.

[Pachet92] F. Pachet. *Représentation de connaissances par objets et règles : le système NéOpus.* Thèse, Université de Paris 6, Septembre 1992.

[Penney87] D.J. Penney, J. Stein. *Class Modification in the GemStone OODBMS.* In Proc. OOPSLA, Orlando, Florida, October 1987, pp11-117.

[Poncelet94] P. Poncelet, M. Teisseire, R. Cicchetti. *Towards event-driven modelling for database design*. Dans 20th VLDB Santiago Chili, Septembre 1994.

[Ra94] Y.G. Ra, E.A. Rundensteiner. *A transparent object oriented schema change using view evolution*. Technical Report CSE-TR-211-94, University of Michigan, Avril 1994.

[Rechenmann88] F. Rechenmann. *Shirka : un système de gestion de base de connaissances centrées objet*. Rapport technique INRIA, 1988.

[Riet89] R.P. van de Riet. *Mokum: an object oriented active knowledge base system*. Data and Knowledge Engineering. North Holland, 1989.

[Roche95] P. Roche. *La persistance dans Shood*. Rapport Interne. Grenoble, Avril 1995

[Roddick92] J.F. Roddick. *A query language extension for databases supporting schema evolution*. ACM SIGMOD Record 21(3). 1992

[Roncancio94] C. Roncancio. *Règles actives et règles déductives dans les bases de données à objets*. Thèse Université Joseph fourier, Grenoble, Décembre 1994.

[Schek90] H. Schek M.H. Scholl. *Evolution of data models*. Lectures notes in computer science. Database systems of 90's. International Symposium, Berlin, Novembre 1990.

[Scherrer93] S. Scherrer, A. Geppert, K.R Dittrich. *Schema evolution in NO2*. Technical Report Nr.93.12. University of Zurich, Avril 1993.

[Schreiber91] G. Schreiber and al. *The KADS framework for modelling expertise*. In proceedings of EKAW'91, 1991

[Shood93] F. Bounaas, C. Djeraba, J. Escamilla, J.F. Millasseau. *Interface Programmeur de Shood*. LGI-IMAG, Grenoble, Mars 1993.

[Simon92] E. Simon, J. Kiernan, and, C. De Mainderville. *Implementing high level active rules on top of a relational DBMS*. In Proceedings 18th VLDB, Vancouver, British Columbia, Août 1992.

[Simon95] E. Simon, J. Kiernan. *The A-RDL system*. In Actives Databases Systems : Triggers and Rules for Advanced Database Processing, Morgan-Kaufmann publisher, San Francisco à paraître 1995.

[Skarra86] A.H. Skarra, S.B. Zdonik. *The Management of Changing Types in OODB*. In Proc of the OOPSLA '86 Conference, Septembre 1986.

[Skarra87] A.H. Skarra, S.B. Zdonik. *Type evolution in an OOD*B. Research Directions in Object Oriented Programming, Eds B. Shriver and P. Wegner, MIT Press, 1987

[Stefik86] M. Stefik, D.G. Brobrow. *Object Oriented Programming : Themes and variantions*. The AI magazine, 1986.

[Tchounikine93] A. Tchounikine. *Activité dans les bases de données objets : le concept de schéma actif*. Université Paul Sabatier, Toulouse 1993.

[Tresh91] M. Tresh. *A Framework for schema evolution by meta object manipulation*. In 3rd Int. Workshop on Fondations of models and languages for data and objects. Austria, Septembre 1991.

[Waller91] E. Waller. *Schema updates and consistency*. In Proc. DOOD 91. Munich, Allemagne, 1991

[Zdonik86] S.B. Zdonik. *Version Management in an Object Oriented Database*. Proc IFIP 2.4. Workshop on Advanced Programming Environments, Norvège, 1986.

[Zhou94] L. Zhou, E.A. Rundensteiner, K.G Shin. *Schema evolution for real-time object oriented databases*. Technical Report CSE-TR-199-94, University of Michigan, Mars 1994.

[Zicari91] R. Zicari. *A framework for schema updates in an object oriented database system*. In Proc. of the 7th Conference on Data Engineering, Kobe, Japan 1991.