Les versions dans les bases de données orientées objet : modélisation et manipulation

Gilles Hubert

N° d'ordre : 2563

INSTITUT DE RECHERCHE EN INFORMATIQUE DE TOULOUSE
Unité Mixte de Recherche 5505 - Centre National de la Recherche Scientifique -
Institut National Polytechnique de Toulouse - Université Paul Sabatier

THESE

présentée devant
l'UNIVERSITE PAUL SABATIER DE TOULOUSE (SCIENCES)

en vue de l'obtention du

DOCTORAT DE L'UNIVERSITE PAUL SABATIER spécialité **INFORMATIQUE**

par

Gilles HUBERT

Les versions dans les bases de données orientées objet : modélisation et manipulation

Soutenue le 9 Janvier 1997 devant le jury composé de :

M. Eric ANDONOFF	*Maître de Conférence à l'Université Toulouse I*
M. Claude CHRISMENT	*Professeur à l'Université Toulouse III*
M. André FLORY	*Professeur à l'INSA de Lyon*
Mme Geneviève JOMIER	*Professeur à l'Université Paris Dauphine*
M. Jacques LUGUET	*Professeur à l'Université Toulouse III*
M. Gia Toan NGUYEN	*Directeur de recherches à l'INRIA Rhône-Alpes*
M. Gilles ZURFLUH	*Professeur à l'Université Toulouse I (Directeur de Thèse)*

Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse Cedex. Tél : 05.61.55.66.11

A tous ceux que j'aime,
A ceux qui m'ont aidé à rendre ceci possible,
...

Je tiens à remercier très sincèrement :

Monsieur Eric Andonoff, Maître de Conférence à l'Université Toulouse I, pour sa collaboration de tous les instants, ainsi que pour son aide constante et précieuse ; les nombreuses discussions animées que nous avons eues, ont contribué à l'aboutissement de cette étude.

Monsieur Claude Chrisment, Professeur à l'Université Paul Sabatier de Toulouse, pour ses conseils et les critiques constructives qui m'ont permis de progresser.

Monsieur André Flory, Professeur à l'INSA de Lyon, pour avoir accepté de rapporter sur ce travail et pour l'intérêt qu'il y a porté.

Madame Geneviève Jomier, Professeur à l'Université de Paris Dauphine, pour l'honneur qu'elle m'a fait en acceptant d'examiner mon travail, à la lumière de son expérience dans le domaine de la gestion de versions ; ses remarques et discussions m'ont été très utiles.

Monsieur Jacques Luguet, Professeur à l'Université Paul Sabatier de Toulouse, pour l'attention qu'il a portée à la lecture de mon travail et qui m'a permis d'améliorer ce manuscrit.

Monsieur Gia Toan Nguyen, Directeur de recherches à l'INRIA Rhône-Alpes, pour avoir accepté d'être rapporteur ; ses remarques et les améliorations qu'il m'a suggérées, m'ont permis de faire progresser mon travail.

Monsieur Gilles Zurfluh, Professeur à l'Université Toulouse I, pour avoir accepté d'être mon directeur de thèse. Je le remercie également pour les nombreux conseils qu'il a dispensés au cours de ces années, ainsi que pour toutes les remarques qu'il m'a faites et qui m'ont permis d'améliorer cette étude.

Je tiens à leur exprimer toute ma gratitude pour l'honneur qu'ils me font en participant à ce jury de thèse.

Je tiens également à exprimer toute ma reconnaissance à Messieurs Chrisment, Luguet et Zurfluh, pour m'avoir accueilli au sein de leur équipe. Qu'ils sachent que j'ai beaucoup appris à leur contact.

Je tiens également à remercier :

Mon père qui m'a donné le goût des études et de la réussite, et ma mère celui du travail bien fait, et pour s'être souvent inquiétés pour moi.

Nadine, qui me supporte depuis déjà tant d'années, et ce n'est sans doute pas tous les jours facile.

Ma soeur pour s'être régulièrement souciée de l'avancée de ma thèse.

Mes amis du trio infernal, Jos et Francky, pour avoir ouvert la voie et m'avoir montré que je n'étais pas le plus rapide (chose que je savais déjà !!!). Que Francky sache que je ne lui en veux pas de me battre systématiquement au squash. Je les remercient également pour l'aide qu'il m'ont apportée à certains moments de ce travail.

Mes amis du monde de la chimie et associés, Steff, Valérie, (l'autre) Valérie, Eric, Ghassoub, pour leur soutien et leur bonne humeur dans les nombreuses soirées et sorties que nous avons faites ensemble ... et qui dissipent le stress du thésard (et certains savent de quoi je parle !!!).

Ceux qui de loin m'ont motivé, ColoradoSteff pendant son séjour aux Etats-Unis, Florence lorsqu'elle était au Canada, Antonella et Walter depuis leur Italie.

Les "versionneurs", Eric, Annig, Olivier et Frédéric pour leur collaboration et les discussions souvent interminables que nous avons eues.

Les membres de l'équipe SIG, passés et présents, pour leur gentillesse, leur bonne humeur, leur soutien, leurs conseils et j'en passe.

Mes copains de l'IRIT, qui m'ont soutenu jour après jour. Je remercie tout particulièrement Ahmed pour tous les trucs systèmes qu'il m'a appris pendant ces années, Florence et Christine pour m'avoir régulièrement encouragé.

Les membres du CERISS et de l'UT1 qui m'ont accueilli ces deux dernières années.

Tous ceux que je n'ai pas cité, qu'ils m'en excusent, et qui à leur manière, ont contribué à rendre tout cela possible.

Sommaire

Liste des Figures

CHAPITRE I. : Etat de l'art

CHAPITRE II. : Un modèle conceptuel intégrant les versions

CHAPITRE III. : Le langage de description et de manipulation VOSQL

CHAPITRE IV. : Réalisation

Introduction

Le cadre de l'étude

Le concept de version est largement utilisé. Les versions permettent notamment de conserver et manipuler l'évolution des entités du monde réel au cours du temps (pour décrire la réalité ou représenter les étapes d'un processus de conception) ; elles permettent également d'avoir plusieurs représentations simultanées d'une entité.

Le concept de version est nécessaire dans de nombreux domaines d'applications comme la gestion de documentations techniques et la conception assistée par ordinateur. Ces domaines ont en général des besoins spécifiques en matière de gestion de version. Ils nécessitent des moyens pour décrire avec précision des bases de données avec des versions. Les versions ne doivent pas être prises en compte seulement lors de la phase d'implantation. De plus, des outils de manipulation, principalement d'interrogation, puissants et adaptés aux versions s'avèrent nécessaires. Toute la sémantique propre aux versions doit pouvoir être exploitée lors de l'interrogation.

Cette thèse propose des solutions pour décrire et manipuler des bases de données intégrant des versions.

L'existant

Des systèmes de gestion de bases de données proposent des fonctionnalités pour gérer les versions. Ils permettent ainsi de prendre en compte les versions lors de l'implantation d'une base.

Certains systèmes, proposent de gérer des versions d'une base (Gançarski, 94a) ou d'une partie de la base (configuration ou contexte) comme le système O2 (O2, 96). Ces systèmes proposent un principe général pour gérer les versions. Ils sont destinés à décrire l'évolution globale d'une base de données.

D'autres systèmes proposent de gérer des versions d'objets comme les systèmes Version Server (Katz, 86) et OVM (Käfer, 92), ou de gérer des versions de classes comme les systèmes Encore (Skarra, 86) et CloSQL (Monk, 92). Rares sont les systèmes, comme ORION (Kim 89b) et Presage (Talens, 94), qui proposent les deux gestions de versions simultanées. Ces systèmes sont en général dédiés à un domaine d'application comme par exemple la représentation de circuit intégré dans le système OVM. C'est dans cette approche que nous nous situons notamment pour répondre aux besoins en matières de gestion de documentation technique. Ce type de système est destiné à décrire et manipuler l'évolution de chaque entité représentée dans la base de données.

Par ailleurs, peu de systèmes offrent de véritables outils de manipulation des versions surtout en ce qui concerne leur interrogation. Seules les bases de données

temporelles, avec des langages tels que TSQL (Navathe, 89) et TSQL2 (Snodgrass, 94), offrent des possibilités d'interrogation spécifiques par rapport au temps. Pour les versions, certains systèmes, comme Presage, ne proposent pas de langage d'interrogation ; les autres systèmes, comme Aristote (Fauvet, 92) et ORION, ne proposent qu'un langage de requête considérant les versions comme de simples objets c'est-à-dire sans prendre en compte leurs spécificités.

Nos propositions

Dans des domaines comme la gestion de documentation technique, il subsiste des besoins en moyens pour décrire simplement et précisément des bases de données intégrant des versions et pour les manipuler.

Pour répondre à cela, nous proposons un modèle conceptuel intégrant les versions d'objets et de classes. Ce modèle étend celui de la méthode OMT (Rumbaugh, 91). Cette extension peut être appliquée aux modèles des autres méthodes de conception orientée objet. La définition de relations de composition et d'association permet de décrire précisément des entités complexes, telles que des documents, pour lesquelles on gère des versions. La sémantique des relations est affinée par des cardinalités associées aux classes liées. Ces relations impliquent des contraintes d'intégrité sur les instances liées (objets et versions d'objets).

Par ailleurs, nous proposons un langage pour la manipulation d'une base de données définie suivant notre modèle. Le langage permet principalement une interrogation structurée de type Select From Where. Il prend en compte de la sémantique propre aux versions. Il permet une interrogation uniforme d'objets et de versions d'objets.

De plus, le modèle et le langage ont été expérimentés au travers d'un prototype. Il s'agit d'une interface graphique pour la manipulation de bases de données avec versions. Cette interface est destinée à des utilisateurs occasionnels. Elle constitue l'une des finalités des travaux sur le modèle et le langage.

L'organisation du mémoire

Cette thèse se décompose en quatre parties.

Le premier chapitre précise le cadre de cette thèse et fait un état de l'art des différents travaux réalisés pour décrire l'évolution dans les bases de données. Il présente les apports des bases de données temporelles notamment pour l'interrogation par rapport au temps. Il présente également les principales solutions pour l'évolution de schéma. Ce chapitre insiste sur les travaux réalisés dans le domaine de la gestion de versions. Il présente les apports des systèmes gérant des versions uniquement au niveau des instances ou du schéma. Il présente plus

particulièrement les systèmes auxquels nous nous intéressons, c'est-à-dire ceux permettant la gestion simultanée des deux types de versions. Ce chapitre positionne notre approche pour décrire des bases de données orientées objet intégrant des versions au travers d'un modèle conceptuel incluant les versions d'objets et de classes ; il positionne également, notre proposition de langage structuré pour la manipulation de telles bases de données.

Les chapitres suivants présentent notre contribution et une mise en oeuvre au travers d'un prototype.

Le chapitre II est consacré au modèle conceptuel intégrant les versions que nous proposons. L'intégration est effectuée dans les modèles statique, dynamique et fonctionnel de la méthode OMT. La modélisation de versions est réalisée au niveau des instances et du schéma. Le modèle de données permet de représenter les entités avec ou sans versions. Il offre une grande puissance dans la représentation d'entités complexes qui peuvent être décrites totalement ou partiellement à l'aide de versions. Des relations de composition et d'association peuvent être définies entre différents types d'instances. Les contraintes inhérentes à la définition de ces relations sont clairement définies ainsi que leur respect par les opérations sur les instances et les classes.

Le chapitre III présente le langage de description et de manipulation associé à notre modèle. Le langage permet la spécification de schéma (création des classes et des relations entre classes) et son évolution (modification de classe, création de versions de classes, ...). Il permet également la définition d'instances (création, définition de version), la modification, la suppression d'instance. Il offre également une interrogation structurée de type Select From Where. Contrairement aux autres, il prend en compte la sémantique propre aux versions. De plus, il permet une interrogation uniforme des objets non versionnalisés et des versions d'objets. Il intègre également les spécificités liées à la gestion de versions de classes.

Le dernier chapitre est consacré aux solutions pour la mise à oeuvre du modèle, des contraintes et du langage ; ce chapitre décrit leur expérimentation au sein de l'interface graphique pour bases de données orientées objet intégrant des versions VOHQL(Version and Object Hypertext Query Language). Cette interface est fondée sur notre modèle et son langage et est destinée à des utilisateurs occasionnels. Le chapitre présente également le prototype d'interface VOHQL développé à l'aide du système de gestion de bases de données orientées objet (SGBDO) O2.

CHAPITRE I.

Etat de l'art

Chapitre I : Etat de l'art

Table des matières

1. L'évolution dans les bases de données orientées objet

Certains domaines comme la conception assistée par ordinateur (CAO), la gestion de documentations techniques, le génie logiciel, où les entités du monde réel décrites sont en continuelle évolution au cours du temps, ont besoin de prendre en compte cette évolution. Pour cela, il est important de décrire l'évolution des entités, de la conserver, de la consulter, de la manipuler.

Dans les bases de données, une entité du monde réel est représentée par un n-uplet (modèle relationnel) ou un objet (modèle objet) suivant un schéma donné (ensemble d'attributs pour le modèle relationnel, attributs et méthodes pour le modèle objet). L'évolution d'une entité du monde réel peut ainsi être traduite au niveau de la base de données au niveau des instances (objet ou n-uplet) et au niveau du schéma.

La prise en compte de l'évolution d'une entité au niveau des instances concerne la modification de la valeur représentant l'entité. En revanche, le niveau schéma traduit une évolution de la description de l'entité.

Les études réalisées au niveau des instances se scindent en deux approches:

- l'approche des bases de données temporelles ; de nombreux travaux ont été réalisés (Kline, 93), pour le modèle relationnel (Adiba, 87), (Snodgrass, 87), (Gadia, 92) et pour des modèles objet (Rose, 91), (Pissinou, 92),
- l'approche gestion de versions ; la gestion de versions est réalisée au niveau des attributs (Sciore, 91), au niveau des objets pour les bases documentaires (Ben Amouzegh, 86), et pour les bases de données orientées objets avec les systèmes tels que Version Server (Katz, 86)(Katz, 90b), ORION (Kim, 87)(Chou, 88), SHOOD (Rieu, 86)(Nguyen, 89), Ode (Agrawal, 91) et OVM (Käfer, 92), ou au niveau de contextes de versions (Cellary, 90), (Kimball, 91).

Les travaux gérant l'évolution au niveau schéma proposent trois grandes approches :

- l'approche modification de schéma, comme dans les systèmes Gemstone (Penney, 87), OTGen (Lerner, 90)(Lerner, 94) et NO2 (Scherrer, 93),
- l'approche gestion de versions pour des systèmes tels que Encore (Skarra, 86), CloSQL (Monk, 93) et les travaux de Clamen (Clamen, 94),
- l'approche gestion de vues (Bertino, 92), (Souza, 93).

Certains systèmes ont également pris en compte l'évolution aux deux niveaux. C'est le cas notamment des systèmes ORION (Chou, 88)(Kim, 88)(Kim, 89b), IRIS (Beech, 88), AVANCE (Björnerstedt, 89), SHOOD (Nguyen, 89) (Escamilla, 90) et Presage (Talens, 93).

2. L'évolution d'instance

Habituellement, dans les bases de données, l'évolution d'une entité du monde réel se traduit au niveau de l'instance qui la représente par une modification de la valeur (cf. (Date, 82), concept de base de données opérationnelle). L'entité reste décrite par le même schéma, seule la valeur de l'instance correspondante change.

Différentes solutions ont été proposées pour représenter l'évolution de l'entité. Une première solution est celle des bases de données temporelles (ou historiques). Les différentes valeurs prises par un attribut sont conservées, soit systématiquement pour les bases de données historiques, soit selon une fréquence temporelle pré-définie ou définie par l'utilisateur.

Une autre solution est l'approche gestion de versions. Pour chaque entité du monde réel représentée, on conserve différentes valeurs prises par l'instance qui la décrit. Ces valeurs sont conservées sur décision de l'utilisateur ou en fonction de règles qu'il définit.

2.1. Les bases de données temporelles

Une première approche pour gérer l'évolution des entités a été d'intégrer le temps au niveau de leur représentation dans la base de données. Les premiers travaux ont porté sur l'extension du modèle relationnel (Snodgrass, 87), (Navathe, 89), (McKenzie, 91). Certains travaux ont ensuite été développés autour de modèles orientés objet (Rose, 91), (Pissinou, 92), (Wuu, 92).

2.1.1. Les modèles relationnels

Parmi les travaux qui étendent le modèle relationnel, deux approches se distinguent :

- l'intégration du temps au niveau des attributs (McKenzie, 91), (Gadia, 92),
- l'intégration du temps au niveau des n-uplets (Ben-Zvi, 82), (Snodgrass, 87), (Navathe, 89).

Le langage d'interrogation est également étendu pour permettre l'interrogation selon des conditions temporelles (Snodgrass, 87), (Navathe, 89).

2.1.1.1. Le temps au niveau des attributs

L'intégration du temps au niveau d'un attribut consiste à conserver les valeurs successives prises par l'attribut d'une entité. Lors de la définition de schéma, une composante de type intervalle de temps est associée à chaque attribut.

Pour chaque n-uplet, chaque valeur prise par un attribut est conservée et associée à la date, ou la période à laquelle elle correspond. Une entité est donc décrite, pour chaque attribut, par un ensemble de couples (valeur, date). Les relations dans ce cas ne sont plus en première forme normale mais s'apparentent au modèle NF^2 (Sheck, 82).

Exemple I-1 : Dans le modèle relationnel intégrant des aspects temporels de Chakravarthy & Kim (Chakravarthy, 93), certains attributs sont couplés à un attribut de type intervalle de temps. Les instances de la relation PERSONNE sont représentées de la manière suivante :

Matricule	Nom	Valid time	Adresse	Valid time	Emploi	Valid time
1000	Durand	(1/80 ~ ∞)	Paris Lyon	(1/80 ~ ∞) (7/82 ~ ∞)	étudiante comptable	(1/80 ~ 6/82) (7/82 ~ ∞)
1001	Lassalle	(5/85 ~ ∞)	Lyon	(5/85 ~ ∞)	stagiaire vendeur	(5/85 ~ 12/88) (1/89 ~ ∞)

La période associée à la valeur d'un attribut est en général la période de validité (appelée également période de l'événement ou période logique) de la valeur associée. La période de validité d'une valeur est l'intervalle pendant lequel le fait décrit est vrai dans le monde réel modélisé (Jensen, 94).

Certains travaux introduisent la notion de date de transaction (appelée également date physique) à la place de la date de validité ou en plus de celle-ci (Gadia, 92). La date de transaction est la date à laquelle une valeur est enregistrée dans la base de donnée.

Pour manipuler le temps, différents types sont définis :

- les points de temps : instants isolés dans le temps (ex. les dates d'obtention de diplôme de Durand),
- les périodes : intervalles de temps (ex. la période d'emploi de secrétaire de Durand),
- les durées : quantités de temps (ex. la durée d'emploi de secrétaire de Durand),
- les valeurs temporelles complexes : union d'instants isolés dans le temps et/ou d'intervalles de temps (ex. l'ensemble des périodes de chômage de Durand).

Des opérations ont été introduites pour manipuler ces types temporels. Différents opérateurs de comparaisons ont été définis (précède, pendant, succède, ...) pour comparer des points de temps, des périodes, des points de temps et des périodes, ... ;

les opérations arithmétiques ont également été spécifiées pour additionner, ..., supprimer des valeurs des différents types temporels (ex. 20 mars 1995 + 3 jours = 23 mars 1995) (Allen, 84), (Snodgrass, 87), (Navathe, 89).

2.1.1.2. Le temps au niveau des n-uplets

Les principes définis pour l'intégration du temps au niveau des attributs ont également été appliqués au niveau des n-uplets. Chaque valeur prise par un n-uplet est associée à une date ou une période. Un n-uplet représente donc une même entité du monde réel durant une période donnée du monde réel modélisé. L'intégration du temps dans une relation est spécifiée lors de la définition du schéma.

Exemple I-2 : Dans le modèle intégrant le temps au niveau des n-uplets de Snodgrass (Snodgrass, 87), la définition du schéma d'une relation PERSONNE prenant en compte le temps est la suivante :

create interval PERSONNE (Matricule is integer, Nom is char, Adresse is char, Emploi is char)

Le mot-clé interval indique que la relation intègre des intervalles de validité.

La relation PERSONNE de l'exemple précédent se représente alors comme suit :

Matricule	Nom	Adresse	Emploi	*From*	*To*
1000	Durand	Paris	étudiante	1-1-80 10:00AM	7-31-82 5:00PM
1000	Durand	Lyon	comptable	7-31-82 5:00PM	forever
1001	Lassalle	Lyon	stagiaire	5-1-85 8:00AM	12-31-88 12:00PM
1001	Lassalle	Lyon	vendeur	12-31-88 12:00PM	forever

L'intégration du temps au niveau des n-uplets permet d'obtenir ces relations en première forme normale. Cependant, la création d'un nouvel n-uplet pour une modification de la valeur d'un attribut pose un problème de redondance d'information pour les autres attributs. Pour pallier cet inconvénient, la notion de dépendance temporelle a été introduite (Navathe, 89). Les attributs ou groupe d'attributs dépendants du temps de manière indépendante ont une dépendance temporelle. Le processus de décomposition de relations est appliqué suivant les dépendances temporelles. Les relations obtenues après décomposition sont alors en forme normale temporelle.

Exemple I-3 : dans l'exemple précédent, l'attribut Nom, l'attribut Adresse et l'attribut Emploi ont chacun une dépendance temporelle propre ; ces attributs ont des valeurs qui évoluent au cours du temps, indépendamment les uns des autres. Il existe donc les trois dépendances temporelles suivantes :

$T \rightarrow \{Nom\}$ $T \rightarrow \{Adresse\}$ $T \rightarrow \{Emploi\}$

T représente le temps

La relation R (Matricule, Nom, Adresse, Emploi, *From*, *To*) est décomposée suivant les dépendances temporelles. L'attribut Matricule est clé primaire, c'est la clé indépendante du temps. Les trois relations R1, R2 et R3 suivantes sont obtenues:

R1 (Matricule, Nom, *From*, *To*)

R2 (Matricule, Adresse, *From*, *To*)

R3 (Matricule, Emploi, *From*, *To*)

R1, R2 et R3 sont en forme normale temporelle, et contiennent les n-uplets suivants :

R1

Matricule	Nom	*From*	*To*
1000	Durand	1-1-80 10:00AM	forever
1001	Lassalle	5-1-85 8:00AM	forever

R2

Matricule	Adresse	*From*	*To*
1000	Paris	1-1-80 10:00AM	7-31-82 5:00PM
1000	Lyon	7-31-82 5:00PM	forever
1001	Lyon	5-1-85 8:00AM	forever

R3

Matricule	Emploi	*From*	*To*
1000	étudiante	1-1-80 10:00AM	7-31-82 5:00PM
1000	comptable	7-31-82 5:00PM	forever
1001	stagiaire	5-1-85 8:00AM	12-31-88 12:00PM
1001	vendeur	12-31-88 12:00PM	forever

2.1.1.3. Les langages d'interrogation temporels

De nombreux travaux ont traité de l'extension des langages pour prendre en compte les aspects temporels (Tansel, 93). De nombreuses extensions ont été apportées aux langages relationnels notamment au langage SQL (Navathe, 89), (Sarda, 90), (Theodoulidis, 94). Le langage TSQL2 est en cours de normalisation (Snodgrass, 94).

Les langages temporels sont essentiellement basés sur l'ajout d'une clause temporelle (clause **WHEN**) au niveau des requêtes. Des conditions de sélection basées sur l'utilisation d'opérateurs temporels sont spécifiées dans cette clause. Les conditions temporelles portent sur la comparaison de valeurs de types temporelles et plus particulièrement celle de la période de validité des n-uplets. La période de validité des n-uplets est également utilisée au niveau de l'affichage du résultat d'une requête.

Exemple I-4 : considérons l'expression en langage de type SQL temporel de deux requêtes appliquées à deux relations correspondant aux relations R1 et R3 définies dans l'exemple précédent.

Question 1 : "A quel âge Durand était-il stagiaire ?"

```
select  Age
from    R1, R3
where   R1.Matricule = R3.Matricule
  and   R1.Nom = "Durand"
  and   R3.Emploi = "stagiaire"
when    R1.Interval overlap R3.Interval
```

Question 2 : "Donner l'historique des emplois de chaque personne, ainsi que les intervalles de validité associés"

```
select  R1.Nom, Emploi, (R1 inter R3).From, (R1 inter R3).To
from    R1, R3
where   R1.Matricule = R2.Matricule
when R1.Interval overlap R2.Interval
```

La clause **when** permet dans ces deux requêtes de mettre en correspondance dans R1 et R3 les n-uplets qui décrivent une personne pour la même période c'est-à-dire les n-uplets pour lesquels les périodes de validité se chevauchent (**overlap**).

2.1.2. Les modèles objets et entité/association temporels

Avec l'apparition des bases de données orientées objet, des modèles objets ont également été étendus par des notions temporelles (Rose, 91), (Edelweiss, 93), (Pissinou, 92), (Wuu, 92). Les travaux sur les modèles objets, encore peu nombreux, s'inspirent des nombreuses études réalisées pour le modèle relationnel. De plus, en l'absence de norme, les différences entre les modèles objets impliquent des travaux sur l'intégration du temps beaucoup plus hétérogènes que pour l'approche relationnelle. Des travaux récents, sur le système ORES (Theodoulidis, 94), ont également étendu le modèle entité/association à des aspects temporels, en se basant sur les études sur les modèles relationnels temporels (cf. § 2.1.1.).

2.2. Les modèles de gestion de versions

Les modèles de gestion de versions permettent également de conserver l'évolution des entités, mais sans être strictement liés au temps. Différentes valeurs d'une instance sont conservées sur décision de l'utilisateur, lorsque celui-ci les juge importantes. Le temps n'est plus le critère principal qui différencie les versions correspondant à une entité modélisée, ce sont les relations entre versions. Le temps

n'est plus utilisé comme unique moyen d'obtenir et manipuler des versions données d'entités.

La gestion de versions au niveau des instances est divisée en trois grandes approches : la gestion de versions d'attributs (Sciore, 91), la gestion de versions d'objets (Katz, 86), (Rieu, 86), (Chou, 88), (Dittrich, 88), (Agrawal, 91), (Käfer, 92), et la gestion de versions de contextes (Reichenberg, 89), (Cellary, 90), (Kimball, 91), (Gançarski, 94a). Certains travaux intègrent des concepts de plusieurs de ces approches mais aussi des bases temporelles. L'approche gestion de versions est la plus utilisée. Des études ont été menées pour :

- des modèles objet, pour les bases documentaires (Ben Amouzegh, 86) (Chrisment, 91)(Comparot, 94), (Dattolo, 95), et pour les systèmes de gestion de bases de données orientées objet tels que Version Server (Katz, 86) (Katz, 90a), SHOOD (Rieu, 86)(Djeraba, 93), ORION (Kim, 89b), Ode (Agrawal, 91) et OVM (Käfer, 92),
- des modèles de type Entité / Association, avec le modèle O2XER (Dijkstra, 93) et le modèle CERM (Dittrich, 90).

2.2.1. Les modèles de gestion de versions d'attributs

La gestion de versions d'attributs s'inspire des études effectuées dans les bases de données temporelles, notamment pour la gestion d'historique au niveau des attributs (cf. § 2.1.1.1.) et ajoute également de nouvelles fonctionnalités. Plusieurs sortes de versionnalisation peuvent être appliquées à un attribut. Sciore (Sciore, 91) propose un modèle de gestion de versions d'attributs basé sur l'utilisation d'annotations. Les annotations sont notamment utilisées pour associer des contraintes à des attributs ; elles sont alors appelées déclencheurs ou annotations actives de valeur. Sciore étend ce principe à différents types de versionnalisation d'attributs. Lors de la spécification de schéma, le concepteur indique les attributs pour lesquels sont gérés des versions, à l'aide de différents types d'annotations.

Exemple I-5 : Spécification du schéma (Sciore, 91) de la classe Personne pour laquelle une annotation de gestion d'historique est associée à l'attribut Salaire.

```
class Personne =
   variables
        Nom: string;
        Age: integer;
        Salaire: HistoryFn(string);
end Personne;
```

L'annotation **HistoryFn** associée à l'attribut Salaire indique que des versions historiques seront conservées pour cet attribut.

Le modèle propose trois sortes de versionnalisation : les **versions historiques**, les **versions révisées** et les **versions alternatives**. De plus, il est possible de combiner les différents types de versionnalisation sur un même attribut.

Les **versions historiques** consistent à conserver les différentes valeurs prises par un attribut en les associant à leur date de validité. Les **versions révisées** consistent à conserver les différentes valeurs prises par un attribut en les associant à leur date de validité physique, c'est-à-dire la date de modification de la valeur dans la base de données. Enfin, les **versions alternatives** permettent de donner plusieurs valeurs en parallèle à un attribut, en les associant à un nom ; elles permettent notamment de mener simultanément plusieurs solutions de conception dans des domaines comme la conception assistée par ordinateur (CAO).

Les annotations, pour la gestion de versions, sont également prises en compte au niveau du langage d'interrogation. Une requête correspond à une séquence d'expressions appliquées à un objet. Le langage permet d'obtenir, par défaut, les versions courantes sans se préoccuper de la gestion des versions. L'utilisateur peut également accéder aux autres versions en changeant de contexte (temporel ou alternatif) à l'aide d'opérateurs appropriés.

Exemple I-6 : Requêtes sur des objets de la classe Personne, avec des conditions portant sur l'attribut Salaire soumis à la gestion de versions historiques.

Durand.↑Salaire **lifespan**;
donne les différents intervalles durant lesquels Durand a eu un salaire.

Durand.↑Salaire **timeWhen:**[:a | a>5000];
donne les intervalles durant lesquels le salaire satisfaisait la condition.

2.2.2. Les modèles de gestion de versions d'objets

L'approche choisie pour la gestion de versions d'objets est de conserver les instances de l'objet et non des attributs séparément. Chaque état conservé d'une instance est appelé **version** (Katz, 86). La gestion de versions repose essentiellement sur la relation **prédécesseur-successeur**, également appelée relation **est-dérivé-de** (Kim, 89a), (Talens, 93), ou **est-version-de** (Katz, 86), (Djeraba, 93), (Comparot, 94), existant entre les versions d'une instance.

2.2.2.1. Le principe de dérivation

Une entité du monde réel est initialement représentée par une seule version appelée **version racine**. Toute évolution de cette entité, représentée au sein de la base de données, induit la définition d'une nouvelle version. Toute nouvelle version est obtenue à partir d'une version précédente, par dérivation. Cette relation

de dérivation entre versions permet de représenter l'évolution d'une entité. La gestion de versions a été introduite notamment pour répondre aux besoins de représentation des processus de conception.

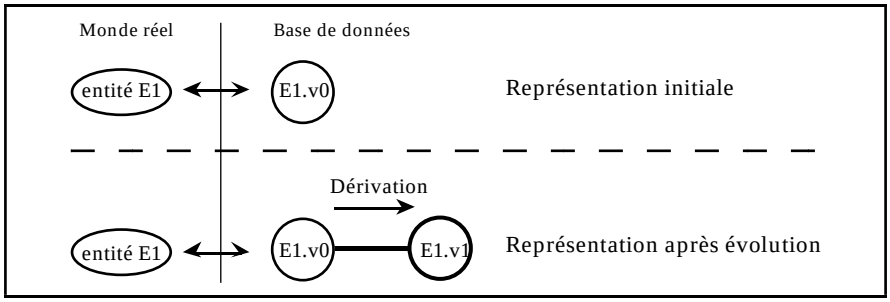


Figure I-1 : dérivation de versions

Dans ce domaine, un besoin supplémentaire est la nécessité de mener plusieurs solutions de conception en parallèle, quitte à en abandonner une au cours du processus. Pour répondre à ce besoin, les systèmes de gestion de versions ont introduit le concept d'**alternatives**. Plusieurs versions peuvent être dérivées à partir d'une même version. Des versions dérivées d'une même version sont des **alternatives** les unes par rapport aux autres ; dans l'exemple qui suit, E1.v2 est une alternative par rapport à E1.v3 et inversement. Dans un système de gestion de versions, une entité est donc représentée par un arbre de dérivation de versions.

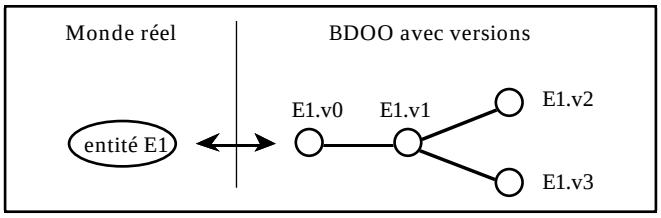


Figure I-2 : arbre dérivation de versions

Les premiers systèmes de gestion de versions ne sont pas basés sur le principe de dérivation. La gestion de versions a été introduite dans le contexte génie logiciel, pour conserver différentes solutions intermédiaires sans pour autant pouvoir utiliser l'ordre dans lequel elles étaient créées.

Le principe de dérivation est apparu pour des domaines où il est important de conserver le processus de conception. Le concepteur peut revenir en arrière dans le processus de conception et explorer une nouvelle solution grâce aux alternatives. De nombreux systèmes ont défini leur modèle de versions sur ce principe : Version Server (Katz, 86), SHOOD (Rieu, 86)(Nguyen, 89), ORION (Kim, 87) (Chou, 88), IRIS (Björnersted, 88) et OVM (Käfer, 92).

2.2.2.2. Les versions d'objets complexes

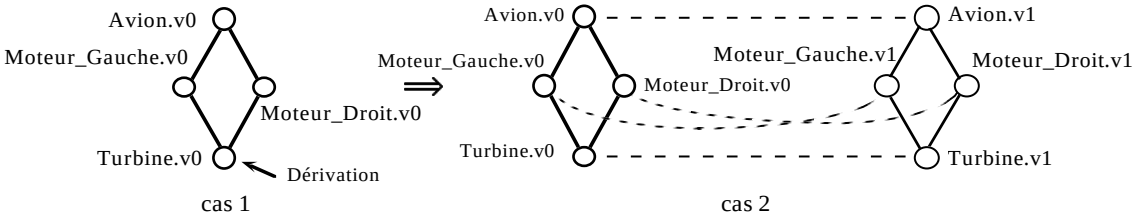
Les objets complexes sont constitués d'objets plus élémentaires. Les versions d'objets complexes suivent le même principe, étant constituées de versions d'objets plus élémentaires. La difficulté réside dans le maintien de la cohérence des versions d'objets complexes vis-à-vis de la définition des versions constituantes. Il s'agit de définir les répercussions de la définition de nouvelles versions constituantes sur les versions d'objets complexes dont elles font partie, ainsi que les répercussions sur les autres constituants.

Les solutions proposées pour la gestion de versions d'objets complexes s'attachent, pour la plupart, à partager des versions de constituants entre plusieurs versions de d'objets complexes (Bensadoun, 89), (Katz, 90a), limitant ainsi la création de versions inutiles. Cependant, cette solution impose de définir des principes d'évolution pour les versions constituantes, pour garantir une évolution cohérente d'un objet complexe ; il s'agit en général de définir des principes de propagation des créations de versions de constituants automatiques ou définis par l'utilisateur.

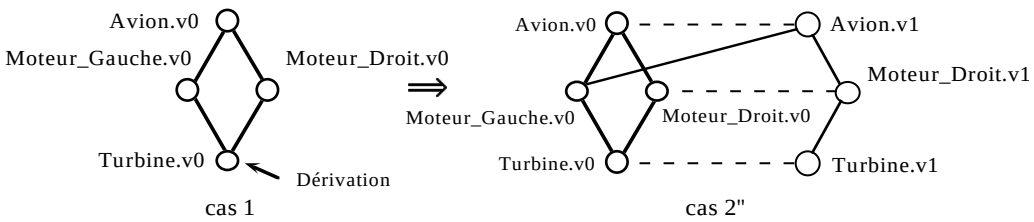
Différentes solutions ont été proposées pour définir des principes de propagation de modifications réalisées sur les composants d'un objet complexe. Afin d'éviter des situations ambiguës introduites par le partage de versions composantes, Katz (Katz, 90b) propose un processus (Check-in / Check-out) permettant de regrouper les constituants dont l'évolution est simultanée. En effet, pour un constituant faisant partie de plusieurs versions, une nouvelle version du constituant peut affecter toutes les versions qu'il compose ou seulement certaines d'entre elles ; l'utilisateur indique les versions qui doivent être affectées simultanément.

Exemple I-7 : Pour une version d'objet complexe cyclique (cas 1), l'évolution de la *Turbine,* constituant du *Moteur Gauche* et du *Moteur Droit* peut impliquer :

- l'évolution des deux moteurs (cas 2) ; c'est le cas obtenu par défaut,



- uniquement l'évolution du *Moteur Droit* (nouvelle version) ; le *Moteur Gauche* reste constitué de l'ancienne version de la Turbine (cas 2"),



- uniquement l'évolution du *Moteur Gauche* (nouvelle version) ; le *Moteur Droit* reste constitué de l'ancienne version de la Turbine.

Katz (Katz, 90a)(Katz, 90b) définit un mécanisme permettant à l'utilisateur de spécifier quel cas il désire obtenir (ex. Check-in(Moteur_Droit,Turbine) permet d'obtenir le cas 2).

Le système Presage (Talens, 93) définit quant à lui la notion d'attributs sensitifs et de versions sensitives (cf. § 4.2.2.2.). Lorsqu'un attribut matérialisant une composition entre versions est désigné sensitif pour une classe, il propage les créations de versions composantes ainsi que de versions composées, et ce, pour toutes les instances de la classe. Un attribut non sensitif conduit au partage de versions composantes. La notion de version sensitive est similaire ; tous les attributs matérialisant des compositions sont sensitifs mais uniquement pour la version désignée.

Par ailleurs, différents travaux (Dittrich, 88), (Katz, 90a), (Käfer, 92) ont introduit la notion de **configuration** issue du génie logiciel. Une **configuration** regroupe un ensemble de versions d'objets de la base de données correspondant à une signification donnée (temporelle, ...). La notion de configuration permet de définir un espace "cohérent" (par rapport à la sémantique donnée à cet espace) dans l'espace des versions. Une seule version d'un objet appartient à une configuration. Une version d'un objet peut appartenir à plusieurs configurations.

Dittrich utilise la notion de **configuration** (ou **environnement**) notamment pour les versions d'objets complexes. Une version peut faire référence à une autre version au travers d'une configuration (ex. la version v1 de l'avion Airbus A320 fait référence à la version de moteur Rolls Royce de la configuration Air France) ; la version référencée peut changer sans affecter la version référençante.

La notion de configuration permet notamment d'obtenir différentes relations entre des versions et une même version sans avoir à dupliquer cette dernière, comme dans le système OVM (cf. § 2.3.2. Figure I-3). Dans ce système, la notion de configuration est utilisée pour respecter les cardinalités fixées pour les compositions ; celles-ci doivent être respectées au sein d'une configuration (cf. § 2.3.3.2.).

Dans le système Version Server (Katz, 90a), la définition de configuration permet de réduire encore la propagation des création de versions composantes en définissant des contraintes de propagation suivant les configurations.

2.2.3. Exemple de système gérant des versions d'objets : OVM

Le système présenté dans cette section ne s'intéresse qu'à la gestion de versions au niveau objet. Il s'agit d'un système de gestion de versions d'objets proprement dit, c'est-à-dire qu'il permet de conserver et manipuler le processus d'évolution d'une entité. Il est basé sur des principes présents dans la plupart des systèmes de gestion de versions d'objets.

Le système OVM (Käfer, 92) a été développé pour répondre aux besoins de conception assistée par ordinateur. Il est construit au-dessus d'un SGBD pour objets complexes, basé sur un modèle d'objets complexes. Il permet la gestion de versions d'objets. Les objets peuvent être versionnalisés ou non versionnalisés ; les objets non versionnalisés sont définis comme des objets ayant une seule version.

2.2.3.1. Le modèle des objets et des versions du système OVM

Le modèle d'objets et de versions d'OVM permet la spécification de types d'objets complexes, construits à partir de la définition de types d'objets élémentaires. Les objets complexes sont définis suivant deux parties distinctes : une partie indépendante de la gestion de versions et une partie spécifique aux versions. Les objets complexes sont définis par :

- le nom des différents constituants (objets élémentaires) ; le concepteur indique si les constituants sont pris en compte par la gestion de versions ou non,
- des attributs "indépendants" de la gestion de versions,
- les attributs "spécifiques" aux versions d'objets complexes.

Les constituants et les attributs désignés indépendants des versions ont des valeurs identiques pour toutes les versions des objets complexes. En revanche, les constituants et les attributs spécifiques aux versions peuvent avoir des valeurs différentes pour chaque version de l'objet complexe.

Les versions (exceptées les racines) sont normalement dérivées de versions précédentes. Une version peut également être dérivée de plusieurs versions (fusion). La relation de dérivation de versions forme ainsi un graphe de versions. La relation de dérivation peut être limitée à la formation d'une liste de versions, d'un arbre ou un graphe orienté acyclique.

De plus, le système décrit des relations entre objets et versions d'objets. Les relations sont définies par des types de liens, séparément de la définition des objets. OVM distingue également des liens entre objets complexes et des liens entre types élémentaires.

Exemple I-9 :

```
---- Définition des types d'objets complexes----
DEFINE OBJECT_TYPE Avion AS
    ---- constituants élémentaires ----
    Moteur, Pièces VERSIONED
    ---- attributs indépendants des versions ----
    (OBJECT_ATTRIBUTES:   av_id:      IDENTIFIER,
                          nom:        LIST_OF(CHAR),
                          catégorie:  LIST_OF(CHAR),
    ---- attributs spécifiques aux versions ----
    VERSION_ATTRIBUTES:   nb_panne:   INTEGER,
                          h_de_vol:   REAL)
  ---- nature du graphe de dérivation ----
  VERSION DERIVATION GRAPH IS LIST;

DEFINE OBJECT_TYPE Documentation AS
    Chapitre NOT VERSIONED
    (OBJECT_ATTRIBUTES:   doc_id:     IDENTIFIER,
                          nom:        LIST_OF(CHAR),
                          niveau:     LIST_OF(CHAR));

---- Définition des types de configuration ----
DEFINE CONFIGURATION_TYPE Av_Doc
    FOR Avion, Documentation
    (ATTRIBUTES:          conf_id:    IDENTIFIER,
                          nom:        LIST_OF(CHAR));

---- Définition des liens entre types d'objets complexes ----
DEFINE LINK_TYPE          Avion.has_doc(1,1)
                          Documentation.describe(0,VAR);

---- Définition des types d'objets élémentaires ----
DEFINE ELEMENTARY_TYPE Moteur (m_id: IDENTIFIER, puiss:INTEGER);

DEFINE ELEMENTARY_TYPE Pièce (p_id: IDENTIFIER, nom:
LIST_OF(CHAR));

DEFINE ELEMENTARY_TYPE Chapitre (c_id: IDENTIFIER, titre:
LIST_OF(CHAR), contenu:TEXT);

---- Définition des liens entre types d'objets élémentaires ----
DEFINE ELEMENTARY_LINK_TYPE
Moteur.to_Pièce(1,VAR),Pièce.to_Moteur(0,VAR);

DEFINE ELEMENTARY_LINK_TYPE
Chapitre.to_Pièce(1,1),Pièce.to_Chapitre(1,1);
```

Le concept de configuration est introduit pour résoudre le problème des liens entre versions d'objets de différents types. En effet, certaines cardinalités fixées pour des liens, n'autorisent pas deux versions d'un objet complexe à partager les mêmes versions de constituants. Pour éviter la solution qui consiste à dupliquer les versions de constituants, OVM utilise la création de configurations. Ainsi, les cardinalités des liens doivent être respectées seulement au sein de chaque configuration.

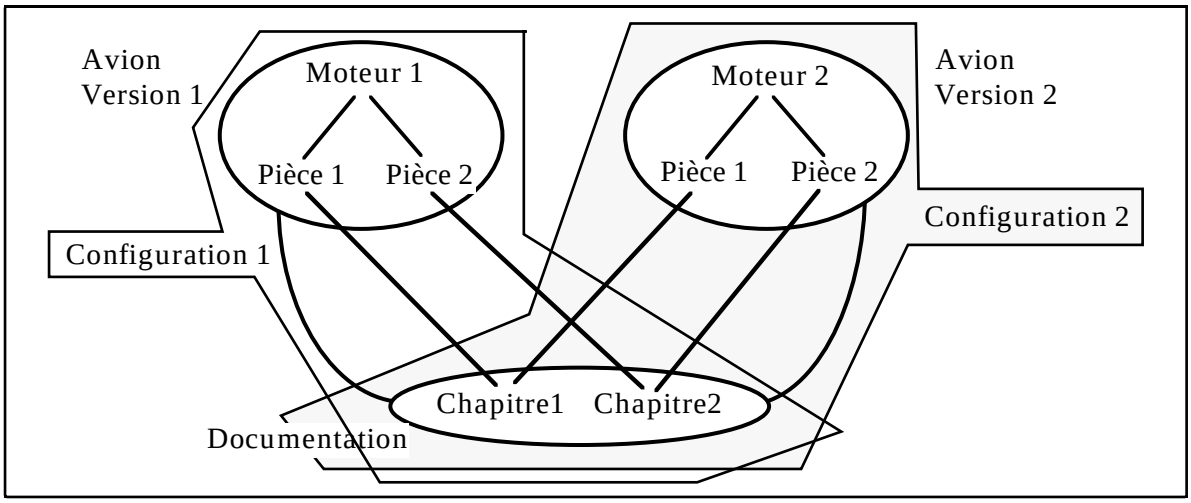


Figure I-3 : gestion de configurations

2.2.3.2. Les opérations dans le système OVM

Les opérations consistent en la création, la modification, la suppression et la recherche d'objets et de versions, ainsi que la création et la recherche de configuration de versions.

OVM différencie la création de l'objet lui-même et la création d'une de ses versions. L'objet est d'abord créé suivant un type défini, et ses attributs sont valorisés. Dans la même opération ou dans une autre opération, la première version de l'objet est créée. Une version peut être créée puis rattachée à l'objet (**CONNECT**), en indiquant sa place dans le graphe (**AS_SUCCESSOR_OF**). La modification est réalisée en créant une nouvelle version.

La manipulation des configurations est effectuée en deux étapes. Tout d'abord, il faut créer une configuration (**CREATE**) et valoriser ses attributs. Ensuite, les versions sont associées à la configuration via l'opération **CONFIGURE**.

Le système OVM fournit l'opération **RETRIEVE** permettant de rechercher des informations sur les objets, les versions, les objets élémentaires appartenant à des versions. Les conditions peuvent porter sur les attributs des objets, des versions ou

des types élémentaires. Le résultat de l'opération **RETRIEVE** est un ensemble d'objets ou un ensemble de configurations.

Les opérations sont construites sur le squelette SQL :

{ **RETRIEVE/CONFIGURE/CREATE/UPDATE**} <projection>

FROM <type d'objet>/<type de configuration>

WHERE <condition>

Exemple I-10 : La création d'un objet de type Avion est réalisée comme suit :

```
CREATE Avion (OBJECT      nom:="Airbus A320"
                          catégorie:="transport passagers")
FROM Avion
--> l'objet Avion reçoit la valeur 456 pour IDENTIFIER
CREATE Avion
    (VERSION nb_pannes:=0, h_de_vol:=50)     -->version_no = 1
    Moteur (puiss:=2000)                     -->m_id=1
    Pièce (nom:="piston")                    -->p_id=10
    Pièce (nom:="turbo-compresseur")         -->p_id=11
FROM Avion
WHERE av_id=456
```

La recherche de versions s'effectue de la manière suivante :

```
RETRIEVE VERSION
FROM Avion
WHERE Avion.nom="Airbus A320" AND Avion.version_no=1
```

Le système OVM a été implanté au-dessus du SGBD pour objets complexes PRIMA (Härder, 87). Les schémas OVM sont traduits en schémas correspondants au modèle de donnée molécule-atome (MAD) du SGBD. De même, les opérations définies dans OVM sont traduites dans le langage de requêtes MQL du modèle MAD ; le résultat fournit par le SGBD est ensuite reconstruit suivant le modèle d'OVM.

2.2.3.3. Bilan sur le système OVM

Le système OVM intègre la notion de version à un modèle pour objets complexes (MAD). Le modèle de versions est basé sur le principe de dérivation, et permet également de limiter l'évolution des objets (linéaire, arborescente, ...). Les objets sont définis, versionnalisés ou non, avec une partie spécifiant les constituants (élémentaires), une partie spécifiant les attributs communs à toutes les versions, et enfin une partie spécifique aux versions. Des liens peuvent être définis entre objets complexes et entre objets élémentaires.

La définition des objets est réalisée en deux étapes : la création de l'objet proprement dit, puis la création d'une version. OVM introduit la notion de configuration, niveau supplémentaire à manipuler, qui regroupe des versions. La notion de configurations est utilisée pour autoriser le partage de versions de constituants, en permettant de vérifier les cardinalités dans chaque configuration.

OVM fournit un langage à structure SQL permettant la recherche d'ensembles d'objets, ou d'ensembles de versions. La sémantique propre aux versions n'est pas prise en compte par ce langage.

OVM ne gère pas les évolutions de schéma. Le modèle de données, bien que puissant, parait lourd à manipuler notamment par la disjonction de l'objet et ses versions, et la manipulation des configurations.

2.2.4. Les modèles Entité/Relation intégrant les versions

Très peu d'études ont été menées pour intégrer les versions dans des modèles sémantiques. Seuls deux systèmes reposant sur des modèles de type Entité/Relation, CERM (Dittrich, 90) et O2XER (Dijkstra, 93), intègrent des notions limitées de versions.

Dans O2XER les versions sont conservées par le système à des fins d'historiques suivant le principe de dérivation ; cependant, pour les entités complexes, une seule version est manipulable, celle constituée de la dernière version de chacun des constituants. De plus, les associations entre objets et versions d'objets sont possibles mais sans définition de cardinalités.

Le modèle CERM développé pour le système DAMOKLES est basé sur le modèle de versions du système XSQL (Dittrich, 88). Des associations sont définies entre classes d'entités. Une entité peut avoir plusieurs versions ; les versions sont gérées indépendamment des associations. Des entités de type configuration peuvent être définies pour lier les versions d'entités associées (une version de chaque entité).

2.2.5. La gestion transparente de versions

L'approche gestion de versions d'objets a également été utilisée de manière interne à un système, non plus pour décrire l'évolution d'entités mais pour gérer les accès concurrents à une base de données, gérer les reprises sur panne, ... Ce type de gestion de version est transparente à l'utilisateur.

Les accès concurrents sont facilités par la création de versions pour les objets manipulés. Cette fonctionnalité est utilisée dans des systèmes tels que SCCS

(Rochkind, 75) pour la manipulation des fichiers, et AVANCE (Björnerstedt, 89). Chaque utilisateur qui accède à un objet manipule en réalité une version de cet objet. La complexité réside dans le mode de fusion de toutes les versions de l'objet pour prendre en compte les modifications de chaque utilisateur.

La création de versions est également utilisée pour la gestion de reprise sur panne dans des systèmes tels que DVSS (Ecklund, 87) et Mosaic (Landis, 86). La gestion de versions est limitée par rapport à celle réalisée au niveau application puisqu'elle se limite à un historique ; elle est effectuée par le système ce qui évite la vérification de la cohérence des versions. La création de versions est réalisée à intervalle régulier ou à la suite d'une transaction. Lors d'une reprise après incident, le système reprend la dernière configuration cohérente de versions des objets de la base.

2.2.6. Gestion de contextes de versions

Le principe des contextes de versions est de regrouper des versions d'entités selon une signification précise (ex. la version de chaque entité utilisée par un utilisateur donné). La notion de contexte permet de faire abstraction d'un certain nombre de versions inutilisées ; elle permet de se replacer dans un espace de travail particulier.

Plusieurs stratégies de définition de contextes ont été proposées (Gançarski, 94a):

- les contextes de transactions (Zdonik, 86), (Katz, 90c) : les contextes regroupent des versions produites par un groupe d'opérations,
- les contextes de versions d'objets complexes (Atwood, 86), (Tichy, 89), (Estublier, 94) : les contextes sont définis par les liens de références entre versions ; une version composée définit un contexte,
- les contextes indépendants de la gestion des versions d'entités (Miller, 89), (Reichenberg, 89), (Cellary, 90), (Kimball, 91) : les contextes sont définis et gérés indépendamment des versions d'entités.

2.2.6.1. Les versions de bases de données

Les travaux sur la gestion de versions de bases de données (Cellary, 90) (Gançarski, 94a)(Gançarski, 94b) proposent une approche générale de gestion de contextes. La gestion de versions d'objets n'est pas réalisée au niveau de chaque objet de manière indépendante. La base fait globalement l'objet d'une gestion de version c'est-à-dire qu'une version de base de donnée rassemble une version de chaque entité représentée. Toute manipulation de version d'objet se fait au travers d'une version de la base. Le principe de dérivation des modèles de gestion de versions d'objets est utilisé, et adapté à la gestion de versions de base de données (VBD). Les différentes VBD sont liées par une relation de dérivation, formant un arbre de dérivation de VBD.

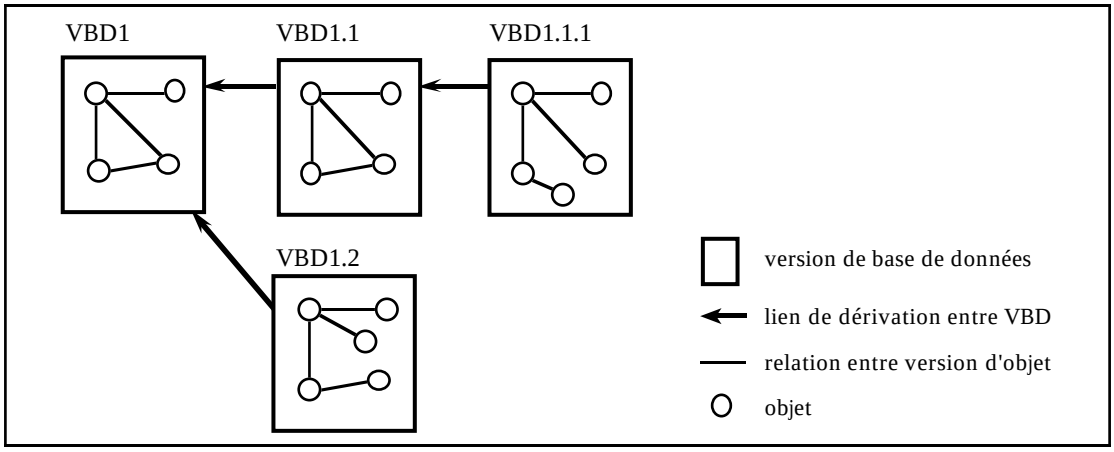


Figure I-4 : Arbre de dérivation de versions de base de données

Le système distingue deux niveaux : le niveau logique, les versions dans une VBD, et le niveau physique, le niveau des versions effectivement stockées. Une nouvelle VBD est créée explicitement par l'utilisateur. Une nouvelle VBD, créée par dérivation d'une VBD existante, est en fait une copie logique de celle-ci (pas de création physique de versions). Les deux VBD ont au départ des versions logiques identiques pour chaque objet de la base ; les versions logiques d'un objet dans les deux VBD partagent la même version physique. L'utilisateur peut ensuite modifier, dans une VBD, les valeurs des versions logiques d'objets (entraînant éventuellement la création de nouvelles versions physiques correspondantes).

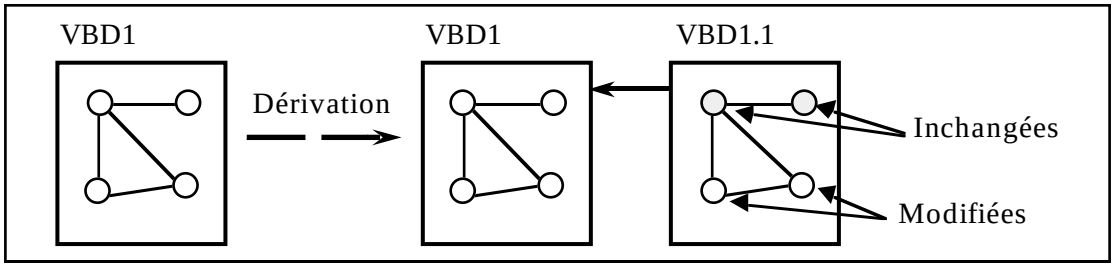


Figure I-5 : Dérivation de VBD

La gestion de versions de schéma a été abordée pour l'approche des VBD (Cellary, 91). Par ailleurs, un modèle formel de l'approche des VBD permet de définir l'ensemble des opérations de base d'un langage général de manipulation. Les bases pour la conception d'une interface utilisateur sont également définies et mises en oeuvre au sein d'un prototype baptisé Modesty développé au dessus du SGBDO O2 (Gançarski, 94a).

2.2.6.2. Les versions de configuration dans le système O2

L'approche choisie par le SGBDO O2, pour fournir un module de gestions de versions (O2, 96), s'apparente à celle des versions de bases de données présentée précédemment ; la gestion n'est pas faite, comme précédemment, sur la base entière mais sur une partie de base via des objets appelés **configurations**. Le module permet

de définir des configurations pour lesquelles on peut gérer des versions suivant le principe de dérivation. Des objets, qui peuvent appartenir à différentes classes, sont ajoutés aux configurations. Un objet ajouté à une configuration sera implicitement versionnalisé ; l'objet peut avoir des valeurs différentes d'une version de configuration à une autre. Des fonctionnalités sont ajoutées au langage de manipulation pour manipuler les versions de configurations. En revanche, la manipulation des objets n'est possible qu'à l'intérieur d'une seule version d'une configuration ; la manipulation des objets dans plusieurs versions de configurations, c'est-à-dire pour manipuler les différentes valeurs prises par un objet dans différentes versions de configuration, s'avère très délicate. Un exemple de manipulation d'une base dans le système O2 est présenté en annexe (cf. Annexe I.).

2.2.6.3. Bilan de l'approche contextes de versions

L'approche contexte de versions a été développé pour des raisons de cohérence (chaque contexte est un état cohérent de la base). Elle est utile, par exemple, lorsque l'on a besoin de représentations multiples d'un système ou lorsque l'on représente l'évolution globale du système. En revanche, elle semble moins adaptée à une manipulation de l'évolution des entités de manière indépendante. En effet, la manipulation des versions d'objets s'effectue au travers des configurations (ou de versions de bases de données). Si l'on considère l'exemple du système O2, pour passer d'une version représentant une entité à une version suivante, il est nécessaire de remonter au niveau des versions de configurations, de changer de configurations et de redescendre au niveau des objets de la version de configuration choisie. La représentation de l'évolution d'une entité (arbre de dérivation de versions d'objet) n'est pas directement manipulable.

Ces systèmes, à l'exception de (Cellary, 90), sont en général spécifiques à un domaine d'application (ex. gestion de configurations logicielles) ou mettent en oeuvre des mécanismes trop complexes pour pouvoir gérer un grand nombre de contextes de taille importante.

2.2.7. Les langages et interfaces pour versions d'objets

Aucun des systèmes de gestion de versions d'objets ne propose un langage prenant en compte la sémantique des versions surtout en matière d'interrogation. Les versions sont généralement considérées comme de simples objets (Adiba, 89), (Kim, 89a), (Käfer, 92) pour lesquels sont ajoutés des attributs spécifiques.

De même, en ce qui concerne les interfaces utilisateur, elles sont inexistantes si l'on met de côté les interfaces hypertextes, riches mais spécifiques à ce type d'application.

Seuls les travaux sur les versions de bases de données (Gançarski, 94a) proposent des débuts de solutions pour un langage et une interface de manipulation adaptés à la gestion de versions de bases de données.

3. L'évolution de schéma

Différents travaux ont porté sur l'évolution des bases de données, non plus au niveau objet mais au niveau de la description de la base c'est-à-dire son schéma. Les travaux menés dans cette direction se divisent en trois principales approches : la modification de schéma avec éventuellement préservation des données, la versionnalisation de schéma et la gestion de vues.

La modification de schéma consiste à pouvoir agir sur le schéma d'une ou plusieurs classes ; seul le schéma après modification est conservé. La modification de schéma peut être complétée par des mécanismes d'adaptation des données existantes au nouveau schéma. On parle de versionnalisation de schéma lorsque celui-ci est conservé avant et après modification ; les données sont manipulables dans chaque version de schéma. La gestion de vues consiste à définir des classes sans instances réelles dans la base de données.

3.1. Approche modification de schéma

Les travaux réalisés dans ce cadre ont essentiellement porté sur l'étude des opérations applicables pour la transformation de schéma. Les solutions se sont ensuite attachées à préserver le contenu sémantique des données au cours des évolutions.

3.1.1. Opérations sur les schémas

Les travaux dans ce domaine se sont attachés à recenser les différentes opérations de modification de schéma :

- les opérations sur les classes c'est-à-dire la création, la suppression de classe,
- les opérations sur la description des classes c'est-à-dire l'ajout, la suppression, la modification des attributs et/ou des méthodes,
- les opérations portant sur la hiérarchie d'héritage.

Plusieurs travaux, pour les systèmes ORION (Kim, 89b), O2 (Zicari, 91), SHOOD (Bounaas, 95), ont établi une liste d'opérations applicables pour la modification de schéma de bases de données orientées objet. Dans le système ORION, les modifications sont de trois ordres :

- modification de la structure d'une classe : ajout d'un attribut, suppression d'un attribut (suppression dans les sous-classes, modification des instances existantes), modification du nom ou du domaine d'un attribut, modification d'héritage d'un attribut (autre attribut de même nom),
- modification des méthodes d'une classe : ajout, suppression, renommage, modification du code, modification de la signature.
- modification du graphe des classes
  - ajouter, supprimer, renommer une classe,
  - rendre une classe S super-classe d'une classe C : la modification ne doit pas introduire de cycle dans le graphe ; C et ses sous-classes héritent des attributs et des méthodes de S (cf. ajout d'attribut et de méthode) et leurs instances reçoivent la valeur nulle pour ces attributs,
  - supprimer une classe S super-classe d'une classe C : la modification ne doit pas introduire de rupture dans le graphe ; tous les attributs et méthodes de C hérités de S (seulement ceux définis dans S) sont supprimés. Les super-classes de S restent super-classes de C, leurs attributs et méthodes restent hérités.

D'autres travaux dans le cadre de systèmes tels que Gemstone (Penney, 87) et OTGen (Lerner, 90) ont considéré d'autres opérations sur les schémas pour augmenter les possibilités d'évolution. Le système OTGen permet notamment de réaliser des opérations de transformation à un niveau plus fin que les opérations classiques. OTGen établit une table de transformation, automatiquement générée par le système pour les opérations de base mais que l'administrateur peut modifier pour affiner la transformation.

Différents travaux sur les systèmes ORION, Gemstone et OTGen ont défini une liste d'invariants permettant d'assurer la validité d'un schéma. Toute opération de modification du schéma doit préserver les invariants.

ORION a établi la liste d'invariants suivante (Banerjee, 87)(Chou, 88), définissant un schéma dit valide :

- invariant graphe de classe : le graphe doit être acyclique orienté connexe,
- invariant nom distinct : tout attribut ou méthode d'une classe a un nom distinct,
- invariant identité (origine) distincte : tout attribut et méthode d'une classe à une identité (origine) unique,
- invariant héritage total : une classe hérite de tous les attributs et méthodes de chaque super-classe sauf violation des invariants de nom et d'identité

(héritage d'au moins un des attributs hérités de même nom mais d'origines différentes, héritage d'un seul des attributs hérités de même nom et même classe d'origine),

- invariant compatibilité de domaine : le domaine d'un attribut est celui dont il hérite ou d'une sous-classe ; la valeur d'un attribut doit être instance de sa classe domaine ou d'une de ses sous-classes.

Pour choisir entre les différents moyens de préserver les invariants, un certain nombre de règles a été établi ; ces règles peuvent être regroupées dans les quatre catégories suivantes :

- règles par défaut de résolution de conflits : pour la résolution de conflits de noms ou d'identités. L'utilisateur peut également résoudre explicitement les conflits,
- règles de propagation de propriétés : pour supporter toutes les modifications des propriétés des attributs et des méthodes (nom, domaine, ...),
- règles de manipulation du graphe : pour gérer les suppressions et ajouts de noeuds et arcs,
- règles d'objets composés : pour prendre en compte la sémantique des objets composés.

(Zicari, 91) propose un outil interactif (Interactive Consistency Checker) destiné à la mise à jour de schéma et proposant un ensemble d'opérations de modification de schéma. L'outil détecte les éventuelles incohérences structurelles ou comportementales induites par les modifications demandées, et il invalide la mise à jour en cas d'incohérence. Des transactions d'évolution peuvent être définies pour réaliser un ensemble d'opérations d'évolution dans une même transaction. Des solutions ont été proposées et mises en oeuvre dans le cadre du SGBDO O2.

3.1.2. Adaptation des données

Il s'agit ici des travaux permettant de faire évoluer le schéma et d'établir des stratégies pour conserver la sémantique des données de l'ancien schéma dans le nouveau.

Différentes approches basées sur le principe de conversion ont été proposées pour conserver les données au cours de l'évolution de schéma. Les données de l'ancien schéma sont converties pour s'adapter au nouveau schéma. Deux stratégies de conversion sont habituellement utilisées, la conversion immédiate (Penney, 87) et la conversion différée (Chou, 88), (Tan, 89), (Lerner, 90). La conversion immédiate consiste à propager immédiatement les modifications de schéma en modifications des instances. La conversion immédiate permet de conserver les mêmes temps

d'accès aux données avant et après modification, mais le processus de conversion peut bloquer les utilisateurs lorsque la base est volumineuse. La conversion différée, en revanche, réalise la propagation sur les instances uniquement à l'instant où elles sont utilisées. La conversion différée permet de ne pas bloquer l'utilisation de la base ; cependant, l'accès aux données est d'autant plus long et l'adaptation d'autant plus compliquée que le nombre de modifications apportées au schéma entre deux accès aux données est grand.

OTGen (Lerner, 90) fournit la possibilité de faire évoluer un schéma selon un jeu d'opérations de base correspondant à celles précédemment citées. Cependant, il fournit également à l'administrateur la possibilité d'effectuer des transformations plus fines, pour l'adaptation des données au nouveau schéma. En effet, pour toute évolution de schéma une table de transformations est produite, sur laquelle il est possible d'agir pour affiner les opérations d'adaptation des instances de l'ancien schéma vers le nouveau. Il fournit la possibilité de réorganiser la base par des opérations supplémentaires :

- initialisation d'attributs,
- modifications dépendantes du contexte,
- déplacement d'informations d'un objet vers un autre,
- création de nouveaux objets,
- partage d'information entre objets.

Exemple I-11 : Supposons que l'évolution du schéma d'une classe C consiste à ajouter un attribut pour lequel on affecte la valeur 0 pour tous les objets existants. La déclaration d'adaptation des instances dans la table de transformation est la suivante :

Class C:
  new self: C
    a1: **Transform** a1
    a2: **Transform** a2
    a3: 0

Les deux attributs a1 et a2 sont conservés sans modification (ex. a1 transformé en a1) et la valeur 0 est affectée à l'attribut a3.

D'autres études, menées dans le cadre du système Boswell, ont porté sur les opérations de modification possibles sur des schémas relationnels et des schémas Entité / Association (Roddick, 93). Les travaux présentent également comment les modifications apportées à un schéma Entité / Association sont traduites en opérations sur le schéma relationnel correspondant.

Une liste d'évolutions possibles pour un schéma Entité/Association est établie :

- au niveau entité : ajout d'une classe d'entités, suppression d'une classe d'entités, modification d'une classe d'entités en attribut d'une association, ...
- au niveau attribut : ajout d'un attribut à une classe d'entités, suppression d'un attribut dans une classe d'entités, suppression d'un attribut d'une association, modification d'un attribut d'une association entre classe d'entités, ...
- au niveau association : ajout d'une association entre deux classes d'entités, ajout d'un lien entre une classe d'entités et une association, ...

Ces travaux sont menés dans le cadre de modèles Entité/Association implantés au-dessus de systèmes relationnels.

3.2. Approche versions de schéma

Une approche différente pour permettre l'évolution de schéma est celle de la gestion de versions. Cette approche consiste à conserver le schéma avant modification et le schéma modifié comme deux versions du schéma (Skarra, 86), (Kim, 88), (Clamen, 92), (Monk, 93).

Toute modification du schéma d'une classe donne lieu à la création d'une nouvelle version de la classe. Toutes les versions d'une classe sont conservées. Chaque version de la classe représente une interface pour chaque entité représentée dans cette classe.

Au niveau des objets d'une classe, plusieurs stratégies ont été proposées. Dans le système Encore (Skarra, 86)(Zdonik, 86), la stratégie est celle de l'émulation (screening). Une instance est associée à la version de classe dans laquelle elle a été définie. Chaque instance est ensuite "émulée" dans les autres versions de la classe c'est-à-dire que le système simule en fait la sémantique de la nouvelle interface au-dessus des instances de l'ancienne et vice-versa.

A chaque classe est associée une interface standard, union des interfaces des versions de la classe. L'émulation est réalisée par la définition de procédures d'exception (handlers) dans les versions de classes par rapport à l'interface standard. Ces procédures d'exception pallient les erreurs provoquées par l'utilisation d'objets créés sous des classes dont l'interface a été changée.

Le principe d'émulation souffre cependant d'un manque de performance. De plus le fait qu'un objet reste toujours instance de la version de classe dans laquelle il a été défini pose un problème lors de la définition d'une nouvelle version d'une classe avec ajout d'un attribut ; en effet, il n'est pas possible aux objets créés dans une

version de classe antérieure d'utiliser cet attribut, excepté pour stocker une valeur constante en utilisant une procédure d'exception.

Le système CloSQL (Monk, 92)(Monk, 93) a repris quant à lui la stratégie de conversion présentée précédemment. Une instance est créée dans une version de classe. Les instances sont converties lorsqu'il y a conflit de versions. La conversion peut être effectuée aussi bien vers une version de classe plus récente que vers une version plus ancienne.

Clamen (Clamen, 92)(Clamen, 94) a adopté une stratégie légèrement différente afin d'éviter les processus d'émulation ou de conversion. La stratégie de Clamen est une généralisation de celle de Zdonik. Chaque instance est composée de plusieurs facettes. Chaque facette encapsule l'état d'une instance pour une interface (i.e. une version de la classe). La représentation de ces instances est l'union disjointe des représentations de chacune des versions de classe. Les relations entre 2 versions d'une classe sont réalisées par les attributs ; ceux-ci sont regroupés en quatre catégories :

- partagé : attributs communs aux deux versions,
- indépendant : la valeur d'un attribut n'est pas affectée par les modifications de la valeur de cet attribut dans l'autre facette,
- dérivé : la valeur d'un attribut peut être directement dérivée de la valeur de l'attribut dans l'autre facette,
- dépendant : la valeur de l'attribut est affectée par les modifications dans l'autre facette mais n'est pas directement calculable.

Lorsque la valeur d'un attribut d'une facette est modifiée, les attributs qui dépendent de celui-ci dans l'autre facette sont mis à jour (copie de valeur pour un attribut partagé, recalcul de la fonction pour un attribut dérivé ou dépendant).

Clamen propose également un processus d'évolutions composées. Une construction, appelée configuration de versions, maintient les dépendances entre les classes qui évoluent.

3.3. Approche gestion de vues

Cette approche s'inspire de la notion de vues établie pour les bases de données relationnelles. La gestion de vues dans les bases de données orientées objet a été introduite pour les mêmes raisons qu'en relationnel à savoir la restructuration des données, la simplification de requêtes, la gestion de la confidentialité des données.

Certains travaux, tels que (Bertino, 92), (Souza, 93), ont utilisé la gestion de vues pour gérer l'évolution de schéma. L'utilisation de vues permet de simuler les évolutions de schémas.

(Bertino, 92) définit une relation de dérivation de vues qui lie les vues de classes et leurs classes sources. Il est possible de modifier (ajout, suppression, ...) la liste d'attributs, de méthodes et également l'héritage au niveau de la définition d'une vue.

Exemple I-12 : définition d'une vue dans le modèle de Bertino :

```
create-view  Nom_de_vue    (properties Noms_de_propriétés)
                           (view-query Requête)
                           (additionalproperties Propriétés)
                           ... )
```

(Souza, 93) propose la possibilité de définir des classes virtuelles (définies par une requête), des classes imaginaires (permettant la composition ou décomposition d'objets de différentes classes), des schémas et des bases virtuelles.

Exemple I-13 : définition d'un schéma virtuel

```
virtual schema family_view from person_schema;
import all;
virtual class Young_Person includes
        select p from p in Person where p.age > 21;
imaginary class Family includes
        select [husband:h,wife:w] from h,w in Person
        where h->spouse = w and w->spouse = h and h->sex = 'M';
...
```

Ces propositions ont été mises en oeuvre au sein du prototype O2Views développé autour du SGBD O2.

4. L'évolution aux deux niveaux

Certains travaux, tels que les systèmes ORION (Kim, 88), CHARLY (Palisser, 89), SHOOD (Nguyen, 93) et Presage (Talens, 94) ont tenté de gérer l'évolution aux deux niveaux d'abstraction que sont le niveau objet et le niveau des classes. Il s'agit ici de considérer les études qui permettent de manipuler une représentation de l'évolution des objets et du schéma d'une base de données.

4.1. Les différentes approches

Différentes études concernant le système ORION (Chou, 88)(Kim, 88) ont proposé des solutions pour la gestion de versions d'objets et la gestion de versions de schéma. ORION offre la possibilité de gérer les deux niveaux simultanément. Le modèle de versions d'objets repose sur le principe de dérivation et autorise les alternatives. Les entités sont représentées par un objet "abstrait" et un ensemble de versions. ORION permet de plus de définir des liens de composition pour la gestion d'objet complexes.

Au niveau des classes, des études (Kim, 88) ont proposé la possibilité de gérer des versions de schéma. Plusieurs versions d'un schéma peuvent être manipulées alternativement. Simultanément, il est possible de manipuler des versions d'objets pour chaque version de schéma. Par défaut, toute nouvelle version du schéma implique la création de toutes les versions d'objets appartenant à la version de schéma précédente. Il est cependant possible de limiter l'ensemble des objets pour lesquels une nouvelle version est créée. De plus, de nouveaux objets et de nouvelles versions d'objets existants peuvent être créés au niveau d'une version de schéma. Les versions de schéma ne sont cependant pas totalement recréées, seule la différence (delta) est créée physiquement.

Le système CHARLY (Palisser, 89)(Palisser, 90) propose quant à lui un processus de gestion de versions au niveau objet et permet simultanément de faire évoluer le schéma. CHARLY est avant tout un système de gestion de versions d'objets basé sur le principe de dérivation de versions et dédié à la conception dans le domaine de l'architecture. L'héritage n'est pas un concept pris en compte. La possibilité de définir des alternatives est limitée à un axe de référence. Un chemin de dérivation de version est pris comme axe de référence et toute alternative n'est définie qu'à partir d'une version appartenant à cet axe.

CHARLY offre la possibilité de faire évoluer le schéma à tout moment mais est limité à l'ajout, la suppression ou la modification d'attribut. Cependant, seul le dernier schéma est utilisé. Au niveau du stockage, CHARLY utilise la technique des fichiers différentiels (Rochkind, 75) pour stocker les versions d'objets en y intégrant l'évolution du schéma.

Le projet SHOOD (Nguyen, 89)(Escamilla, 90)(Rieu, 91)(Nguyen, 93) a pour but de gérer l'évolution, au sens large, dans les systèmes de gestion de bases de données. Des travaux ont notamment proposé des solutions pour permettre les évolutions de schémas en intégrant la notion de versions d'objets. L'étude a été orientée sur la gestion d'objets évolutifs, et notamment sur l'incomplétude et l'incohérence des données par rapport à un schéma tout au long d'un processus de conception.

L'incomplétude et l'incohérence des instances sont gérées à l'aide de structures dites "significatives". A partir d'une classe initiale, l'ensemble des structures qui représentent une réalité (structures significatives) sont générées. Une instance est rattachée à la structure significative maximale, c'est-à-dire la structure pour laquelle tous les attributs sont valorisés. Le rattachement d'une instance à une structure permet de déterminer son état de complétude et de cohérence.

L'évolution d'un objet est basée sur la notion d'amélioration vis-à-vis de règles structurelles. Si lors de son évolution, l'objet ne "s'améliore" pas selon les règles, le système génère une **boucle de dégradation** constituée des versions de l'objet après chaque modification (Escamilla, 90). La boucle de dégradation est complétée par toute nouvelle version jusqu'à ce que l'objet possède un degré de complétude et un degré de cohérence supérieur à la version initiale. Dès qu'il y a amélioration par rapport à la première version, toute la boucle de dégradation est supprimée et l'instance n'est plus versionnalisée. La boucle peut être un arbre si l'utilisateur décide de repartir d'une des versions de la boucle, autre que la version courante, pour simuler des opérations de mise à jour.

Le modèle SHOOD intègre différents liens sémantiques (Djeraba, 93), notamment pour la représentation d'objets complexes et la gestion de versions d'objets. La gestion de versions d'objets repose sur le principe de dérivation de versions (cf. § 2.2.2.1.).

L'évolution de schéma dans SHOOD (Nguyen, 89)(Bounaas, 95) prend en compte les modifications au niveau du graphe des classes (ajout de classe, création de superclasse, ...), au niveau des classes (ajout d'attribut, ...), au niveau des liens entre classes ainsi qu'au niveau des contraintes. La modification de schéma induit des vérifications afin que le schéma reste cohérent vis-à-vis du graphe de spécialisation, et que les instances correspondent encore au schéma modifié. SHOOD offre un processus immédiat de conversion des instances (après la modification du schéma). Le modèle SHOOD est en plus un modèle actif qui permet de spécifier les règles d'évolution de classes et d'instances.

Le projet SHOOD s'oriente actuellement vers l'ingénierie simultanée ou concourante (travail en équipes par projet et méthodes à forte connectivité entre divers intervenants), par la définition de méthodes et d'outils d'aide à la spécification pour les plates-formes d'intégration (Guetari, 96).

Le système Presage (Talens, 94) propose une gestion de versions à la fois des objets et des classes. Les versions d'objets, mais également les versions de classes, suivent le principe de dérivation et autorise la formation d'une arborescence de dérivation. La définition de versions d'objets et de versions de classes peuvent être combinées. La définition d'une version de classe se fait indépendamment de celle des versions

d'objets c'est-à-dire qu'elle n'entraîne pas de nouvelle version des objets. Par contre, la dérivation d'une version d'objet n'est pas limitée à l'évolution de la valeur de l'objet mais peut intervenir au niveau du schéma. La dérivation d'une version peut en fait être effectuée d'une version de classe à une autre. Les versions qui correspondent à la même entité peuvent appartenir à différentes versions de classes.

Ainsi, le système Presage permet de manipuler à la fois l'évolution d'une entité en termes de valeur et de schéma. Presage permet la représentation d'objets complexes, sans gérer de contraintes via des cardinalités. De plus, il ne fournit pas de langage d'interrogation des versions.

4.2. Exemples de systèmes gérant des versions aux deux niveaux

Les systèmes présentés ont proposé deux stratégies différentes pour gérer l'évolution au niveau des objets et des classes par la gestion de versions. Le système ORION (Kim, 88) gère des versions d'objets et de schémas de manière globale mais de manière indépendante. Le système Presage (Talens, 94) permet la gestion de versions d'objets et de versions de classes de manière uniforme en combinant les deux gestions de versions.

4.2.1. Le système ORION

Les études autour du système ORION ont abordé la gestion de l'évolution aux deux niveaux d'abstraction que sont le niveau schéma et le niveau objet. Des travaux sur la gestion de versions et l'évolution de schéma ont été menés séparément. Ces travaux ont ensuite été repris pour définir un système de gestion de versions de schéma intégrant la gestion de versions d'objets.

4.2.1.1. Le modèle de versions d'objets de ORION

Le modèle de versions d'objets (Chou, 86) est basé sur le principe de dérivation de versions. Une entité du monde réel est représentée par un objet "générique" qui regroupe un ensemble de versions organisées en hiérarchies de dérivation. Les versions sont de deux types : **temporaires** c'est-à-dire modifiables et supprimables, et **de travail** c'est-à-dire non modifiables mais supprimables. Une version temporaire est créée à partir de rien ou dérivée d'une version précédente. Les versions temporaires passent ensuite à l'état de travail soit explicitement spécifiée par le concepteur soit implicitement par le système lors de dérivation.

ORION permet de gérer des versions d'objets complexes en définissant des liens de composition entre objets (Kim, 87)(Kim, 89a). Une version d'un objet peut

référencer d'autres versions d'objets ou des objets génériques. La liaison entre versions est une liaison statique ; une version d'un objet est liée à une version particulière d'un autre objet. En revanche, la liaison version-objet générique est une liaison dynamique ; une version d'un objet n'est pas liée à une version précise de l'autre objet. En fait, la version d'objet est liée à la version par défaut associée à l'objet générique, cette version pouvant changer au cours du temps. La version par défaut d'un objet générique est une version de la hiérarchie de dérivation associée à l'objet générique ; elle est choisie par le concepteur ou désignée par le système en fonction de la date de création.

4.2.1.2. L'évolution de schéma dans ORION

Le schéma d'une base de données orientée objet est un graphe de classe ; ORION définit donc deux types de modification de schéma (Kim, 89b) : modification de la définition d'une classe du graphe (contenu d'un noeud du graphe) et modification de la structure du graphe (noeuds et arcs).

Les modifications de définition de classes sont l'ajout et la suppression d'attributs et de méthodes. Les modifications du graphe sont l'ajout, la suppression de classe et la modification de la relation EST_UN entre classes (ajouter ou supprimer une relation est-un entre deux classes). Ces opérations ont été détaillées précédemment (cf. § 3.1.1.).

4.2.1.3. Versions de schéma dans ORION

Le modèle de versions de schéma du système ORION (Kim, 88) reprend les principaux concepts établis par son modèle de versions d'objets c'est-à-dire le principe de dérivation de versions, les notions de version temporaire et de travail, et la version par défaut. Les modifications possibles pour définir une nouvelle version de schéma sont celles citées précédemment et qui correspondent aux travaux sur l'évolution de schéma.

L'utilisateur se place dans une seule version du schéma ; il n'est pas possible de manipuler les instances simultanément dans deux versions du schéma. Une nouvelle version du schéma reprend automatiquement les versions d'objets et les objets de la version de schéma précédente. Il est possible de définir de nouvelles versions d'objets et de nouveaux objets dans la nouvelle version de schéma. Toutes les modifications apportées aux instances d'une version du schéma sont visibles dans les versions de schéma issues de celle-ci ; les modifications ne sont pas visibles dans les versions de schéma antérieures.

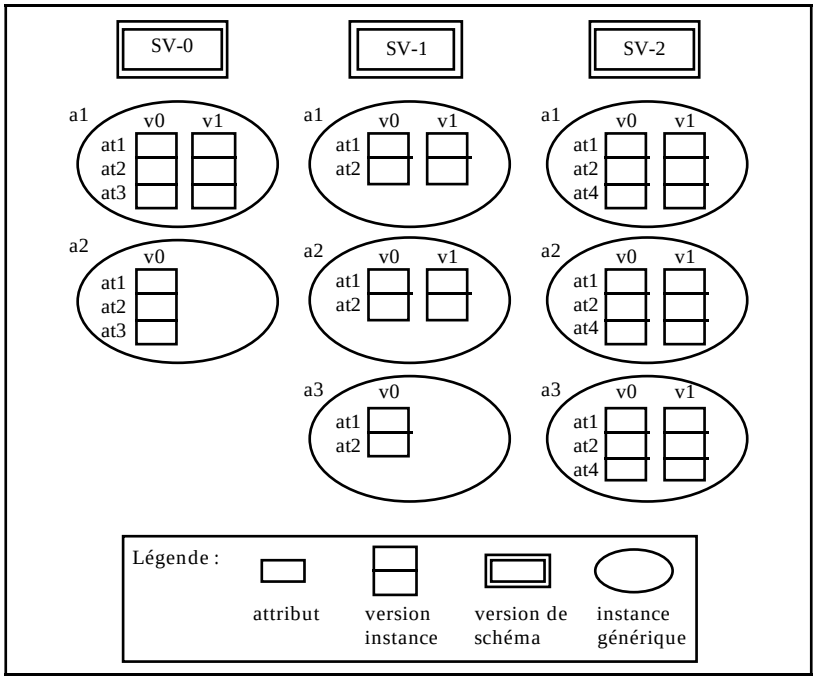


Figure I-6 : Exemple de versions de schéma

Sous la version de schéma initiale SV-0, deux objets versionnalisés a1 et a2 sont créés, en créant leur version initiale a1.v0 et a2.v0. Une seconde version a1.v1 est créée à partir de a1.v0. Une nouvelle version de schéma SV-1 est ensuite dérivée de SV-0, en supprimant l'attribut at3. Tous les objets créés sous SV-0 sont visibles sous SV-1. Une nouvelle version de a2 est créée ainsi qu'un nouvel objet versionnalisé a3 (via sa version initiale a3.v0). Une troisième version de schéma SV-2 est créée, dérivée de SV-1, en ajoutant un attribut at4. Tous les objets visibles de SV-1 sont visibles dans SV-2 avec en plus l'attribut at4. Une nouvelle version de a3 est dérivée sous SV-2.

4.2.1.4. Bilan sur le système ORION

Le système ORION a proposé des solutions pour gérer l'évolution au niveaux des objets et du schéma, d'abord dans des approches séparées puis en intégrant les deux approches. ORION a proposé un modèle de versions de schéma intégrant les versions au niveau objet. Des règles sont établies pour la définition de versions de schéma et les conséquences possibles sur ses objets (reprise de tous les objets de l'ancienne version de schéma ou d'aucun, modification interdite dans l'ancienne version de schéma sauf indication explicite).

Les versions de schéma sont manipulables alternativement c'est-à-dire que l'on ne peut manipuler les versions d'objets qu'au sein d'une version de schéma. De plus, la définition d'une nouvelle version de schéma agit de manière globale sur les instances, c'est-à-dire que tous les objets et les versions d'objets de la version de

schéma précédente se retrouvent dans la nouvelle version de schéma ; versions d'objets et versions de schémas ne sont pas utilisées de manière complémentaire.

4.2.2. Le système Presage

Le système Presage (Talens, 93)(Talens, 94)(Urtado, 96) aborde la gestion de versions au niveau des objets et des classes. Les versions d'objets et de schémas sont utilisées de manière complémentaire mais non dépendante. La gestion des versions aux deux niveaux s'effectue de manière uniforme, c'est-à-dire en utilisant les mêmes concepts.

Le modèle de version est générique mais plus particulièrement adapté à des applications de modélisation hiérarchisée / multi-vues (ex. planification de réseaux de télécommunications).

4.2.2.1. Le modèle de versions du système Presage

Le système Presage définit un modèle de versions basé sur une approche orientée objet, c'est-à-dire utilisant les concepts de classe et d'instance. Le modèle de versions défini est applicable pour les instances et pour les classes.

Le système s'adresse à deux types d'utilisateurs :

- le concepteur d'application, spécialiste du domaine qui est censé utiliser les versions de classes pour définir les modèles appropriés pour le type d'application du domaine.
- l'utilisateur final, qui utilise le modèle d'application pour construire sa propre application et fournir les données initiales. Il utilise les versions d'instances.

Une entité du monde réel est décrite par un arbre de dérivation de versions. Les versions d'objets, exceptée la première, sont créées par dérivation d'une précédente. Il est possible d'obtenir des alternatives (excepté à la racine) en dérivant plusieurs versions à partir de la même version.

Une classe peut être définie versionnalisable ou non. Plusieurs versions de classes peuvent être définies pour une classe versionnalisable.

Les versions de classes sont combinées aux versions d'objets. Les versions d'une classe versionnalisable appartiennent en fait à l'une des versions de classe associées à celle-ci. La dérivation des versions d'objets peut s'effectuer d'une version de classe à une autre.

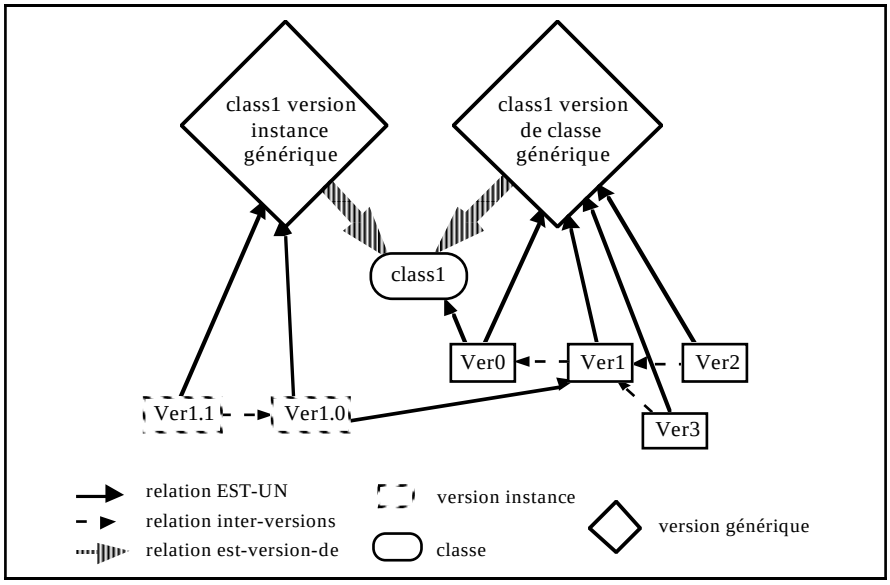


Figure I-7 : Versions de classes et versions d'instances

Presage définit également deux états possibles pour les versions (de classes ou d'objets) : permanent et temporaire. Une version permanente est stable, elle n'est donc pas modifiable mais peut être supprimée. Une version temporaire, est une version feuille dans l'arbre de dérivation, elle peut être modifiée et supprimée.

Une version temporaire peut être dérivée d'une version temporaire ou permanente. Une version temporaire devient permanente, explicitement sur décision de l'utilisateur, ou implicitement lorsqu'elle est à son tour dérivée. Une version est créée temporaire par défaut. Une version permanente peut redevenir temporaire si aucune version n'a été dérivée à partir d'elle.

Les différentes versions dérivées sont liées par les relations :

- "est-dérivée-de-avec-exception" : qui spécifie les attributs supprimés dans la nouvelle version.
- "est-dérivée-de-avec-ajout" : qui spécifie les attributs ajoutés dans la nouvelle version.
- "est-dérivée-de-avec-modification" : qui indique les attributs modifiés dans la nouvelle version.
- "est-dérivée-de-avec-référence" : qui permet, lorsqu'une version a plusieurs prédécesseurs, d'indiquer la version choisie en cas de conflit d'attributs.

4.2.2.2. Versions d'objets composés dans Presage

Presage permet la manipulation d'objets à structure complexe c'est-à-dire composés d'autres objets pouvant à leur tour être composés, formant ainsi une hiérarchie. L'objet au plus haut de la hiérarchie de composition est appelé objet composé (ou composite).

Les versions composées peuvent être créées :

- explicitement par l'utilisateur, de manière descendante, ascendante ou mixte c'est-à-dire que l'ordre de création entre les versions composantes et la version composée n'est pas imposé,
- implicitement par le système, en propageant les créations de versions composantes.

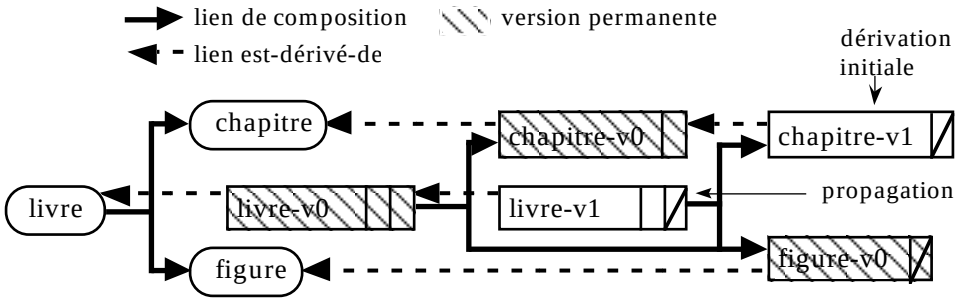
La propagation de création de versions consiste à créer automatiquement des versions d'un objet composé lorsqu'il y a création d'une nouvelle version d'un composant. La propagation, si elle n'est pas limitée, conduit invariablement à la création d'un nombre trop important de versions. Le système Presage propose deux mécanismes pour limiter la propagation : la notion d'**attribut sensible** inspirée de celle proposée par (Zdonik, 86), et la notion de **version sensible**.

Un attribut matérialisant une composition entre versions, désigné sensible, propage la création de versions de composants (sensible ascendant) ou la création de versions qu'elle compose (sensible descendant) ou les deux (sensible mixte) tandis que les attributs qui ne sont pas sensibles (sensible faux) ne la propagent pas. La propagation n'est réalisée que lorsque la version composante créée devient permanente.

Exemple I-14 : Considérons la classe livre telle que :

livre :
Composé-de --> ((*chapitre* contenu) (*figure* illustration))
contenu --> domaine : *chapitre*
sensitif : ASCENDANT
illustration --> domaine : *figure*
sensitif : FAUX

Soit une version de livre livre-v0 composée d'une version de chapitre chapitre-v0 et d'une version de figure figure-v0. L'attribut sensible ascendant *contenu* implique que la création d'une nouvelle version de chapitre chapitre-v1 entraîne la création d'une nouvelle version de livre livre-v1.



En revanche, une nouvelle version de figure figure-v1 n'entraîne pas une nouvelle version de livre.

En plus des attributs sensibles, l'utilisateur peut désigner une version comme étant sensible à la propagation. Lorsqu'une version est déclarée sensible, elle propage la création d'une version d'une de ses versions composantes ainsi que la création d'une des versions qu'elle compose. Lorsqu'une version d'un composant est créée, la propagation implique la dérivation des versions sensibles qu'elle compose ; de même, la dérivation d'une version composée implique la dérivation de ses versions composantes sensibles. La notion de version sensible est prioritaire à celle d'attribut sensible ; pour une version sensible, la propagation est effectuée aussi bien pour les attributs sensibles que ceux qui ne le sont pas.

4.2.2.3. Bilan sur le système Presage

Le modèle de versions du système Presage est un modèle unifié pour la gestion de versions d'objets et de versions de classes. Versions d'objets et versions de classes sont utilisées de manière complémentaire, c'est-à-dire que la définition de versions de classes n'a pas de conséquences directes sur les versions d'instances.

Le modèle permet la manipulation de versions d'objets complexes. Deux principes sont introduits pour gérer la propagation lors de création de versions composantes. D'une part, les attributs désignant des composants peuvent être définis sensibles autorisant ainsi la propagation. D'autre part, des versions peuvent également être désignées sensibles pour activer la propagation ; la notion de version sensible est prioritaire à la notion d'attribut sensible c'est-à-dire que, pour la version, tous les attributs sont sensibles.

Cependant, le système Presage ne propose pas de langage d'interrogation des versions d'instances. De plus, l'expression de contraintes sur les liens de composition via des cardinalités et le maintien des contraintes inhérentes à la composition ne sont pas considérés.

5. Les apports et les limites des travaux existants

Les travaux réalisés jusqu'ici apportent leurs contributions essentiellement au niveau de l'implantation d'une base de données. L'apport des systèmes existants est de différents ordres :

- des solutions pour l'optimisation du stockage des versions comme le stockage différentiel (Rochkind, 75), (Ben Amouzegh, 86), (Palisser, 89).
- la gestion du temps notamment dans les langages d'interrogation pour les bases de données temporelles comme le langage TSQL2(Snodgrass, 94),

- les principes fondamentaux de la gestion de versions (Rieu, 85), (Ben Amouzegh, 86), (Katz, 86), comme le principe de dérivation de versions, la distinction d'états pour les versions ou la définition des opérations de bases pour les versions (dérivation, gel, ...),
- des solutions pour optimiser la gestion de versions d'objets complexes comme les principes de propagation de création de versions (Urtado, 96),
- la gestion de contextes de versions, comme la gestion de versions de bases de données (Gançarski, 94a) ou de versions de configurations (O2, 96), pour conserver des évolutions d'espaces de travail. Ces approches proposent un principe général pour la gestion de versions,
- des solutions pour faire évoluer le schéma d'une base de données comme la définition d'opérations sur les classes (Chou, 88), (Bounaas, 95) et sur les relations entre classes (Roddick, 93), ou les techniques d'adaptation des instances (Lerner, 90).

Cependant, certaines limites et besoins subsistent, notamment pour les systèmes orientés vers un domaine d'application donné :

- il est nécessaire de disposer de moyens pour décrire des bases de données intégrant des versions. Aucun modèle conceptuel n'offre réellement la possibilité de décrire avec précision l'évolution du monde réel. Les modèles CERM(Dittrich, 90) et O2XER(Dijkstra, 93), de type Entité/Relation intègrent de manière limitée les versions (pas de cardinalité pour les associations objets-versions, pas de gestion simultanée de versions d'objets et de classes),
- les langages de manipulation, des systèmes intégrant des versions, sont absents comme dans le système Presage (Talens, 94) ou limités aux objets. Dans les systèmes de gestions de versions tels que ORION (Kim, 89b) ou Aristote (Fauvet, 92), les versions sont interrogées comme de simples objets c'est-à-dire sans tenir compte de leurs spécificités,
- il n'existe pas, à notre connaissance, d'interface utilisateur pour la manipulation des versions dans les bases de données. Les bases pour la conception d'une interface utilisateur sont définies uniquement pour l'approche gestion de versions de bases de données (Gançarski, 94a).

Dans cette thèse, nous proposons plusieurs contributions pour combler ces limites, notamment vis à vis des besoins de domaines d'application gérant de la documentation technique.

6. Notre approche pour la gestion de versions

Nos travaux sont orientés vers les besoins en gestion de documentation technique. Nous avons menés au sein de notre équipe, en collaboration avec le secteur industriel, plusieurs études pour répondre à différents besoins en matière de documentation technique :

- pour gérer les informations relatives au marché des composants électroniques pur la société IBM-Compec (Sallaberry, 92),
- pour manipuler la documentation technique aéronautique pour la formation des équipages et personnels de maintenance pour la société Aéroformation (Thomazeau, 93),
- pour consulter les informations sur l'activité scientifique des laboratoires à l'intérieur du CNRS pour le SOSI-CNRS (Dubac, 95).

Une nécessité supplémentaire est la prise en compte des versions pour de telles applications.

Dans les applications auxquelles nous avons été confrontés, nous avons recensé plusieurs besoins en gestion de versions. Le niveau manipulé est celui de l'entité. Un concepteur de base doit pouvoir décrire avec précision le monde réel en distinguant, parmi les classes qu'il modélise, celles pour lesquelles on gérera des versions au niveau des valeurs et/ou au niveau du schéma. Il faut également pouvoir manipuler simultanément, pour une même entité, des versions ayant des schémas différents ; par exemple, lorsque la structure des documentations des A320 évolue, les versions de documentations des A320 déjà réalisées, restent et sont manipulées dans leur structure d'origine. Pour chaque entité, l'utilisateur définit les versions (c'est-à-dire les états significatifs) qu'il veut conserver, pour décrire l'évolution de cette entité. L'utilisateur veut ensuite pouvoir retrouver certaines versions ou l'évolution des entités telles qu'il les a décrites.

Deux approches se distinguent pour gérer les versions :

- la gestion de versions d'entités, c'est-à-dire au niveau de chaque objet et au niveau de chaque classe ; c'est l'approche courante,
- la gestion de contextes de versions, c'est-à-dire la gestion de versions à un niveau plus élevé à savoir des versions de configurations ou des versions de bases de données.

La gestion de versions d'entités conserve les liens de dérivation entre versions ; ces liens sont ainsi directement exploitables lors de la manipulation. La gestion de versions d'objets complexes est cependant lourde dans certains cas. La gestion de

contextes propose un principe général de manipulation en dissociant la gestion de versions, des objets manipulés par l'utilisateur ; cette approche évite les inconvénients liés aux objets complexes, en se replaçant dans le cadre des bases de données monoversions.

Compte tenu des besoins recensés auxquels nous voulons répondre, la gestion de versions d'entités semble bien adaptée ; c'est l'approche généralement choisie pour ce type d'applications (Käfer, 92), (Talens, 94). La gestion de contextes introduit un niveau supplémentaire, qu'est le contexte, dont la manipulation ainsi que la perte des liens entre versions d'objets, semble rendre plus difficile l'obtention des versions d'objets recherchées par un utilisateur ; par exemple, dans le système O2 basé sur cette approche, certaines des requêtes généralement formulées ne sont pas exprimables (cf. Annexe I.).

La plupart des études sur la gestion de versions ont été menées au niveau système ; les modèles d'implantation sont généralement complexes à utiliser. Pour répondre à cela, nous proposons un **modèle conceptuel** objet intégrant les versions au niveau des objets et des classes (Andonoff, 95)(Hubert, 95a)(Andonoff, 96) ; ce modèle, puisque conceptuel, est a priori plus simple à utiliser. Le modèle repose sur le principe de dérivation de versions. La gestion de versions d'objets et la gestion de versions de classes sont combinées (Hubert, 95b), pour décrire le monde réel et son évolution avec précision ; il est ainsi possible d'avoir des versions d'objets ou des objets avec ou sans versions de classes. Des liens d'héritage, de composition et d'association peuvent être définis entre tous les types de classes possibles.

Compte tenu du type de gestion de versions choisi, il est nécessaire de trouver des solutions pour la gestion des liens de composition, d'association et d'héritage. Il est nécessaire d'étudier les différents cas de figures pour ces liens notamment les contraintes inhérentes à leur définition. Ces contraintes sont également prises en compte d'un point de vue dynamique ; le respect des contraintes par les opérations sur les classes et les instances est étudié.

De plus, nous proposons un **langage**, associé au modèle, pour la manipulation des classes et des instances. Une interrogation de type Select From Where est proposé ; il est possible d'interroger les différents types de classes en prenant en compte les spécificités des versions tout en conservant les possibilités d'interrogation des classes sans gestion de version. Le langage d'interrogation permet de répondre aux types de requêtes recensés.

Enfin, nous proposons une implantation du modèle conceptuel et du langage associé, au travers d'une interface utilisateur (Le Parc, 96a), basé sur SGBD O2. Le modèle et le langage servent de base à la définition d'une interface graphique

destinée à des utilisateurs occasionnels. Cette interface répond à un besoin exprimé par ces utilisateurs occasionnels et constitue une finalité de nos travaux.

Notons enfin que notre étude n'est pas spécifique au modèle proposé ; elle peut s'appliquer aux modèles des méthodes de conception existantes fondées sur l'approche objet (Flory, 90), pour les étendre à la gestion des versions. Ainsi, des méthodes telles que OOA(Coad, 91), OMT(Rumbaugh, 91), OOM(Rochfeld, 92), pourraient répondre aux besoins des domaines tels que la documentation technique, la conception assistée par ordinateur, ou le génie logiciel.

Système	*Modèle*	*Evolution des instances*	*Evolution de schéma*	*Liens sémantiques*	*Contraintes*	*Langage*	*Interface*
TSQL (Navathe, 89)	relationnel	historique	-------	clé primaire/ clé étrangère	-------	SQL temporel	-------
(norme) TSQL2 (Snodgrass, 94)							
TQUEL (Snodgrass, 87)	relationnel	historique	-------	clé pri./clé étr.	-------	QUEL temporel	-------
TOODM-TOOSQL (Rose, 91)	objet	historique	-------	héritage, associations	contraintes sur les valeurs & temporelles	TOOSQL : SQL objet temporel	-------
ORES (Theodoulidis, 94)	entité/relation	historique	-------	composition, association	-------	ERT-SQL : SQL temporel	-------
GemStone (Penney, 87)	objet	conversion immédiate	modification	héritage, référence	contraintes de valeurs	N.P.	-------
NO2 (Lerner, 90)	objet	conversions complexes	modifications complexes	héritage, composition	contraintes d'éxclusivité et de dépendance	Quod : langage spécifique l'évolution de schéma	-------
ICC-O2 (Zicari, 91)	objet	-------	modification	héritage, référence	-------	OQL de O2 + modifs de schéma	N.P.
O2Views (Souza, 93)	objet	-------	classes virtuelles & imaginaires	héritage, référence	-------	OQL de O2 + définition des classes virt. & imagi.	application graphique O2
CloSQL (Monk, 93)	objet	conversion dynamique	versions de classes	héritage	contraintes de valeurs	fonctions de modification de classe	interface graphique spécifique
(Roddick, 93)	entité/relation & relationnel	-------	versions de schéma	héritage, association	cardinalités	N.P.	-------

------- : non évoqué dans la littérature N.P. : évoqué, mais non précisé dans la littérature

Figure I-8a : Synthèse des systèmes représentatifs des différentes approches

Système	*Modèle*	*Evolution des instances*	*Evolution de schéma*	*Liens sémantiques*	*Contraintes*	*Langage*	*Interface*
(Sciore, 91)	objet	versions d'attributs (annotations)	-------	héritage, composition, référence	-------	langage fonctionnel spécifique	-------
Version Server (Katz, 90a)	objet	versions, alternatives	-------	héritage, composition	contraintes intra & extra configurations + propagation	N.P.	-------
OVM (Käfer, 92)	objets complexes	versions partielles d'objets, alternatives	-------	héritage, composition	respect des cardinalités dans une configuration	type SQL objets & configurations	-------
Ode (Agrawal, 91)	objet	versions, alternatives	-------	héritage, référence	contraintes pour les objets (SGBD Actif) N.P. pour les versions	langage de programmation objet O++	OdeView : interface graphique pour les objets
O^2XER (Dijkstra, 93)	entité/relation	1 seule version pour les entités complexes	-------	héritage, aggrégations, associations	cardinalités (1 seule version/entité complexe) pas de propagation	-------	-------
O2 (O2, 96)	objet	au travers de versions de configurations	-------	héritage, référence	pas dans version commerciale	OQL + méthodes spécifiques aux configurations	fonctions O2 look
VBD (Cellary, 90)	objet	au travers de versions de bases de données (VBD)	au travers des VBD (Cellary, 91)	héritage, composition	intra & extra VBD	langage de définition	Modesty : gestion de multi-graphes de VBD

------- : non évoqué dans la littérature N.P. : évoqué, mais non précisé dans la littérature

Figure I-8b : Synthèse des systèmes représentatifs des différentes approches

Système	*Modèle*	*Evolution des instances*	*Evolution de schéma*	*Liens sémantiques*	*Contraintes*	*Langage*	*Interface*
Charly (Palisser, 90)	objet	versions, variantes	évolution de schéma	composition	-------	liste de fonctions spécifiques	interface textuelle (questionnaires)
SHOOD (Nguyen, 93)	objet	boucle de dégradation de versions + lien est-version-de	évolution de schéma	héritage, composition, dépendance, représentation	inhérentes aux liens & contraintes d'évolution (SGBD Actif)	-------	interface graphique spécifique
ORION (Kim, 89)	objet	versions, alternatives	évolution & versions de schéma	héritage, composition	inhérentes à la composition & cardinalités	interrogation SQL objet sans version	-------
Presage (Talens, 94)	objet	versions, alternatives	versions de classes	héritage, composition	inhérentes à la composition avec propagation	-------	-------

------- : non évoqué dans la littérature

Figure I-8c : Synthèse des systèmes représentatifs des différentes approches

CHAPITRE II.

Un modèle conceptuel intégrant les versions

Chapitre II. : Un modèle conceptuel intégrant les versions

Table des matières

1. Orientations

Ce chapitre décrit le modèle objet, intégrant le concept de version, que nous proposons pour décrire précisément une base de données intégrant des versions. Nous cherchons à décrire de l'évolution de chaque entité représentée dans la base, de manière indépendante. D'autres travaux s'intéressent à la représentation de l'évolution globale d'une base de données (Gançarski, 94a).

Actuellement, des systèmes tels que OVM(Käfer, 82) ou Presage(Talens, 93) offrent des fonctionnalités pour intégrer la gestion de versions lors de l'implantation d'une base de données. Néanmoins, la description d'une base suivant ce type de modèle d'implantation est relativement complexe.

Notre modèle conceptuel permet décrire simplement et précisément l'évolution d'entités complexes, telles que des documents, à l'aide de versions. L'évolution de chaque entité, est décrite de manière indépendante. Il définit pour cela des relations de composition et d'association entre versions ainsi qu'entre versions et objets. La sémantique des relations est affinée au travers de cardinalités. Les spécificités des versions doivent être prises en compte pour exprimer, sur les versions et les objets, les contraintes sous-jacentes aux relations. De plus, les opérations agissant sur les instances doivent respecter ces contraintes pour maintenir l'intégrité de la base. De plus, notre modèle permet la gestion simultanée des versions au niveau des instances et des classes. Il permet ainsi de décrire à la fois l'évolution de valeur et l'évolution de schéma. La gestion de versions de classes a également un impact sur les relations de composition, d'association et d'héritage qu'il est nécessaire d'étudier.

Notre étude s'appuie sur les principaux concepts liés à la gestion de versions proposés au niveau système pour la gestion de versions d'objets et la gestion de versions de classes. Nous intégrons ces concepts au modèle de la méthode OMT pour l'étendre à la gestion de versions ; ce modèle possède les principales caractéristiques des modèles de méthodes de conception orientées objet.

2. Notre modèle de versions

Le modèle de versions, que nous avons défini, est basé sur la notion de "relation de dérivation" adoptée par les systèmes de gestion de versions d'objets. La gestion de versions peut être effectuée de manière indépendante uniquement au niveau des objets (Ben Amouzegh, 86), (Katz, 86), (Käfer, 92), ou uniquement au niveau des classes (Monk, 94) ; elle peut également être simultanée aux deux niveaux comme dans le système Presage (Talens, 93). Pour permettre la description précise du monde réel et de son évolution, nous choisissons de pouvoir appliquer uniformément la gestion de versions au niveau des objets et au niveau des classes.

2.1. Les versions d'objets

Avec la gestion de versions d'objets, une entité du monde réel est décrite par un ensemble d'objets appelés **versions** (ou **versions d'objets**). Une version décrit un état de l'entité pendant une période de son cycle de vie.

Une version est définie par **dérivation** d'une version existante, dont elle reprend les valeurs avec éventuellement des modifications ; la nouvelle version est ainsi liée à la précédente par un lien **est dérivée de** (ou **est version de**). Seules les versions initiales (ou **versions racines**) sont définies complètement par création, les suivantes sont définies par dérivation.

Le concept d'**alternative** est également introduit. Deux versions sont des alternatives l'une par rapport à l'autre si elles sont dérivées de la même version (ex. dans la figure II-1, E1.v3 et E1.v5 sont des alternatives car elles sont dérivées de la même version E1.v2). Les alternatives permettent par exemple de proposer plusieurs solutions pour une documentation, conçues par différents auteurs.

Le concept d'alternative est étendu aux versions racines. Deux versions racines définies par création et décrivant la même entité du monde réel sont des alternatives appelées **alternatives de premier niveau** (ex. dans la figure II-1, E1.v0 et E1.v1 sont des alternatives de premier niveau décrivant l'entité Entité1). La notion d'alternative de premier niveau est par exemple utile dans un processus de conception d'un avion pour distinguer, dès le départ, plusieurs études correspondant à différents budgets.

Les versions décrivant une entité, liées par la relation de dérivation, forment une forêt de dérivation (ou hiérarchie de dérivation) c'est-à-dire un ensemble d'arbres de dérivation de versions (ex. dans la figure II-1, l'entité Entité1 est décrite par une forêt de dérivation constituée de 2 arbres).

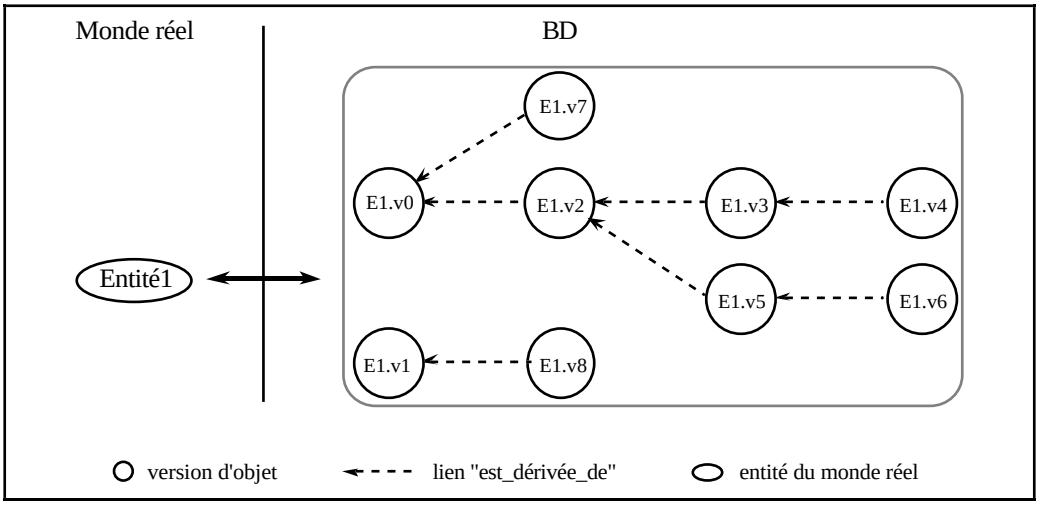


Figure II-1 : forêt de dérivation de versions

Nous considérons deux catégories de versions :

- les **versions gelées** (ou définitives) : elles décrivent un état d'une entité du monde réel à un instant donné. Elles ne peuvent pas être modifiées. Une version est "gelée" explicitement lorsque l'utilisateur applique l'opération de **gel** ou implicitement lorsqu'il en dérive une nouvelle version.

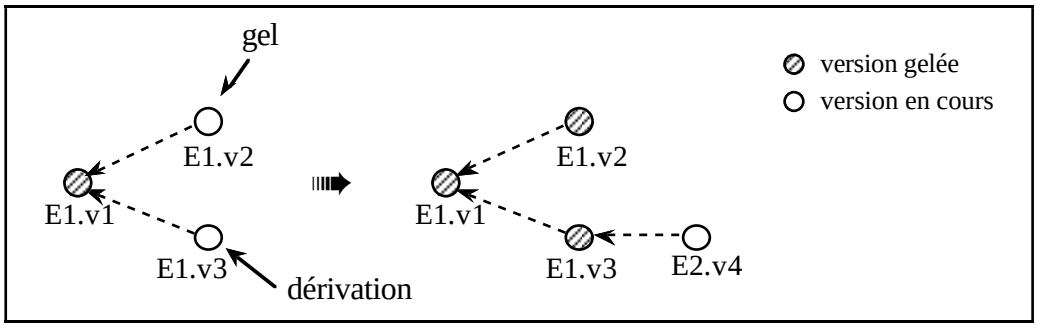


Figure II-2 : Versions gelées - Versions en cours

Les versions gelées peuvent être supprimées notamment pour réduire le nombre de versions conservées lorsque certaines d'entre elles ne sont plus utiles. Par exemple, de nombreuses versions peuvent être conservées durant l'élaboration d'une documentation pour pouvoir revenir facilement en arrière ; une fois la documentation établie, le concepteur peut décider de conserver uniquement les versions significatives.

- les **versions en cours** (ou provisoires) : elles décrivent l'état le plus récent d'une entité. Elles peuvent être modifiées au fur et à mesure que l'état de l'entité évolue ; elles peuvent également être supprimées. Toute nouvelle version est a priori en cours. Une version gelée qui n'a pas été dérivée, appelée également **version feuille** dans la forêt de dérivation de versions, peut redevenir provisoire si l'utilisateur lui applique l'opération de **dégel**.

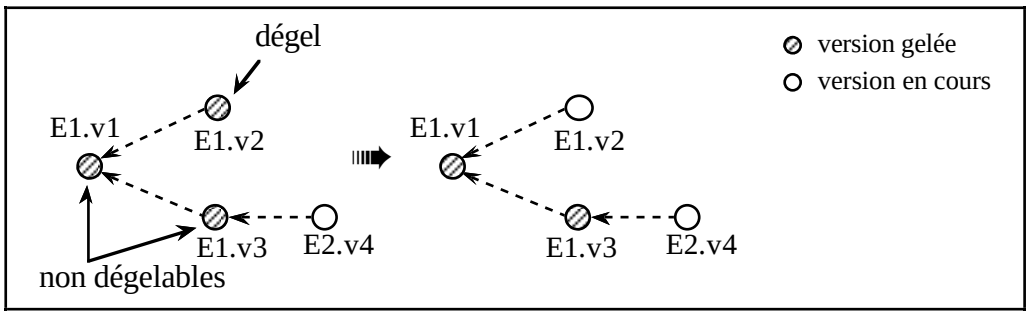


Figure II-3 : Dégel d'une version feuille

D'autres catégories de versions, qui sont des sous-ensembles de celles que nous définissons, ont été distinguées notamment pour une gestion concurrente multi-utilisateurs et multi-bases dans des systèmes tels que Version Server (Katz, 86) et ORION (Kim, 89b).

De plus, une version "par défaut" est distinguée pour chaque hiérarchie de versions décrivant une entité du monde réel ; elle est désignée par l'utilisateur comme la plus représentative de l'entité ou suivant des critères pré-définis. Cette notion s'inspire de celle de dernière version validée (LAST-VERSION) dans (Ben Amouzegh, 86) ou de DEFAULT VERSION dans (Chou, 88).

2.2. Les versions de classes

L'évolution du schéma d'une classe est décrite par un ensemble de classes appelées **versions de classe**. Les principes définis pour les versions d'objets s'appliquent aux versions de classes. Une version de classe dérivée d'une version existante reprend son schéma avec éventuellement des modifications au niveau des attributs, des méthodes et des relations de composition, d'association et d'héritage. Les **versions alternatives de classe** permettent par exemple d'avoir plusieurs schémas en parallèle pour des groupes d'utilisateurs différents. Les versions d'une classe, liées par la relation de dérivation, forment une hiérarchie de dérivation.

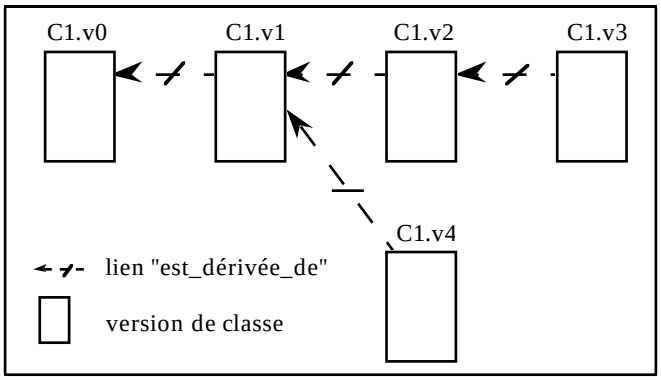


Figure II-4 : Versions de classes

Comme pour les objets, les versions de classes en cours peuvent être modifiées et supprimées tandis que les versions gelées peuvent seulement être supprimées. Les passages de l'état en cours à l'état gelé et inversement suivent les mêmes principes que pour les objets (cf. § 2.1.). De plus, dès qu'une version d'objet est gelée, la version de classe à laquelle elle appartient est également gelée.

2.3. Combinaisons des versions d'objets et de classes

2.3.1. Gestion simultanée de versions de classes et d'objets

Lorsque l'on combine la gestion de versions de classes avec la gestion de versions d'objets, les versions d'objets décrivant une entité peuvent appartenir à différentes versions d'une classe. Les dérivations de versions d'objets peuvent s'effectuer dans la même version de classe ou dans des versions de classe dérivées. La nouvelle version d'objet définie dans une version de classe dérivée reprend les valeurs de la

version d'objet précédente, compatibles avec le schéma de la version de classe dérivée. La hiérarchie de dérivation de versions d'objets représentant une entité peut s'étendre sur la totalité de la hiérarchie de dérivation de versions de classes correspondant à la classe d'entités.

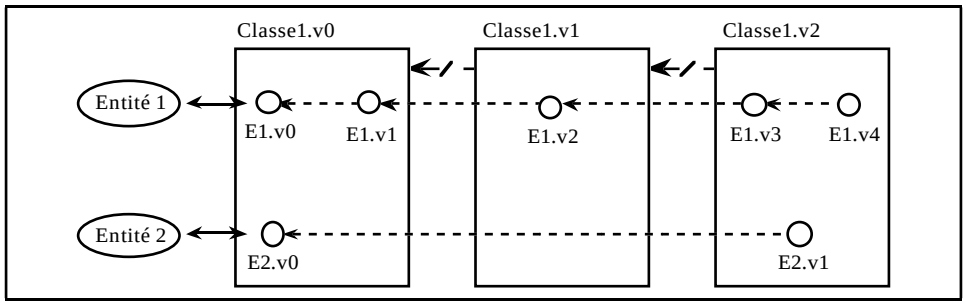
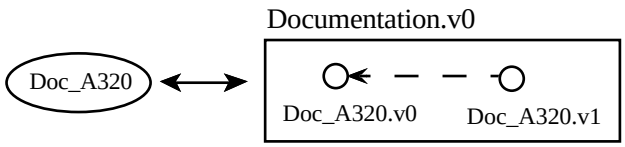
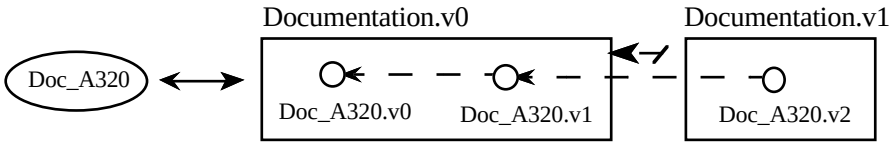


Figure II-5 : Arbre de dérivation de versions d'objets sur plusieurs versions de classes

Exemple II-1 : La gestion combinée de versions d'objets et de versions de classes est par exemple applicable dans le cas de documentations d'avions. Les avions construits sont en général des évolutions d'un avion conçu, et à chacun correspond une version de la documentation. Les documentations sont conçues suivant une certaine structure donnée (ex. titre puis chapitres puis tableaux). On définit une première version de classe Documentation.v0 dont le schéma décrit la structure des documentations. Pour une documentation, par exemple celle de l'Airbus A320, les deux premières versions de la documentation (ex. Doc_320.v0 et Doc_A320.v1) sont conçues selon le schéma de la version de classe Documentation.v0.



Supposons que la structure des documentations soit complétée pour les prochains appareils construits ; les versions de documentations de l'A320 déjà définies seront néanmoins toujours utilisées. On définit alors une nouvelle version de classe Documentation.v1, dérivée de Documentation.v0, dont le schéma complète celui de la version de classe précédente. La nouvelle version de documentation de l'A320 (Doc_A320.v2), qui est une évolution de la précédente (Doc_A320.v1), est basée sur celle-ci mais est définie dans la nouvelle version de classe.



Lorsqu'une nouvelle version d'une classe est définie, les versions en cours peuvent être utilisées avec le nouveau schéma sans les conserver dans l'ancien schéma. Pour cela, l'utilisateur fait "migrer" une version en cours vers la nouvelle

version de classe. La version d'objet est "adaptée" au schéma de la nouvelle version de classe c'est-à-dire qu'il y a conservation de toutes les valeurs de l'ancien schéma valides dans le nouveau. Nous nous limitons dans un premier temps à des cas d'adaptation simple ; des études ont été réalisées pour des modifications de schéma plus complexes (Penney, 87), (Tan, 89), (Lerner, 90), (Bounaas, 95).

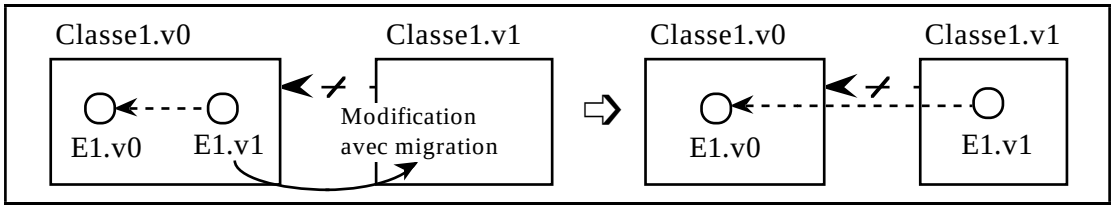


Figure II-6 : Migration d'une version provisoire vers une version de classe dérivée

2.3.2. Gestion de versions d'objets seule

La gestion de versions d'objets sans gestion de versions de classes permet de gérer des versions d'objets de même schéma. La totalité de l'arbre de dérivation de versions d'objets décrivant une entité appartient à une seule classe.

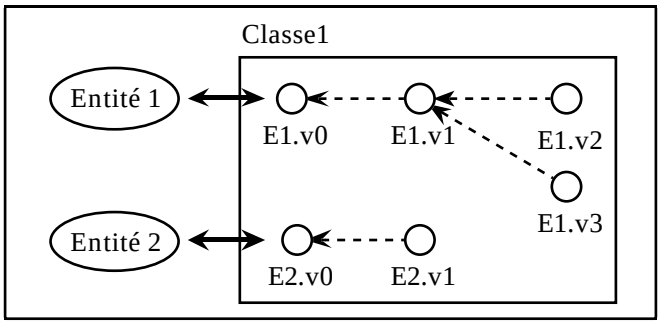


Figure II-7 : Versions d'objets sans versions de classes

2.3.3. Gestion de versions de classes seule

Un objet est créé dans une version de classe donnée. Lorsqu'il y a évolution de schéma via la dérivation d'une nouvelle version de classe, l'objet évolue en termes de schéma en "migrant" vers la nouvelle version de classe (cf. § 2.3.1.).

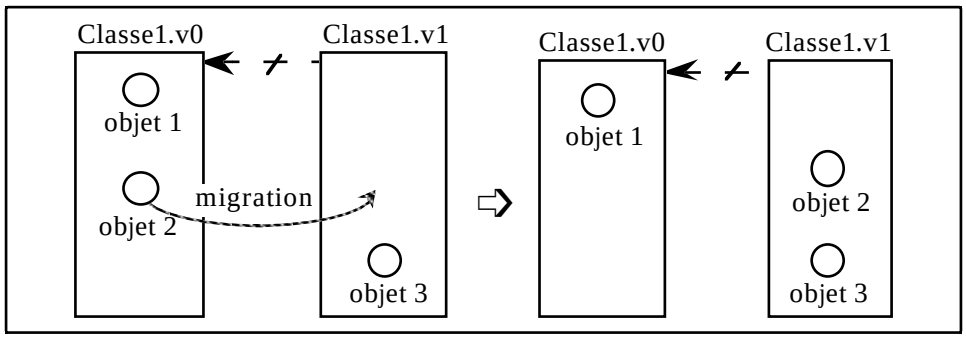


Figure II-8 : Migration d'objet

La gestion de versions de classes sans gestion de versions d'objets peut être utile notamment pour gérer l'incomplétude des données dans un processus de conception ; ce principe rejoint des travaux réalisés dans le cadre du projet SHOOD (Escamilla, 90).

Exemple II-2 : Un processus de conception de pièces d'avions est décomposé en différentes phases réalisées par des équipes de travail différentes. Pour chaque phase un ensemble d'informations sont demandées. Le passage d'une phase à la suivante n'est autorisé que lorsque toutes les informations demandées sont spécifiées.

Les différentes phases de conception peuvent être modélisées par des versions de classes ; le schéma de chaque version de classe précise les informations correspondant à la phase qu'elle décrit. Chaque version de classe ajoute des propriétés par rapport à la version précédente. Chaque pièce d'avion à concevoir est représentée par un objet. L'objet est défini dans la première version de classe (décrivant la première phase de conception). Lorsque tous les attributs sont valorisés (phase terminée), le concepteur fait migrer l'objet dans la version de classe suivante (pour entamer la phase suivante) ; de nouveaux attributs sont alors à valoriser. La version de classe finale regroupe les objets dont la conception est achevée.

3. Le modèle sémantique de données intégrant les versions

Le modèle sémantique de données que nous proposons repose sur les principaux concepts adoptés dans la plupart des modèles de données des méthodes de conception et les étend pour prendre en compte la gestion de versions d'objets et de versions de classes. Notre modèle permet de modéliser des entités pour lesquelles on gère des versions au niveau des valeurs et des schémas. Le modèle permet la modélisation d'entités complexes au travers de la définition de relations de composition et d'association entre les différents types de classes du modèle. Nous étudions en détail comment est traduite au niveau des instances, objets ou versions d'objets, la sémantique des relations entre classes ; nous exprimons les contraintes sur les instances, inhérentes à la définition des relations entre classes ainsi que les règles permettant leur respect par les opérations sur les instances et sur les classes.

3.1. Les concepts du modèle objet OMT

Notre modèle de données s'appuie sur le modèle objet de la méthode OMT (Rumbaugh, 91) dont nous reprenons le formalisme ; il est enrichi au niveau de la sémantique de la composition. Nous avons choisi OMT car son modèle comprend tous les concepts habituellement présents dans les modèles sémantiques et qu'il est utilisé dans l'industrie.

3.1.1. Les classes

Les classes regroupent les instances ayant les mêmes propriétés (structure et comportement). Une classe décrit à la fois l'ensemble des instances qu'elle regroupe, appelé l'**Extension** de la classe, et leur **Schéma**. Le schéma spécifie la structure et le comportement des instances ainsi que la place de la classe dans la hiérarchie d'héritage.

La structure est représentée par un ensemble d'attributs définis sur des domaines de base (entier, réel, ...) et par un ensemble de relations de composition et d'association avec d'autres classes. Les attributs peuvent être mono-valués ou multi-valués.

Le comportement des instances est représenté par un ensemble de méthodes. Une méthode est décrite par une signature et un corps. Les méthodes sont de deux types : pré-définies ou définies par l'utilisateur.

3.1.2. L'héritage

La relation d'héritage permet de factoriser les propriétés de structure, de comportement communes à plusieurs classes, appelées **sous-classes**, en une classe de plus haut niveau hiérarchique, appelée **super-classe**. L'héritage est mécanisme permettant le transfert des propriétés d'une super-classe vers ses sous-classes.

L'héritage que nous retenons est un héritage par spécialisation (Atkinson, 89). La spécialisation impose que le schéma d'une super-classe soit inclus dans celui de la sous-classe ; la spécialisation consiste à définir le schéma d'une sous-classe par enrichissement (ajout d'attributs et/ou de méthodes) ou par affinement (substitution du domaine de définition d'un ou plusieurs attributs par un sous-domaine) du schéma d'une super-classe.

Le graphe représentatif de la relation d'héritage liant sous-classes et super-classes est appelé **graphe d'héritage**. Ce graphe est acyclique, il décrit un treillis de classes. Le modèle supporte l'héritage multiple. Une sous-classe peut donc hériter des propriétés de plusieurs super-classes.

3.1.3. Les relations entre classes

Les relations entre classes permettent de représenter des entités complexes. Les relations entre classes sont de deux sortes : les compositions et les associations.

D'une part, la sémantique de la composition n'est pas clairement précisée dans OMT ; pour pallier ce manque, nous précisons la sémantique que nous choisissons

qui est celle habituellement considérée dans les modèles de méthodes de conception orientée objet (Giraudin, 95). La composition exprime une relation "est-une-partie-de" entre un composant appartenant à une classe composante, et un composé appartenant à une classe composée. La sémantique associée à la composition indique :

- qu'un composant ne peut exister sans faire partie d'un composé,
- que l'existence du composant dépend de celle des composés auxquels il est lié ; la disparition des composés implique celle de leurs composants,
- que la création d'un composant n'est effectuée qu'au travers d'un composé.

Les liens de composition peuvent être de deux catégories dans notre modèle :

- les **liens exclusifs** : un composant n'est une partie que d'un seul composé,
- les **liens partagés** : un composant peut faire partie de plusieurs composés.

L'exclusivité ou le partage sont spécifiés au niveau de la classe composante par une cardinalité associée à la classe (cardinalité 1-β, avec β=1 pour exprimer l'exclusivité). De plus, une cardinalité est définie sur le lien pour la classe composée indiquant les nombres minimum et maximum de composants autorisés pour un composé.

Notre définition des liens de composition rejoint celle des références composites dépendantes du modèle du système ORION (Kim, 89b).

D'autre part, une relation d'association exprime le fait que deux instances de deux classes sont liées. La disparition d'une instance n'implique pas celle des instances auxquelles elle est liée.

Une cardinalité est fixée pour chacune des classes liées ; elle indique les nombres minimum et maximum d'instances de l'autre classe auxquelles une instance d'une classe peut être liée.

La sémantique des relations de composition et d'association, ainsi que les cardinalités associées sont traduites au niveau des instances des classes liées par différentes contraintes sur les liaisons possibles entre instances. Ces contraintes sont des contraintes d'intégrité structurelle (Amghar, 94), (Bouaziz, 95). Elles doivent être respectées notamment lors des opérations de création d'instances, suppression, ... L'intégration du concept de version rend encore plus complexe l'expression de ces contraintes sur les instances.

3.2. Les nouveaux types de classes dus aux versions

Pour décrire le monde réel, il est nécessaire que notre modèle conceptuel permette :

- d'une part, de modéliser les entités pour lesquelles on veut conserver des versions au niveau des valeurs, et à l'opposé, de modéliser celles pour lesquelles on ne veut conserver qu'une seule valeur,
- de plus, de modéliser les entités pour lesquelles on conserve différentes versions du schéma et modéliser celles pour lesquelles un seul schéma est conservé.

Pour modéliser les versions au niveau des valeurs, nous distinguons deux catégories d'instances :

- **les objets** : ce sont les instances de tout modèle objet. Les objets ayant les mêmes propriétés (structure et comportement) sont regroupés en classes. Un objet est identifié par un identifiant unique indépendant de sa valeur (Khosafian, 86).

- **les versions d'objets** : les versions (d'objets) sont des objets liés par des liens de dérivation qui traduisent le fait qu'elles décrivent l'évolution de la même entité du monde réel. Les versions d'objets comme les objets sont regroupées en classes qui décrivent leur structure (à l'aide d'attributs et de relations) et leur comportement (à l'aide de méthodes). Dans une classe dont les instances sont des versions, différents ensembles de versions (ou hiérarchies de dérivation) représentent différentes entités du monde réel. Une version d'objet est identifiée par un identifiant unique pour l'entité modélisée et par un numéro de version unique dans l'ensemble des versions décrivant l'entité (Rieu, 85), (Ben Amouzegh, 86).

  Toutes les versions, quelle que soit la classe à laquelle elles appartiennent, possèdent plusieurs critères, dits "internes", systématiquement ajoutés en plus de ceux définis par le concepteur. Ces critères internes sont les suivants : la date de création de la version, son numéro, le nom de son créateur, l'état de la version.

Pour décrire les versions au niveau du schéma, nous introduisons le concept de **version de classe**. Notre modèle distingue deux catégories de classes :

- **les classes simples ou non-versionnalisables** : ce sont les classes des modèles orientés objet ; les classes simples ne permettent pas la gestion de versions de classes c'est-à-dire de conserver l'évolution de schéma. Le schéma d'une classe simple peut néanmoins évoluer par modification ; le schéma modifié

remplace alors le schéma avant modification. Les classes simples peuvent avoir pour instances des objets ou des versions d'objets.

Pour les classes simples d'objets, lors d'une modification du schéma, les valeurs des objets dans l'ancien schéma sont conservées si elles sont valides dans le nouveau schéma. Différentes solutions sont proposées dans la littérature pour gérer la conservation de valeurs lors de modifications de schéma au niveau système (Chou, 88), (Lerner, 90), (Bounaas, 95).

Pour les classes de versions, la conservation de valeurs compatibles avec le nouveau schéma s'applique lorsqu'il n'y a que des versions provisoires comme instances, car elles sont modifiables. La modification du schéma de la classe n'est plus possible dès qu'une des versions instances de la classe est gelée (par définition non modifiable tant au niveau valeur que schéma).

- **les versions de classe** : elles décrivent différentes évolutions du schéma d'une classe. Les versions d'une classe sont des classes simples liées par des liens de dérivation ; elles forment une hiérarchie de dérivation de classes. Les versions de classes peuvent également avoir pour instances des objets ou des versions d'objets.

Toutes les combinaisons sont possibles entre les catégories de classes et d'instances.

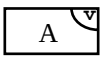
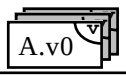
schéma / instances	classe simple	versions de classe
objet	pas de gestion de version disponible, ni au niveau du schéma ni au niveau des instances. formalisme graphique : 	gestion de version disponible au niveau du schéma mais pas au niveau des instances. (1) formalisme graphique : 
versions d'objets	gestion de version disponible au niveau des instances mais pas au niveau du schéma. formalisme graphique : 	gestion de version disponible au niveau du schéma et au niveau des instances. (2) formalisme graphique : 

Figure II-9 : Formalismes concernant les quatre types de classes

(1) Chaque entité du monde réel est représentée par un objet unique appartenant à l'une des versions de la classe (cf. § 2.3.3.).

(2) Chaque entité est représentée par un ensemble de versions d'objets, chaque version appartenant à l'une des versions de classe ; une version dérivée appartient à la même version de classe que la version dont elle est issue ou une version de classe dérivée (cf. § 2.3.1.).

3.3. Les opérations

3.3.1. Les opérations sur les objets et les versions d'objets

Au niveau des instances, il existe un ensemble d'opérations de base ; le concepteur d'une base définit dans le schéma d'une classe, les opérations spécifiques aux instances de cette classe (méthodes).

Les opérations de base sont la création, la suppression et la modification qui s'appliquent aux objets et aux versions d'objets. Pour les versions, la création concerne uniquement la définition des versions qui seront des racines d'arbres de dérivation (les autres versions sont définies par dérivation). La modification de version ne s'applique qu'à des versions provisoires. Certaines opérations liées aux versions d'objets sont ajoutées :

- l'opération de dérivation qui définit une version à partir d'une version existante en reprenant ses valeurs ; la version dérivée est liée à la version dont elle est issue par un lien **est_dérivée_de**. La dérivation s'applique à toutes les versions. Lorsqu'il y a des versions de classes, la dérivation de versions d'objets peut s'effectuer d'une version de classe dans la même version de classe ou vers une version de classe dérivée.
- les opérations de gel et de dégel qui transforment une version en cours en version gelée et inversement. Le dégel ne s'applique qu'à des **versions feuilles** c'est-à-dire qui n'ont pas été dérivées. Une version est également gelée, implicitement, lorsqu'une version en est dérivée.

Pour les instances de versions de classes, l'opération de migration est ajoutée. La migration consiste à déplacer une instance d'une version d'une classe vers une autre ; les valeurs compatibles avec le nouveau schéma sont conservées comme lors d'une modification de schéma. La migration n'est autorisée que d'une version de classe vers une version de classe dérivée, c'est-à-dire en suivant le sens d'évolution de schéma. Elle s'applique aux objets et aux versions en cours (non gelée).

3.3.2. Les opérations sur les classes

Au niveau des classes, les modifications possibles sont les principales modifications établies dans la littérature (cf. chapitre I. § 3.1.1.) incluant des modifications liées aux relations entre classes. Les modifications interviennent via l'opération de modification ou de dérivation d'une classe. On distingue les modifications du schéma d'une classe et les modifications du graphe d'héritage.

Les modifications du schéma d'une classe regroupent l'ajout, la suppression et la modification d'attributs, de méthodes et de relations entre classes.

La modification d'un attribut consiste à renommer un attribut, restreindre son domaine de définition ou au contraire l'élargir ; lors de la restriction de domaine, si la valeur de l'attribut n'est plus valide dans le domaine, elle est réinitialisée.

L'ajout de relations de composition ou d'association, lorsqu'il existe des instances, est limitée aux relations autorisant les valeurs nulles c'est-à-dire ayant des cardinalités minimum de 0 ; les liens pour les instances existantes sont initialisés à vide. La modification d'une relation consiste à renommer une relation, à élargir les cardinalités ou à passer de l'exclusivité au partage pour les relations de composition.

Les modifications du graphe d'héritage consistent à ajouter, supprimer ou renommer une classe, ajouter une classe comme super-classe d'une autre (aucun cycle ne doit apparaître), ou supprimer une classe comme super-classe d'une autre. Les mêmes restrictions sont apportées concernant les relations de composition et d'association ajoutées au travers d'une super-classe.

3.4. Héritage et gestion de versions

Une relation d'héritage peut être spécifiée entre toutes les catégories de classes. La gestion de versions de classes influe sur l'héritage lors de la dérivation de versions de super-classes et de sous-classes.

3.4.1. Les cas d'héritage

Les classes de notre modèle peuvent avoir deux catégories d'instances : les objets et les versions d'objets qui ont des comportements différents. De plus, les classes peuvent appartenir à deux catégories différentes : les classes simples et les versions de classes. Les classes simples sont dites "plus restrictives" en termes d'évolutivité que les versions de classes puisqu'elles ne permettent pas la gestion de versions au niveau du schéma.

Compte tenu de la définition de l'héritage, deux règles permettent d'obtenir une hiérarchie d'héritage cohérente :

Règle 1: les instances des sous-classes et de leurs super-classes sont de même catégorie (objets ou versions d'objets).

Règle 2: la catégorie d'une super-classe ne peut être moins restrictive (en termes d'évolutivité) que les catégories de ses sous-classes. Cette règle interdit d'avoir des versions d'une super-classe lorsque l'une de ses sous-classes ne l'est pas.

3.4.2. Héritage et versions de classes

En considérant les possibilités d'évolutivité des classes liées par un lien d'héritage, la dérivation d'une sous-classe ou d'une super-classe entraîne différentes conséquences sur la relation d'héritage selon les cas de figures (Hubert, 95a).

Lorsqu'il y a des versions de super-classes et de sous-classes, la dérivation peut s'appliquer à une super-classe ou à une sous-classe.

- ***Dérivation d'une super-classe***

La dérivation d'une super-classe n'implique pas la dérivation de ses sous-classes ; l'évolution des sous-classes est réalisée au niveau des sous-classes. Les sous-classes dans ce cas héritent toujours de l'ancienne version de la super-classe.

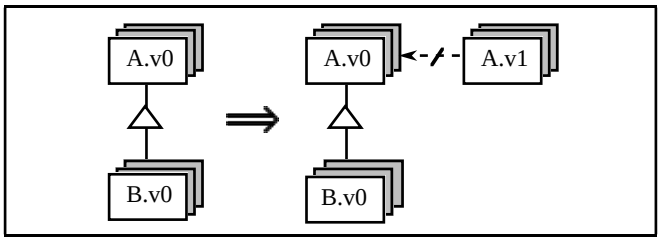


Figure II-10 : Héritage - Dérivation de super-classe

- ***Dérivation d'une sous-classe***

Pour la dérivation d'une sous-classe, plusieurs cas de figure peuvent exister selon les modifications effectuées lors de la dérivation :

- si la dérivation correspond à une redéfinition des attributs hérités ou à un changement au niveau des attributs spécifiques, la sous-classe dérivée hérite de la même super-classe.

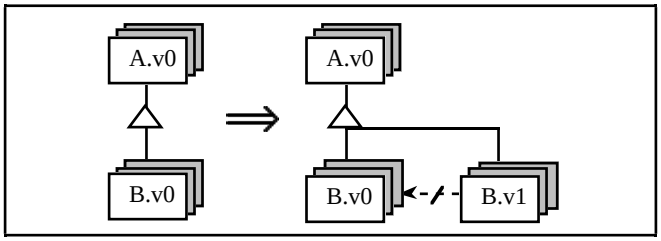


Figure II-11a : Dérivation de sous-classe -cas 1

Dans ce cas de dérivation, la sous-classe dérivée peut également hériter d'une version dérivée de la super-classe.

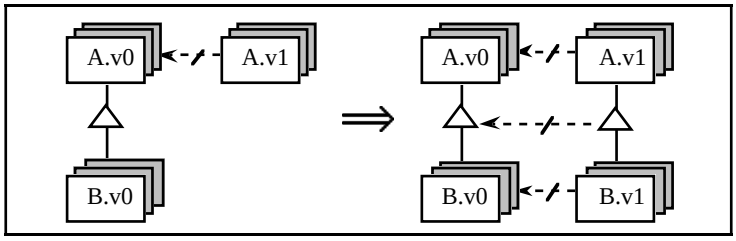


Figure II-11b : Dérivation de sous-classe -cas 2

- si la dérivation correspond à une modification des attributs hérités, la dérivation de la sous-classe entraîne la dérivation de la super-classe ; la sous-classe dérivée hérite de la version dérivée de la super-classe.

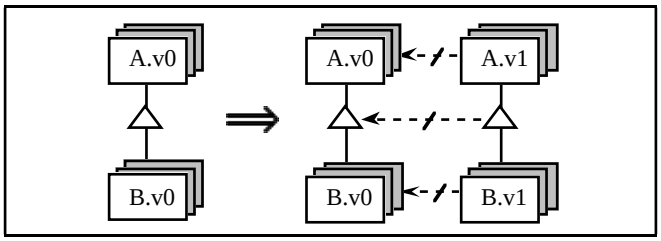


Figure II-11c : Dérivation de sous-classe -cas 3

Lorsque seule la sous-classe est dérivable, seul le premier cas évoqué est possible. De plus, dans ce cas la dérivation ne peut correspondre à une modification des attributs hérités.

3.4.3. Synthèse des conséquences des versions sur l'héritage

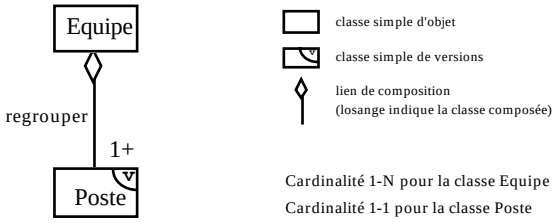
	Sous-classe		Super-classe
Instances	versions	$\Longleftrightarrow$	versions
	objets	$\Longleftrightarrow$	objets
Schéma	versions de classe	$\Longrightarrow$	(pas de conséquence)
	versions de classe	$\Longleftarrow$	versions de classe
	classe simple	$\Longrightarrow$	classes simples
	dérivation sous-classe avec modifications d'attributs hérités (impossible si super-classes simples)	$\Longrightarrow$	dérivation des super-classes qui ont les attributs modifiés
	dérivation sous-classe sans modification d'attribut hérité	$\Longrightarrow$	mêmes super-classes ou des dérivées des super-classes (si elles existent)
	(pas de conséquence)	$\Longleftarrow$	dérivation super-classe

Figure II-12 : Tableau récapitulatif des conséquences des versions sur l'héritage

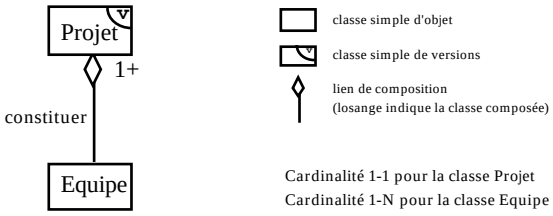
3.5. Compositions et associations entre tous types de classes

Pour fournir une puissance de modélisation maximale, notre modèle définit des compositions et des associations entre les quatre types de classes (Andonoff, 95) (Hubert, 95a). Ce modèle permet ainsi de décrire des entités composites pour lesquelles on gère des versions uniquement sur certains composants.

Exemple II-3 : Considérons que l'on veuille décrire des équipes constituées de différents postes de travail ; on désire conserver les principales évolutions des postes de travail. On définit alors un lien de composition entre une classe simple d'objets *Equipe* et une classe simple de versions *Poste* comme suit, suivant le formalisme OMT étendu aux versions :



Au contraire, si l'on veut décrire des projets composés d'équipes de travail ; on désire conserver les principales évolutions des projets mais pas des équipes. On définit alors un lien de composition entre une classe de versions *Projet* et une classe d'objets *Equipe* comme suit, suivant le formalisme OMT étendu aux versions :



Les relations de composition et d'association définies entre classes se traduisent par des contraintes sur les instances des classes ; ces contraintes dépendent du type d'instances liées, du type de relation (composition ou association) qui les lie, et des cardinalités fixées. Ces contraintes sont des contraintes d'intégrité structurelle (Amghar, 94) ; nous allons les définir pour les différents cas de composition et d'association possibles dans notre modèle.

3.6. Stratégie du partage des versions d'objets

L'expression des relations de composition et d'association, au niveau des instances des classes, doit décrire la réalité. Par exemple, un avion composé d'un moteur peut évoluer sans que le moteur ne change. En termes de versions, cela

signifie que la nouvelle version de l'avion reste composée de la même version du moteur.

Ainsi, pour décrire précisément la réalité, la traduction de la sémantique des relations entre classes doit considérer le partage des versions composantes.

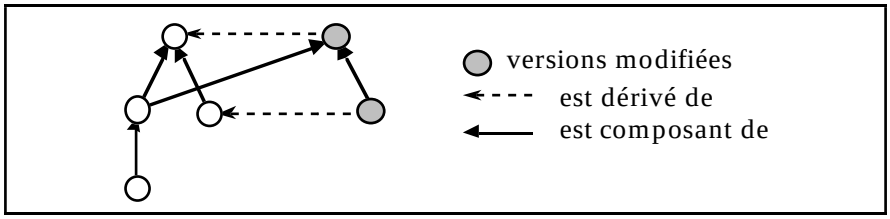


Figure II-13 : Limitation de duplication de versions

Une telle stratégie a déjà été adoptée dans des travaux effectués au sein de notre équipe (Ben Amouzegh, 86), (Comparot, 94). Dans d'autres systèmes tels que Presage (Talens, 93), les choix de propagation des dérivations de versions entraînent, dans certains cas, la dérivation de chacune des versions composantes ; les versions composantes ne sont alors plus partagées.

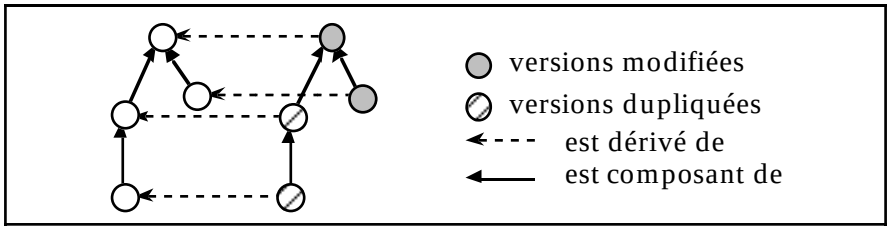


Figure II-14 : Duplication de versions

3.7. Contraintes sur les instances, inhérentes aux compositions et associations

3.7.1. Composition et association d'objets

Les contraintes sur les objets de classes liées par une composition sont définies sur les liaisons inter-objets possibles. Ce principe rejoint celui défini dans (Kim, 89a).

Par exemple, pour une composition (1-1, exclusive) entre classes simples d'objets, un objet composé n'a qu'un seul objet composant et inversement. L'exclusivité est assimilée à une cardinalité 1-1 pour la classe composante. Des principes analogues s'appliquent aux associations.

Les contraintes inhérentes aux liens doivent être respectées par les opérations agissant sur les objets liés. La composition implique par exemple que la suppression d'un objet composé entraîne la suppression de ses objets composants, excepté si ceux-ci composent d'autres objets, dans le cas d'un lien partagé. Par ailleurs les

contraintes limitent les objets auxquels il est possible d'être lié. Dans le cas d'une association 1-1, 1-1 un objet créé ou modifié dans une classe ne peut être lié qu'à un nouvel objet créé dans la même transaction dans l'autre classe.

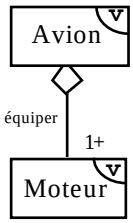
3.7.2. Composition et association de versions d'objets

A l'image des relations entre classes d'objets, les relations entre classes de versions induisent des contraintes sur les versions instances ; ces contraintes doivent être respectées par les opérations appliquées sur les versions (Andonoff, 95).

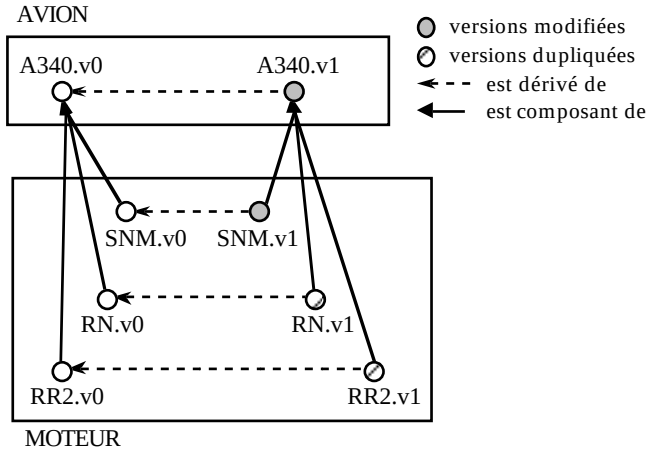
3.7.2.1. La problématique liée au partage de versions

Dans un modèle de versions d'entités comme le nôtre, il n'est pas suffisant, contrairement aux objets, de définir les contraintes liées aux cardinalités, sur les liaisons entre versions d'objets. En effet, cela oblige à multiplier les versions identiques. Si l'on considère par exemple un lien de composition entre classes de versions, la définition des contraintes sur les liaisons inter-versions pour l'exclusivité conduit à dupliquer systématiquement les versions composantes.

Exemple II-4 : Considérons le lien de composition 1-N et exclusif entre la classe de versions AVION et la classe de versions MOTEUR suivant :



Le respect de l'exclusivité du lien de composition implique, pour toute nouvelle version composante (ex. SNM.v1) dans une hiérarchie de composition, que toutes les autres versions composantes soient dupliquées (ex. RN.v1 et RR2.v1).



Pour résoudre ce problème, les contraintes liées aux cardinalités et à la notion d'exclusivité / partage doivent donc s'exprimer autrement que sur les liaisons inter-versions.

Le système ORION, basé sur un modèle de versions d'entités, propose une solution pour gérer les contraintes inhérentes à la composition entre classes simples de versions. Il introduit deux catégories d'instances pour gérer les versions d'objets :

- les versions instances qui représentent les états conservés des entités,
- les objets génériques qui décrivent les entités représentées ; un objet générique regroupe les versions instances d'une entité représentée.

Ce modèle définit les contraintes sur les liaisons inter-versions. L'exclusivité pour une classe composante implique par exemple qu'une version ne peut composer qu'une seule version composée. Ainsi, lorsque l'on dérive la version composée, celle-ci ne peut avoir la même version composante compte tenu des contraintes. Pour éviter de dupliquer la version composante, le modèle propose de rattacher la nouvelle version composée à l'objet générique correspondant à la version composante ; les contraintes définies sur les versions d'objets sont ainsi contournées.

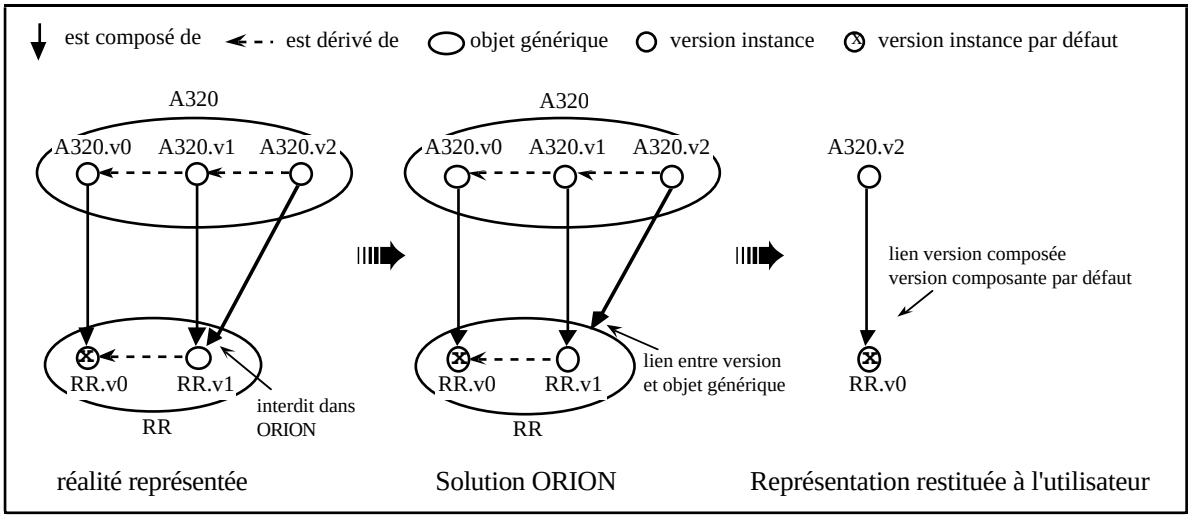


Figure II-15 : La gestion des contraintes dans le modèle ORION

Cependant, une version composée liée à un objet générique est en fait considérée comme étant dynamiquement liée à la version instance par défaut correspondante. Ainsi, les liens entre versions composées et versions composantes restituées à l'utilisateur ne sont pas celles qui ont été spécifiées. La solution proposée dans le modèle du système ORION ne permet pas de décrire fidèlement l'évolution des entités du monde réel.

3.7.2.2. Les solutions pour la composition

Dans les modèles de versions d'entités comme le nôtre, les mécanismes pour la gestion des versions d'objets composites proposés jusqu'ici ne permettent pas de décrire fidèlement l'évolution des entités complexes du monde réel. Nous proposons une solution différente pour l'expression des contraintes sur les versions d'objets, pour les relations de composition. Notre solution tient compte de la stratégie du partage de versions composantes.

3.7.2.2.1. L'expression des contraintes sur les versions d'objets

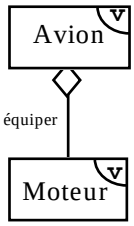
La sémantique de la composition implique que l'on retrouve la hiérarchie de composition des instances au travers des composés. De ce fait, une instance composée ne possède qu'une seule sous-hiérarchie de composants ; l'inverse conduirait à ne plus pouvoir retrouver comment se composent les instances. Pour retrouver l'évolution d'une hiérarchie de composition de versions d'objets, il est donc nécessaire de conserver chaque fois une version d'un composé pour lequel la sous-hiérarchie de versions composantes évolue.

Compte tenu de ce principe, la cardinalité fixée pour la classe composée peut être traduite par des contraintes sur les liaisons entre versions composées et versions composantes. Une cardinalité 1-1 pour une classe de versions composée implique, par exemple, qu'une version composée possède une seule version composante.

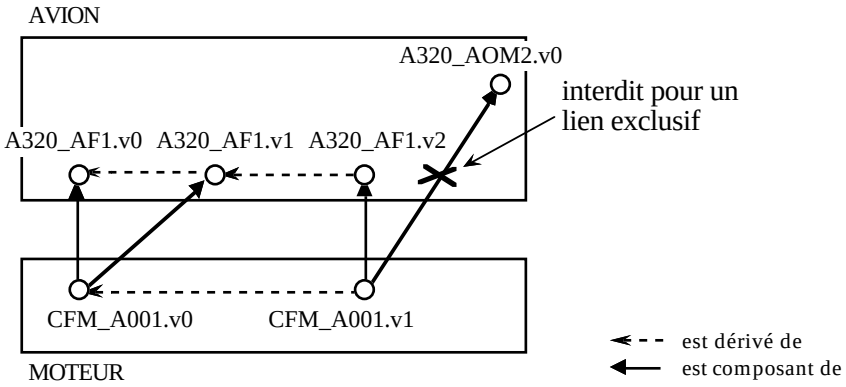
A l'inverse, la notion d'exclusivité/partage ne pouvant s'exprimer par des contraintes sur les liaisons entre versions, nous considérons l'abstraction de toutes les liaisons entre une version composante et les versions qu'elle compose et qui appartiennent à une même hiérarchie de dérivation. Les contraintes s'expriment alors sur les liaisons entre versions composantes et hiérarchies de dérivation de versions composées.

Pour un lien de composition exclusif, une version composante ne peut être liée qu'à une seule hiérarchie de dérivation de la classe composée. L'exclusivité impose que les versions composées auxquelles peut être liée une version composante, appartiennent à la même hiérarchie de dérivation et soient des dérivées successives. En revanche si le lien est partagé, une version composante peut être liée à des versions composées appartenant à des arbres de dérivation différents.

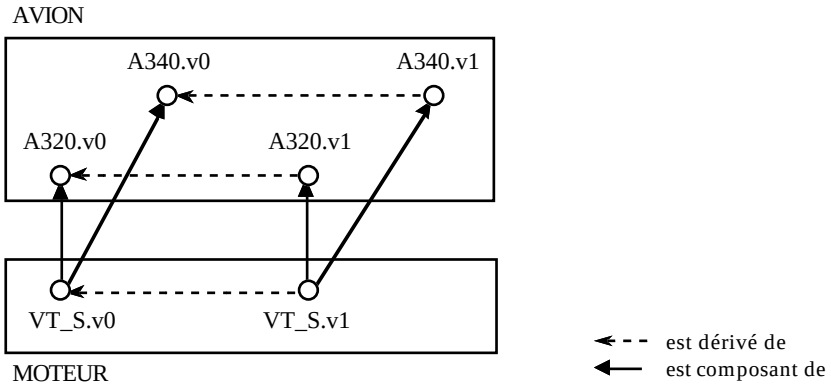
Exemple II-5 : Considérons le lien de composition (1-1, exclusif) entre la classe de versions AVION et la classe de versions MOTEUR suivant :



Si la composition est exclusive, un seul moteur (ex. CFM_A001 : le moteur SNECMA n° A001) ne peut équiper qu'un seul avion (ex. A320_AF1 : l'Airbus A320 n° 1 d'Air France). L'exclusivité impose que les différentes versions existantes décrivant le moteur CFM_A001 ne composent que des versions décrivant l'Airbus A320_AF1.



En revanche, si la composition est partagée, il s'agit plutôt de décrire qu'un type de moteur (ex. VT_S, le moteur VT2500 d'IAE de type S) peut équiper plusieurs avions ; les versions décrivant le type le moteur VT_S peuvent faire partie des versions décrivant les Airbus A320 et des versions décrivant les Airbus A340.



3.7.2.2.2. Respect des contraintes par les opérations

Les opérations agissant sur les liens entre versions d'objets sont la création, la dérivation, la suppression et la modification.

• ***Création d'une version composée***

La création correspond uniquement à la définition d'une nouvelle version racine. La définition de toute autre version correspond à la dérivation.

D'une part, la cardinalité fixée pour la classe composante impose et/ou restreint le nombre de versions composantes pour la nouvelle version composée créée. Par exemple, une cardinalité 1-1 impose et restreint le nombre de versions composantes à un.

D'autre part, la notion d'exclusivité/partage définit les versions composantes possibles pour la nouvelle version composée créée. Si le lien de composition est exclusif, les versions composantes possibles sont :

- soit de nouvelles versions racines créées en même temps que la version composée,
- soit des versions dérivées de versions feuilles "libres" de hiérarchies déjà existantes, définies en même temps que la version composée.

Une version feuille composante est dite "libre" si elle n'est liée à aucune version feuille d'une hiérarchie composée. Une version feuille libre correspond à un ancien composant d'une hiérarchie ; cette hiérarchie a par la suite changé de composant. Une nouvelle version dérivée de cette version feuille libre peut donc être un composant d'une nouvelle version composée.

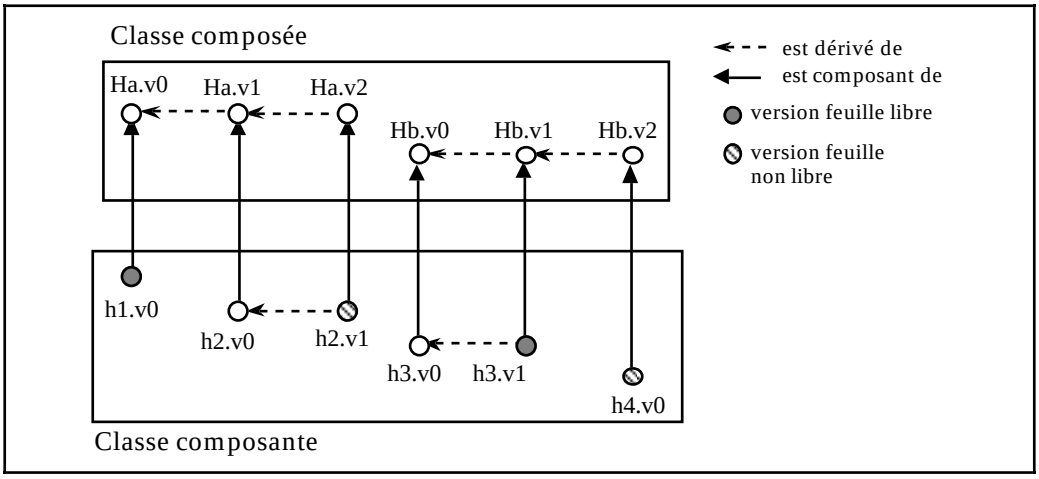
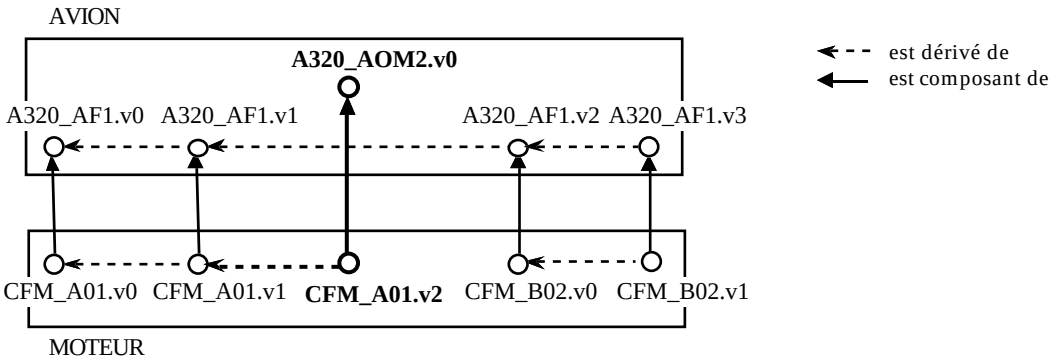


Figure II-16 : Versions feuilles libres

Si le lien de composition est partagé, les versions composantes peuvent également être des versions déjà existantes de hiérarchies de la classe composante.

Exemple II-6 : Reprenons le lien de composition exclusif entre la classe de versions composées AVION et la classe de versions composantes MOTEUR.



Le moteur SNECMA n° A01 (CFM_A01) qui était choisi initialement pour l'avion Airbus A320 n° 1 d'Air France (versions v0 et v1 de CFM_A01 liées aux 2 premières versions de A320_AF1) ne fait plus partie de l'avion (versions suivantes v2 et v3 de A320_AF1 liées à CFM_B02). Le moteur CFM_A01, qui est "libre", peut donc être choisi pour l'avion A320_AOM2 (nouvelle hiérarchie composée liée à une version dérivée de CFM_A01.v1). En revanche, le moteur CFM_B02 qui compose encore l'avion A320_AF1 (dernière version A320_AF1.v3 liée à la version CFM_B02.v1), n'est pas "libre" pour faire partie d'un autre avion.

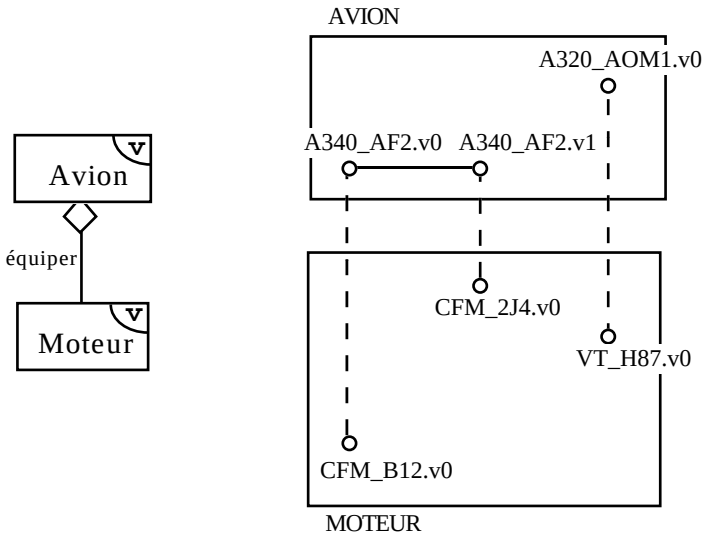
- ***Dérivation d'une version composée***

Le respect de la cardinalité fixée pour la classe composée suit le principe précédent défini pour la création d'une version composée.

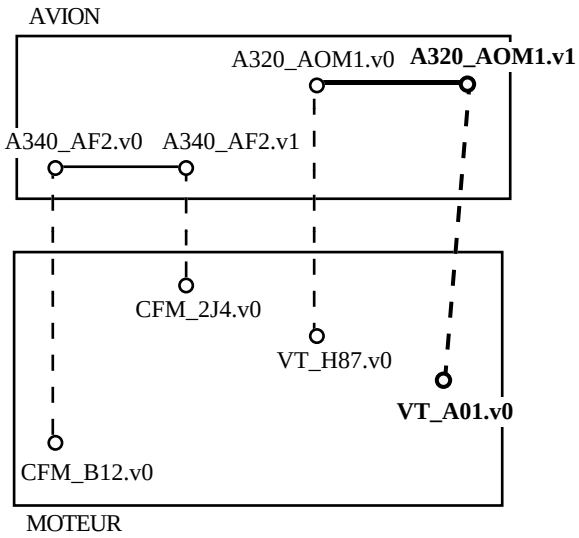
Pour le respect de la notion d'exclusivité / partage, la dérivation autorise d'autres versions composantes possibles pour la version composée créée par dérivation pour les liens de composition exclusifs ; les versions composantes qui peuvent être liées à la nouvelle version composée sont :

- soit les même types de versions composantes que pour la création d'une version composée (cf. § précédent) c'est-à-dire de nouvelles versions racines ou des versions dérivées de versions composantes feuilles "libres",
- soit les mêmes versions composantes que la version composée dont la nouvelle version dérive,
- soit des versions dérivées des versions composantes de la version composée dont la nouvelle version dérive.

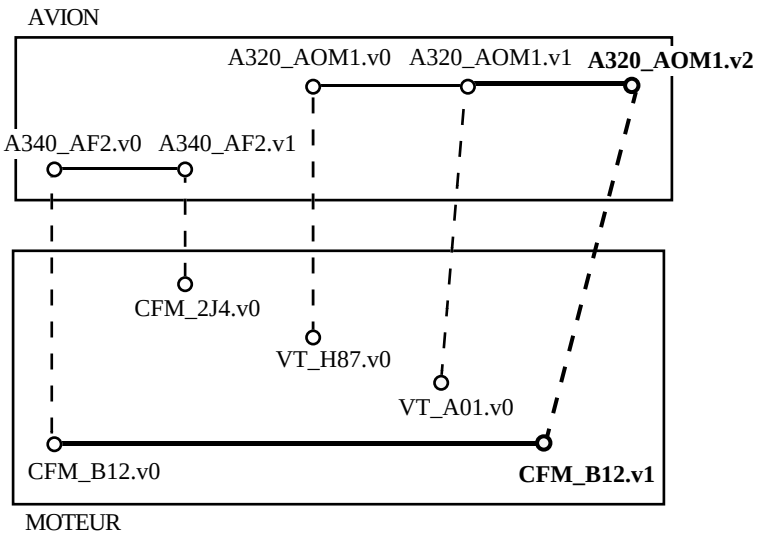
Exemple II-7 : Considérons le lien de composition (1-1, exclusif) entre les classes de versions AVION et MOTEUR. Le lien traduit le fait qu'un avion ne peut être composé à un instant donné que d'un seul moteur et qu'inversement un moteur ne peut composer qu'un avion. Prenons le point de départ suivant au niveau des instances :



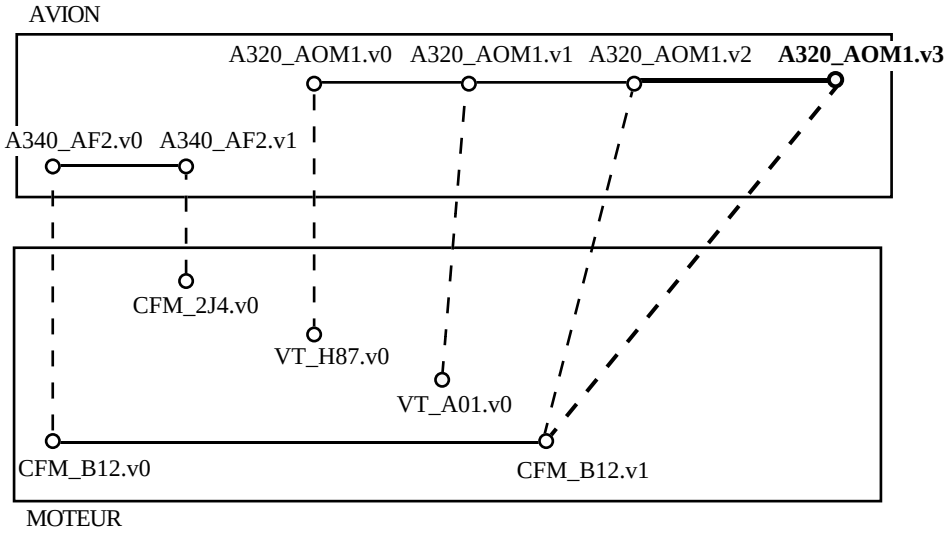
Les différents cas de rattachement d'une version composée à une version composante lors d'une dérivation de version composée vont être illustrés.



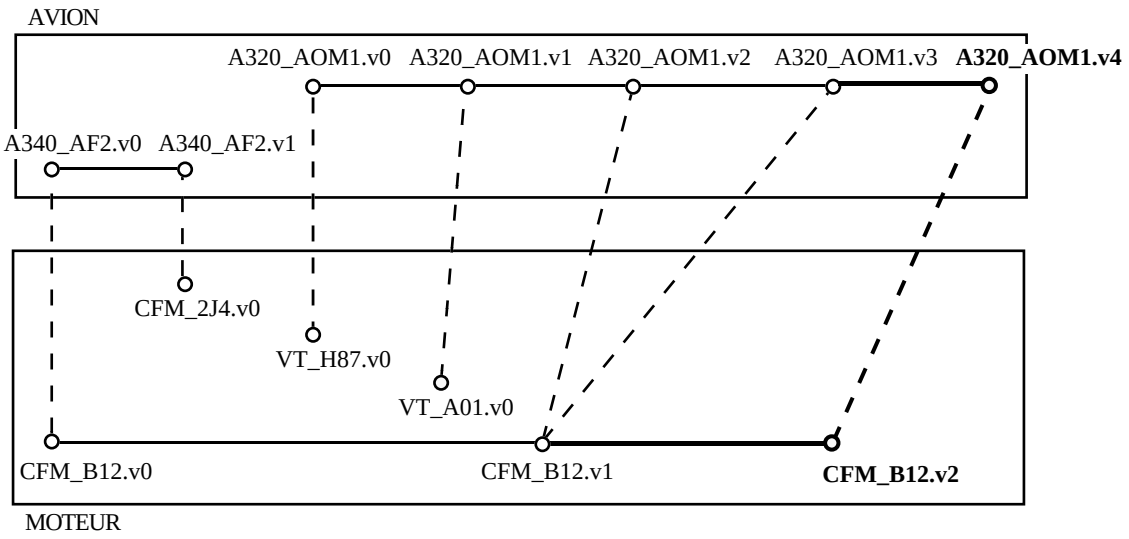
Lorsque la version A320_AOM1.v0 est dérivée pour créer la version A320_AOM1.v1, elle peut être composée d'une nouvelle version composante VT_A01.v0 racine d'une nouvelle hiérarchie VT_A01 ; l'avion A320_AOM1 change de moteur pour un nouveau moteur VT_A01 n'existant pas dans la base.



Lorsque la version A320_AOM1.v1 est dérivée pour créer A320_AOM1.v2, elle est composée d'une version composante CFM_B12.v1 dérivée d'une feuille libre CFM_B12.v0 de la hiérarchie CFM_B12 ; l'avion A320_AOM1 change de moteur pour être équipé du moteur CFM_B12 qui équipait mais n'équipe plus un autre avion, l'A340_AF2.



Lorsque la version A320_AOM1.v2 est dérivée pour créer A320_AOM1.v3, la version A320_AOM1.v3 est composée de la même version composante CFM_B12.v1 que la version A320_AOM1.v2 dont elle dérive ; l'avion A320_AOM1 évolue mais conserve le même moteur qui lui ne change pas.



Enfin, lorsque la version A320_AOM1.v3 est dérivée pour créer A320_AOM1.v4, la version A320_AOM1.v4 est composée de la version composante CFM_B12.v2 dérivée de la version composante CFM_B12.v1 composant la version A320_AOM1.v3 dont A320_AOM1.v4 dérive ; l'avion A320_AOM1 évolue conservant le même moteur qui lui aussi a évolué.

- ***Dérivation d'une version composante***

La dérivation d'une version composante entraîne la dérivation de toutes les versions de la hiérarchie de composition "supérieure" à la version dérivée ; ceci est nécessaire pour retrouver les hiérarchies de composition de versions (cf. début § 3.7.2.2.1.). Le principe de répercussion de la dérivation d'une version composante sur les versions composées est le suivant :

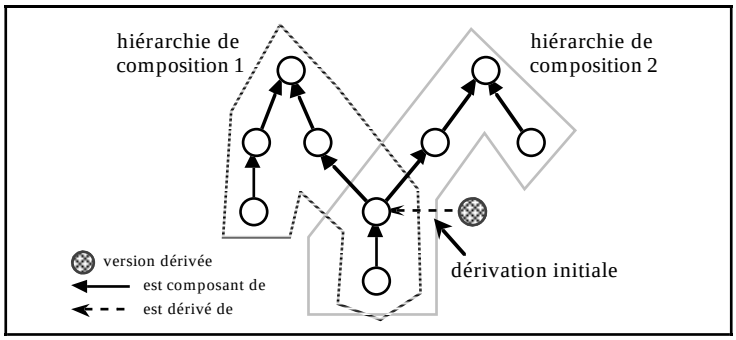


Figure II-17a : Dérivation d'une version composante

- pour tous les chemins de versions partant de la version dérivée, jusqu'à la version racine de la hiérarchie de composition, les versions des chemins sont dérivées. Les nouvelles versions sont liées suivant des chemins analogues aux versions précédentes,

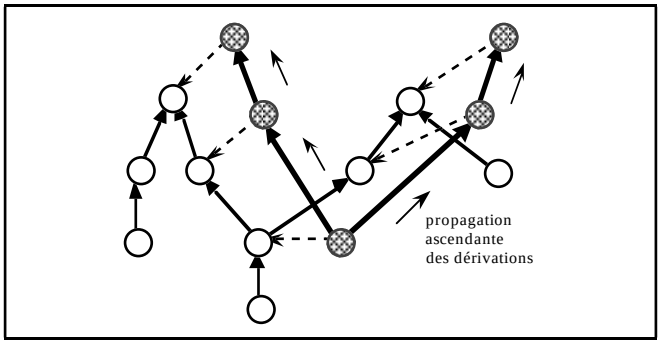


Figure II-17b : Propagation de dérivation d'une version composante

- pour chaque version qui est dérivée, la nouvelle version créée est liée aux mêmes versions composantes que la version précédente.

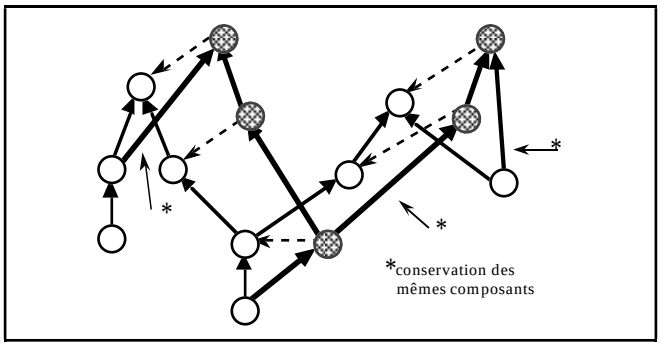


Figure II-17c : Conservation des mêmes versions composantes

Pour obtenir d'autres configurations de dérivation, notamment pour des hiérarchies de composition partageant les mêmes composants, la dérivation doit être descendante et débuter à la version pour laquelle la hiérarchie de composition est modifiée. Ce principe rejoint le principe du Check-in proposé par Katz (Katz, 90a).

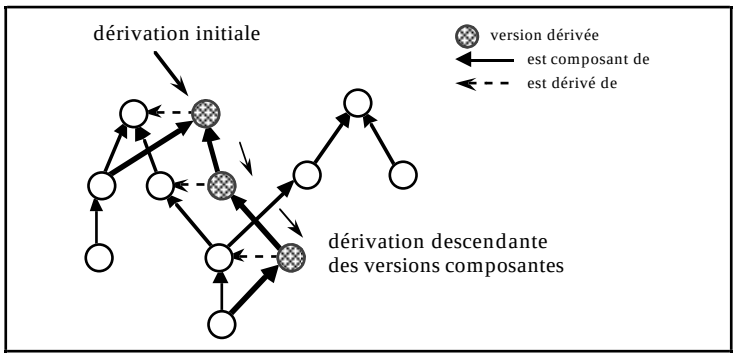


Figure II-18 : Dérivation descendante des versions composantes

• ***Suppression d'une version***

Compte tenu de la sémantique de la composition, la suppression d'une version composée entraîne la suppression de ses versions composantes, excepté si :

- elles sont liées à d'autres versions composées du même arbre de dérivation ; plusieurs versions composées peuvent en effet avoir les mêmes versions composantes,
- elles composent d'autres versions composées d'autres arbres de dérivation (uniquement lorsque le lien est partagé).

La suppression d'une version composante implique la suppression des versions définitives ; les versions composées provisoires sont soit supprimées ou rattachées à au moins une autre version composante lorsque la version composante supprimée est le dernier composant des versions composées.

L'élagage qui consiste en plusieurs suppressions de versions dérivées successives, suit les règles définies pour la suppression de versions.

• ***Modification d'une version***

Seule la modification des versions composées peut modifier les liens avec les versions composantes ; dans ce cas, les conséquences établies pour la suppression d'une version composée s'appliquent aux versions composantes qui ne sont plus liées, et les règles pour la création d'une version composée s'appliquent pour les liens avec les nouvelles versions composantes.

3.7.2.3. Les solutions pour l'association

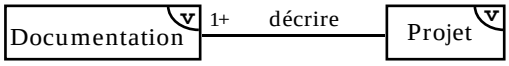
A l'inverse de la composition, la sémantique de l'association n'impose pas de sens hiérarchique pour les liaisons entre instances. Il n'est donc pas nécessaire de dupliquer des versions pour conserver une hiérarchie comme pour la composition (cf. § 3.7.2.2.1.). Les versions d'objets liées par un lien d'association évoluent de manières indépendantes. Les contraintes ne s'expriment alors pas comme pour la composition si l'on veut éviter la duplication inutile de certaines versions. Seules les cardinalités fixées pour les classes associées induisent des contraintes.

La solution pour l'expression de ces contraintes est basée sur celle définie pour la notion d'exclusivité/partage des liens de composition c'est-à-dire d'exprimer des contraintes sur les liaisons entre versions et hiérarchies de versions (cf. § 3.7.2.2.1.).

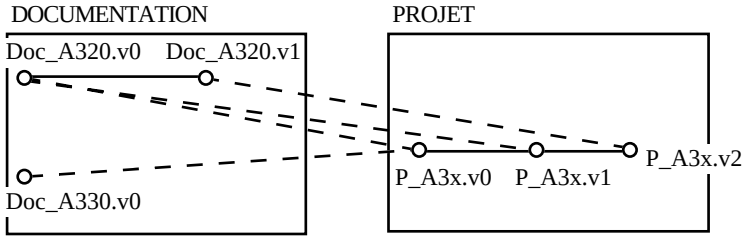
3.7.2.3.1. L'expression des contraintes sur les versions d'objets

Selon l'approche choisie, la cardinalité fixée pour chacune des classes de versions associées impose et/ou restreint le nombre de liaisons entre une version d'une classe et des hiérarchies de versions de la classe associée. Une cardinalité de type α-β pour une classe de versions impose que toute version de la classe soit liée à au moins α versions et au plus β hiérarchies de la classe associée.

Exemple II-8 : Considérons le lien d'association entre les classes simples de versions Documentation (cardinalité 1-1) et Projet (cardinalité 1-N) suivant :



Les différentes possibilités de liens entre des instances des deux classes sont maintenant illustrées.



Une version de Projet (ex. P_A3x.v0) doit être liée à au moins une version de Documentation (ex. Doc_A320.v0) et peut être liée à plusieurs hiérarchies de Documentation (ex. Doc_A320 et Doc_A330) via des versions de ces hiérarchies (ex. Doc_A320.v0 et Doc_A330.v0). Inversement, une version de Documentation (ex. Doc_A320.v1) doit être rattachée à une seule hiérarchie de Projet (ex. P_A3x) via au moins une version de cette hiérarchie (ex. P_A3x.v0) ; elle peut néanmoins être liée à plusieurs versions mais appartenant toutes à la même hiérarchie (ex. P_A3x.v0 et P_A3x.v1 appartenant à P_A3x).

3.7.2.3.2. Respect des contraintes par les opérations

Les opérations qui doivent prendre en compte les contraintes sont les mêmes que pour la composition ; pour l'association, les opérations agissent de manière identique pour les versions de chacune des classes, puisqu'il n'y a pas la distinction version composée/version composante présente dans la composition. La création et la dérivation de versions reprennent une partie des principes définis pour la composition. La suppression (et l'élagage) de versions dans une classe suit les mêmes règles que la suppression de versions composantes (cf. § 3.7.2.2.2.). La modification respecte les règles établies pour la création, que ce soit pour le rattachement à d'autres versions ou pour la suppression de versions pour les versions qui ne sont plus liées.

• ***Création d'une version racine***

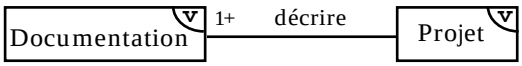
D'une part, lorsque l'on crée une version racine, la cardinalité fixée pour la classe impose le nombre minimum et le nombre maximum de liens qu'il faut spécifier avec des versions de la classe associée. Une cardinalité 1-N pour une classe implique uniquement que toute version racine créée dans la classe doit être liée à au moins une version de la classe associée.

D'autre part, lors de la création de la version racine, la cardinalité de la classe associée définit les versions auxquelles la version racine peut être liée. Si la cardinalité maximum est de 1 pour la classe associée, la nouvelle version créée peut être liée à :

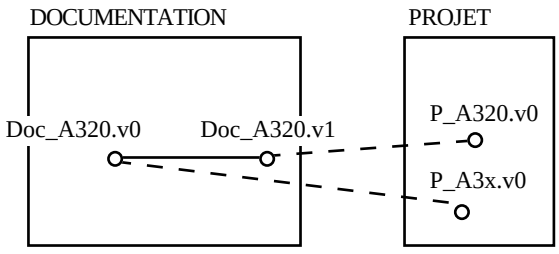
- une nouvelle version créée dans la classe associée,
- une version dérivée d'une version feuille libre d'une hiérarchie déjà existante dans la classe associée ; la notion de feuille libre est celle définie précédemment pour la composition (cf. § 3.7.2.2.2.),
- une version feuille non rattachée (uniquement possible lorsque la cardinalité minimum est 0 pour la classe).

Si la cardinalité maximum est de N pour la classe associée, la nouvelle version créée peut être liée à toute version de la classe associée.

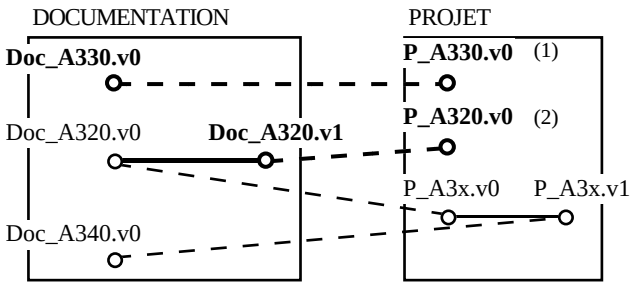
Exemple II-9 : Considérons le lien d'association entre les classes simples de versions Documentation (cardinalité 1-1) et Projet (cardinalité 1-N) suivant :



Supposons que la situation initiale au niveau des instances soit la suivante :



Dans ce cas d'association, seulement deux possibilités de rattachement sont possibles lors de la création d'une version dans la classe Documentation.



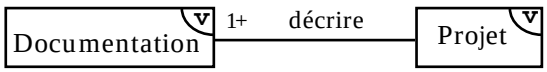
Par exemple, une version racine créée dans PROJET est liée à une nouvelle version racine créée en même temps dans DOCUMENTATION (ex. (1) P_A330.v0 est créée, liée à Doc_A330.v0 créée en même temps) ; le nouveau Projet est associé à une nouvelle Documentation. Autrement une version racine créée dans PROJET est liée à une version dérivée d'une version feuille libre de DOCUMENTATION (ex. (2) P_A320.v0 est créée, liée à Doc_320.v1 dérivée de la version feuille libre Doc_320.v0 abandonnée par P_A3x) ; le nouveau Projet reprend une Documentation qui avait été abandonnée.

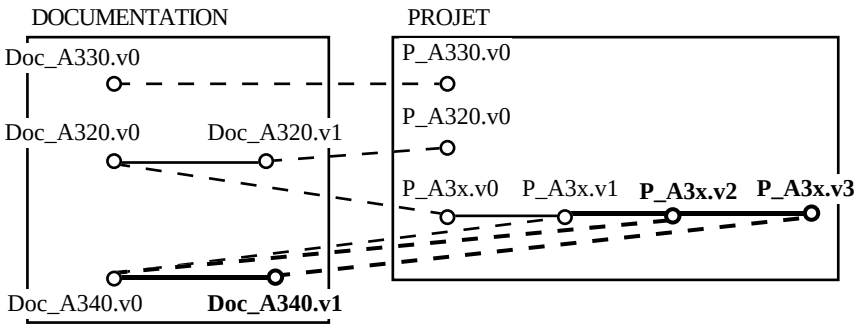
• ***Dérivation d'une version***

Une nouvelle version créée dans une classe par dérivation respecte les règles précédentes établies pour la création d'une version. Cependant, la dérivation autorise des liens supplémentaires avec des versions de la classe associée. Les versions de la classe associée qui peuvent être liées à la version créée sont :

- les mêmes catégories de versions que lors de la création (cf. § précédent),
- les mêmes versions que la version précédente,
- des versions dérivées des versions précédentes liées.

Exemple II-10 : Reprenons l'exemple d'association précédent entre les classes de versions Documentation et Projet, dans son état final au niveau des instances :





Une nouvelle version dérivée de Projet peut être liée à des versions de la classe Documentation comme lors de la création (cf. exemple précédent) ; elle peut en plus être liée à :

- la même version que la version dont elle dérive (la nouvelle version P_A3x.v2 créée par dérivation de P_A3x.v1 est associée à la même version de documentation Doc_A340.v0 ; le projet P_A3xa Doc_A320 évolue en restant associé à la même documentation qui elle n'a pas évolué).
- une version dérivée de celle précédemment liée (la nouvelle version P_A3x.v3 créée par dérivation de P_A3x.v2 est associée à une nouvelle version Doc_A340.v1 dérivée de la version Doc_A340.v0 associée à P_A3x.v2 dont dérive P_A3x.v3 ; le projet P_A3x évolue en restant associé à la même documentation Doc_A340 qui elle aussi a évolué).

3.7.3. Composition et association entre objets et versions

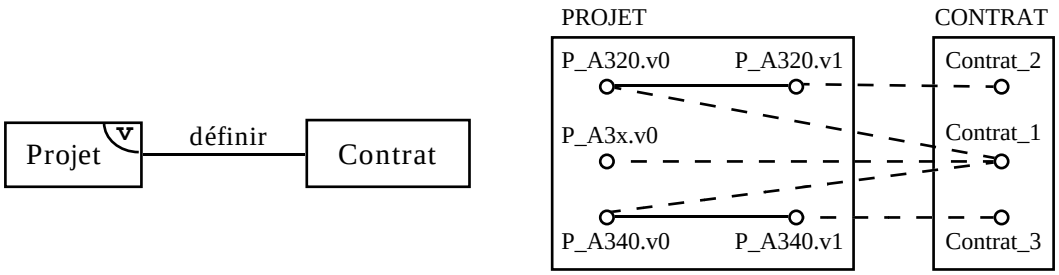
Le fait que l'évolution est traduite différemment avec des objets et avec des versions induit certaines spécificités pour les contraintes inhérentes aux relations de composition et d'association entre classes de versions et classes d'objets. Deux cas de composition se distinguent suivant que l'on a des versions d'objets composées d'objets ou des objets composés de versions d'objets. Le cas d'objets composés de versions suit les principes définis pour l'association.

3.7.3.1. La problématique liée aux différences entre objets et versions

La problématique des relations entre classes d'objets et classes de versions réside dans la différence de comportement entre objets et versions d'objets c'est-à-dire dans le fait que l'on relie des entités pour lesquelles on ne conserve qu'un seul état (objet) à des entités pour lesquelles on conserve plusieurs états (versions d'objets) et que ces relations entre entités évoluent dans le temps.

Lorsque l'on conserve des versions d'objets pour décrire une entité, cela implique que l'on conserve leurs liens avec les objets pour décrire les relations entre entités représentées. Un objet se retrouve ainsi lié à toutes les versions décrivant toutes les entités auxquelles l'entité décrite par l'objet à été reliée au cours de son évolution.

Exemple II-11 : Supposons une relation entre des contrats pour lesquels on ne gère pas de versions et des projets pour lesquels on gère des versions ; un contrat ne correspond à chaque fois qu'à un seul projet et inversement mais un contrat peut changer de projets au cours du temps et inversement. Il s'agit d'une association 1-1, 1-1 entre une classe d'objets Contrat et une classe de versions Projet.



Le contrat Contrat_1 correspond tout d'abord au projet P_A320, puis au projet P_A340 puis au projet P_A3x. Les différentes versions des projets sont conservées et par conséquent leurs liens avec l'objet Contrat_1. Le problème réside dans l'expression des cardinalités 1-1 au niveau des instances des deux classes liées, en particulier pour les objets (ex. Contrat_1).

La traduction des cardinalités fixées pour les relations entre classes doit prendre en compte ce phénomène. A notre connaissance aucune solution pour ce cas de figure n'a été proposée dans la littérature pour les systèmes à gestion de versions d'entités.

3.7.3.2. Les solutions pour l'expression des contraintes

Classe de versions composée d'une classe d'objets

Ce cas peut être considéré comme un cas particulier de composition entre classes simples de versions (cf. § 3.7.2.2.) ; en effet, un objet composant peut être considéré comme une version d'objet composante qui ne peut être dérivée. Les contraintes sous-jacentes à la définition d'un lien de composition suivent ainsi les principes suivants :

- la cardinalité fixée pour la classe de versions composées, impose le nombre minimum et le nombre maximum d'objets composants pour une version composée. Une cardinalité 0-1 pour la classe composée indique par exemple qu'une version ne peut avoir qu'un seul objet composant mais qu'elle peut également ne pas avoir d'objet composant.
- la notion d'exclusivité / partage précise le nombre minimum et le nombre maximum de hiérarchies de versions qu'un objet composant peut composer. L'exclusivité impose qu'un objet composant ne peut composer qu'une seule hiérarchie c'est-à-dire plusieurs versions composées mais appartenant toutes à la même hiérarchie de dérivation.

Association entre classe d'objets et classe de versions

Pour respecter les cardinalités fixées pour la classe de versions, la conservation de versions implique la conservation de leurs liens avec des objets ; la différence de comportement entre les instances liées, à savoir des objets et des versions d'objets,

pose alors un problème pour l'expression des contraintes liées à la cardinalité fixée pour la classe d'objets. Pour exprimer par exemple qu'une entité du monde réel est liée à plusieurs autres entités au cours du temps et à une seule à chaque instant, les solutions choisies précédemment pour exprimer les contraintes ne sont plus suffisantes ; il n'est pas possible d'exprimer des contraintes :

- ni sur les liaisons entre objets et versions (cf. § 3.7.2.2.1. respect de la cardinalité d'une classe composée) ; une cardinalité 1-1 impliquerait par exemple de ne pouvoir lier un objet qu'à une seule version ce qui est trop restrictif par rapport à la réalité représentée,
- ni sur les liaisons entre objets et hiérarchies de versions (cf. § 3.7.2.2.1. respect de la notion d'exclusivité / partage) ; une cardinalité 1-1 impliquerait de ne lier un objet qu'à une seule hiérarchie de versions ce qui est encore trop restrictif par rapport à la réalité représentée.

La solution à ce problème est d'exprimer les contraintes sur les liens entre un objet et des hiérarchies de versions et de distinguer deux catégories de liens pour les objets :

- **les liens courants** : un lien courant pour un objet est le dernier lien défini entre l'objet et une version. Une hiérarchie est courante pour un objet lorsque l'une de ses versions est liée à l'objet par un lien courant.
- **les liens passés** : les liens passés existent en raison de la conservation des versions qui ont été liées à l'objet.

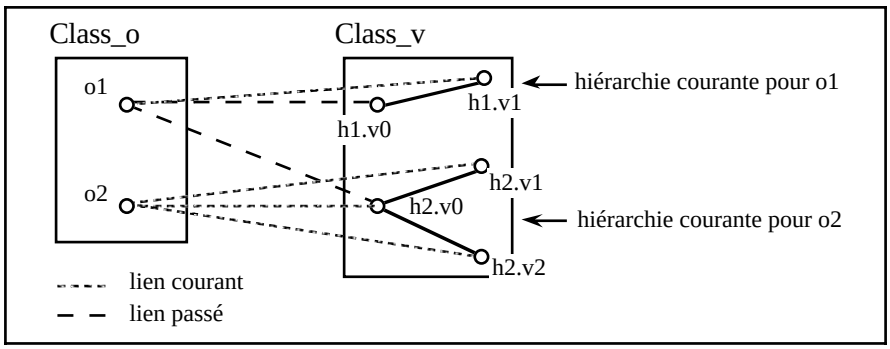


Figure II-19 : Liens courants - liens passés - hiérarchie courante

La cardinalité fixée pour la classe d'objet impose ainsi le nombre minimum et le nombre maximum de hiérarchies courantes pour un objet. Une cardinalité 0-1 pour la classe d'objet implique par exemple que l'objet est lié au maximum à une seule hiérarchie courante mais il peut rester non lié.

A l'opposé, la cardinalité fixée pour la classe de versions indique les nombres minimum et maximum d'objets liés à une version.

Classe d'objets composée d'une classe de versions

Les principes définis pour l'association entre classes d'objets et classes de versions s'appliquent également à la composition entre une classe d'objets composée d'une classe de versions.

3.7.3.3. Respect des contraintes par les opérations

Les opérations agissant sur les liens entre versions d'objets qui doivent respecter les contraintes sont la création, la dérivation, la suppression et la modification lorsque celle-ci porte sur la modification d'un lien. Seul le deuxième cas de composition (classe d'objets composée d'une classe de versions) est présenté ; le premier cas (classe de versions composée d'une classe d'objets) peut être considéré comme un cas particulier de composition entre classes de versions (cf. § 3.7.2.2.2.), certaines possibilités de liaisons n'étant pas possibles compte tenu qu'un objet ne peut être dérivé.

Classe d'objets composée d'une classe de versions - Association entre classe d'objets et classe de versions

- ***Création d'un objet***

Pour la création d'un objet, ce cas s'apparente aux relations entre classes de versions. Pour la composition entre une classe d'objets composée d'une classe de versions, la création d'un objet composé suit des principes similaires à la création d'une version composée définis pour la composition entre classes de versions (cf. § 3.7.2.2.2.). Pour l'association entre classes d'objets et classes de versions, la création d'un objet s'apparente à la création d'une version racine pour une association entre classes de versions (cf. § 3.7.2.3.1.).

- ***Création d'une version (uniquement pour l'association)***

Lorsque l'on crée une version racine, la cardinalité de la classe de versions, impose les nombres minimum et maximum d'objets de la classe liée auxquels la nouvelle version doit être liée.

A l'opposé, la cardinalité de la classe d'objets liée restreint les objets auxquels il est possible de se lier. Si le lien est mono-valué (cardinalité de type α-1) pour la classe d'objets, la nouvelle version peut être liée uniquement :

- à de nouveaux objets créés dans la même transaction que la nouvelle version,
- à des objets existants libres. Un objet libre est un objet qui n'est pas lié à une version feuille d'une hiérarchie de versions de la classe liée ; cette notion

s'apparente à la notion de version feuille libre définie précédemment (cf. § 3.7.2.2.1.),

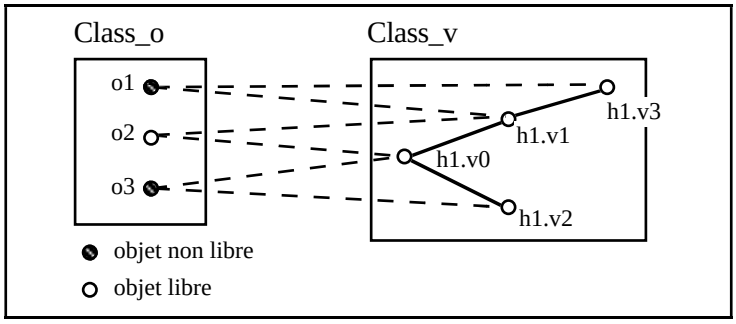
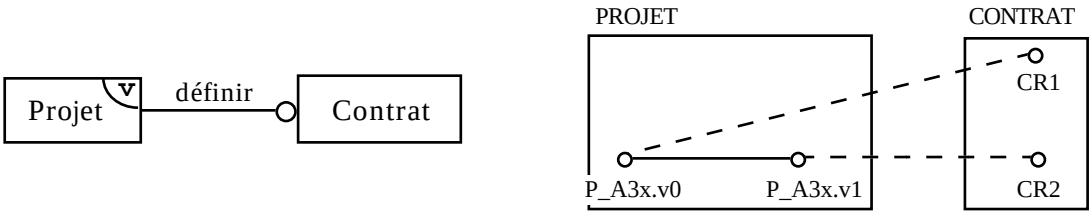


Figure II-20 : Objets libres

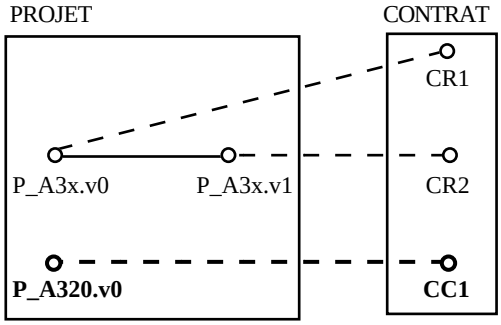
- soit des objets existants non liés. Ces objets n'existent que pour une cardinalité 0-1 pour la classe d'objets.

Pour une cardinalité de type α-N pour la classe d'objets, la nouvelle version peut également être liée à tout objet existant dans la classe d'objets.

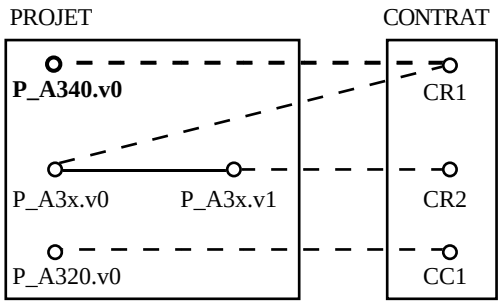
Exemple II-12 : Considérons le lien d'association "définir" entre la classe de versions Projet (cardinalité 0-1) et la classe d'objets Contrat (cardinalité 1-1) suivant :



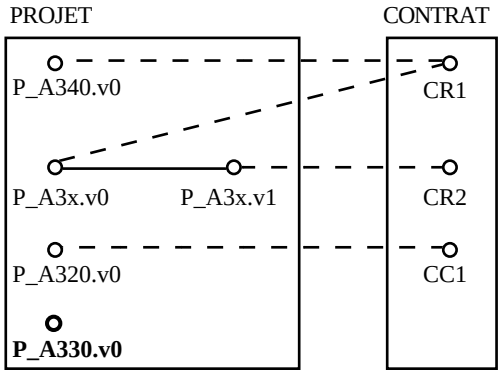
D'une part, la cardinalité 0-1 pour la classe Projet implique qu'une version créée dans la classe Projet peut être associée à un seul objet de la classe Contrat mais peut ne pas être associée à un objet de la classe Contrat. Les possibilités d'association d'une version racine créée dans la classe Projet sont illustrées ci-après avec la création des versions racines P_A320.v0, P_A340.v0 et P_A330.v0.



La nouvelle version racine P_A320.v0 créée est associée à un nouvel objet CC1 créé en même temps ; le nouveau projet est associé à un nouveau contrat.



La nouvelle version racine P_A340.v0 créée est associée à un objet CR1 existant libre (CR1 est associé au projet P_A3x qui maintenant est associé au contrat CR2 via P_A3x.v1) ; le nouveau projet est associé à un contrat existant qui a été abandonné par un autre projet.



La nouvelle version racine P_A330.v0 créée n'est associée à aucun objet de Contrat ; le nouveau projet ne correspond pour l'instant à aucun contrat.

- ***Dérivation d'une version***

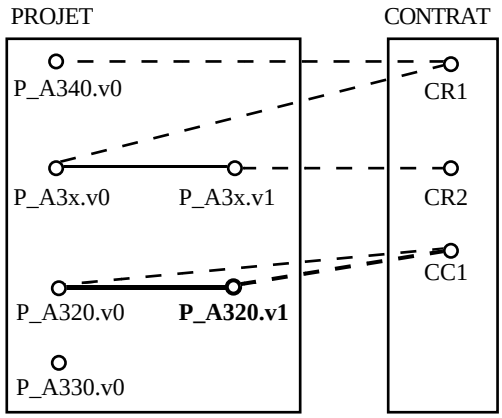
La dérivation est une opération applicable uniquement sur les versions d'objets.

Pour la composition, la dérivation ne porte que sur les instances de la classe composante. Une version composante créée par dérivation est automatiquement rattachée au même objet composé que la version dont elle dérive ; le changement de version composante ne peut s'effectuer qu'au niveau d'un objet composé.

Pour l'association, la dérivation d'une version suit les mêmes règles pour le respect de la cardinalité de la classe de versions que celles définies précédemment pour la création d'une version racine. La différence se situe au niveau des objets de la classe liée auxquels la nouvelle version peut être rattachée. Ces objets sont :

- soit les mêmes catégories d'objets que lors de la création (cf. paragraphe précédent) c'est-à-dire de nouveaux objets, des objets libres, des objets non liés (cardinalité de type 0-β pour la classe d'objets uniquement), ou tout objet existant (cardinalité de type α-N pour la classe d'objets uniquement)
- soit les mêmes objets liés à la version précédente.

Exemple II-13 : Reprenons l'exemple de la création de version précédent avec les mêmes instances :



La nouvelle version P_A320.v1 créée par dérivation de P_A320.v0 reste associée au même objet CC1 ; le projet évolue mais pas le contrat auquel il est associé.

• ***Suppression d'un objet, suppression d'une version***

Que ce soit dans une classe composée pour la composition ou dans une classe liée pour l'association, la suppression d'un objet suit les mêmes principes que ceux définis pour la suppression d'une version pour des relations analogues entre deux classes de versions (cf. § 3.7.2.2. et 3.7.2.3.).

D'autre part, la suppression d'une version composante affecte l'objet composé uniquement lorsque cette version est le dernier composant de l'objet et que la cardinalité est de type 1-β pour la classe composée. Dans ce cas, l'objet composé doit être supprimé ou rattaché à au moins une autre version composante.

Le même principe s'applique lorsque l'on supprime la dernière version liée à un objet pour une association avec une cardinalité de type 1-β pour la classe d'objets.

3.7.4. Synthèse des contraintes pour la composition et l'association

Ce paragraphe fait la synthèse des conséquences de la gestion de versions d'objets pour les contraintes, sur les instances, inhérentes aux relations de composition et d'association. Il résume également les principales conséquences sur les opérations, c'est-à-dire lors de la définition de nouvelles instances composée, pour la composition, et lors de la définition de nouvelles instances pour l'association.

3.7.4.1. Synthèse des contraintes pour la composition

Les contraintes inhérentes à la composition, exprimées sur les instances composées, sont résumées dans le tableau suivant :

classe composée		classe composante	
		objets	versions
objets	0-β	1 objet composé peut rester sans objet composant	1 objet composé peut rester sans version / hiérarchie composante
	1-1	1 objet composé à 1 seul objet composant	1 objet composé lié à 1 seule hiérarchie composante courante
	α-N	1 objet composé peut avoir plusieurs objets composants	1 objet composé peut être lié à plusieurs hiérarchies composantes
versions	0-β	1 version composée peut rester sans objet composant	1 version composée peut rester sans version / hiérarchie composante
	1-1	1 version composée à 1 seul objet composant	1 version composée lié à 1 seule hiérarchie composante courante
	α-N	1 version composée peut avoir plusieurs objets composants	1 version composée peut être lié à plusieurs hiérarchies composantes

Figure II-21 : Tableau récapitulatif des contraintes sur les instances composées

Les contraintes inhérentes à la composition, exprimées sur les instances composantes, sont résumées dans le tableau suivant :

classe composante		classe composée	
		objets	versions
objets	1-1	1 objet composant est une partie d'1 seul objet composé	1 objet composant est une partie d'1 seule hiérarchie composée
	1-N	1 objet composant peut faire partie de plusieurs objets composés	1 objet composant peut faire partie de plusieurs hiérarchies composées
versions	1-1	1 version composante est une partie d'1 seul objet composé	1 version composée est une partie d'1 seule hiérarchie composée
	1-N	1 version composante peut faire partie de plusieurs objets composés	1 version composante peut faire partie de plusieurs hiérarchies composées

Figure II-22 : Tableau récapitulatif des contraintes sur les instances composantes

3.7.4.2. Synthèse des contraintes pour l'association

Le tableau suivant résume les contraintes, sur les instances, inhérentes aux associations :

classe A		classe associée B	
		objets	versions
objets	0-β	1 objet de A peut exister sans être lié à 1 objet de B	1 objet de A peut exister sans être lié à 1 hiérarchie de B
	1-1	1 objet de A est lié à 1 seul objet de B	1 objet de A est lié à 1 seule hiérarchie courante de B
	α-N	1 objet de A peut être lié à plusieurs objets de B	1 objet de A peut être lié à plusieurs hiérarchies de B
versions	0-β	1 version de A peut exister sans être liée à 1 objet de B	1 version composée peut exister sans être liée à 1 hiérarchie de B
	1-1	1 version de A est liée à 1 seul objet de B	1 version de A est liée à 1 seule hiérarchie de B
	α-N	1 version composée peut avoir plusieurs objets composants	1 version composée peut être lié à plusieurs hiérarchies composantes

Figure II-23 : Tableau récapitulatif des contraintes pour les associations

3.7.4.3. Synthèse des conséquences sur les opérations

Le tableau suivant illustre les différents composants possibles pour une nouvelle instance composée, définie par création ou par dérivation :

classe composante		classe composée instances composantes auxquelles il est possible de lier une nouvelle instance composée		
		objets	versions	
		création	création	dérivation
objets	1-1	• nouvel objet composant	• nouvel objet composant + objet composant libre	• idem création + même objet composant
	1-N	• idem 1-1 + tout objet composant existant	• idem 1-1 + tout objet composant existant	
versions	1-1	• nouvelle version composante racine + dérivée de version composante libre	• nouvelle version composante racine + dérivée de version composante libre	• idem création + même version composante + dérivée de version composante
	1-N	• idem 1-1 + toute version composante existante	• idem 1-1 + toute version composante existante	

Figure II-24 : Tableau récapitulatif des créations / dérivations d'instances composées

Le tableau suivant illustre, pour la définition d'une nouvelle instance dans une classe, les possibilités de rattachement aux instances de la classe liée par une association :

<table>
<tr><td colspan="2" rowspan="3">classe
associée B</td><td colspan="3">classe A
instances de B auxquelles il est possible
de lier une nouvelle instance de A</td></tr>
<tr><td>objets</td><td colspan="2">versions</td></tr>
<tr><td>création</td><td>création</td><td>dérivation</td></tr>
<tr><td rowspan="4">objets</td><td>1-1</td><td>• nouvel objet de B</td><td>• nouvel objet de B
+ objet libre de B</td><td>• idem création
+ même objet lié de B</td></tr>
<tr><td>0-1</td><td>• idem 1-1
+ objet existant de B non lié</td><td>• idem 1-1
+ objet existant de B non lié</td><td>• idem création
+ même objet lié de B</td></tr>
<tr><td>0-N</td><td>• idem 0-1
+ tout objet existant dans B</td><td colspan="2">• idem 0-1
+ tout objet existant dans B</td></tr>
<tr><td>1-N</td><td>• idem 1-1
+ tout objet existant dans B</td><td colspan="2">• idem 1-1
+ tout objet existant dans B</td></tr>
<tr><td rowspan="4">versions</td><td>1-1</td><td>• nouvelle version racine de B
+ dérivée de version libre de B</td><td>• nouvelle version racine de B
+ dérivée de version libre de B</td><td>• idem création
+ même version liée de B
+ dérivée de version liée de B</td></tr>
<tr><td>0-1</td><td>• idem 1-1
+ version existante de B non liée</td><td>• idem 1-1
+ version existante de B non liée</td><td>• idem création
+ même version liée de B
+ dérivée de version liée de B</td></tr>
<tr><td>0-N</td><td>• idem 0-1
+ toute version existante de B</td><td colspan="2">• idem 0-1
+ toute version existante de B</td></tr>
<tr><td>1-N</td><td>• idem 1-1
+ toute version existante de B</td><td colspan="2">• idem 1-1
+ toute version existante de B</td></tr>
</table>

Figure II-25 : Tableaux récapitulatif des créations / dérivations pour les associations

3.8. Versions de classes et relations de composition et d'associations

Lorsque des versions de classes sont liées par des relations de composition ou d'association, la définition d'une nouvelle version de classe par dérivation implique la conservation de la relation ou une évolution de la relation. La dérivation d'une version de classe peut impliquer une modification de la relation (cardinalités, ...). Au niveau des instances, les contraintes restent analogues à celles définies pour les classes simples en tenant compte des particularités liées au fait que les différentes versions d'une classe décrivent les mêmes entités du monde réel (Hubert, 95b).

3.8.1. Nouvelle version de classe et relations existantes

L'opération de dérivation appliquée aux classes, qui consiste à définir une version d'une classe à partir d'une version de classe existante, tient compte des liens définis pour la version de classe dérivée.

Composition

Pour une relation de composition, seule la dérivation de la classe composée peut modifier les caractéristiques de la relation de composition, voire la supprimer. La dérivation de la classe composante implique nécessairement la conservation telle quelle de la relation de composition.

La dérivation de la classe composante entraîne la dérivation de la classe composée avec conservation systématique de la relation de composition sans modification de la cardinalité de la classe composée. Il n'est pas possible d'agir sur classe composée à partir de la classe composante.

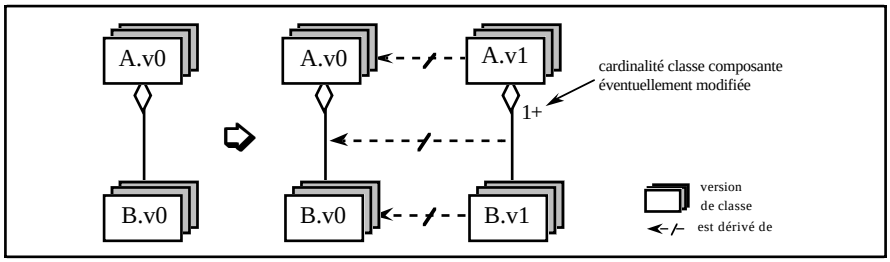


Figure II-26 : Dérivation d'une version de classe composante

En revanche, la dérivation de la classe composée implique plusieurs cas de figure possibles :

- dérivation de la classe composée sans dérivation de la classe composante ; la cardinalité de la relation peut éventuellement être modifiée mais pas la nature de la relation,

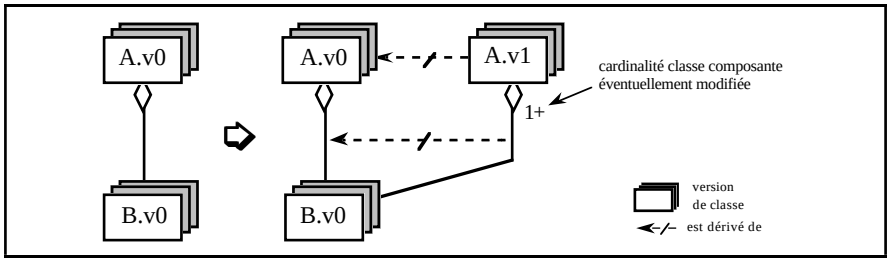


Figure II-27a : Dérivation d'une version de classe composée seule

- dérivation de la classe composée avec dérivation de la classe composante et modification de la nature de la relation ; la cardinalité de la relation peut éventuellement être modifiée,

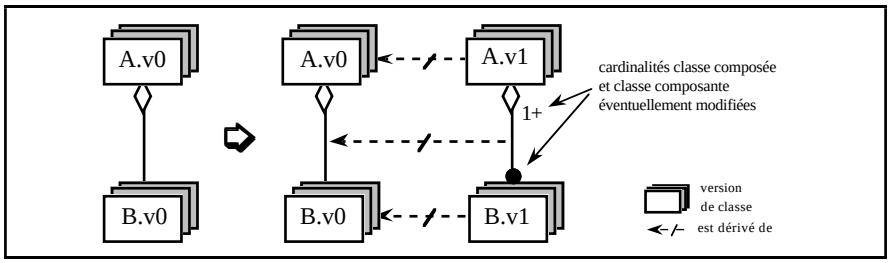


Figure II-27b : Dérivation simultanée des classes composée et composante

- dérivation de la classe composée sans conservation de la relation de composition,

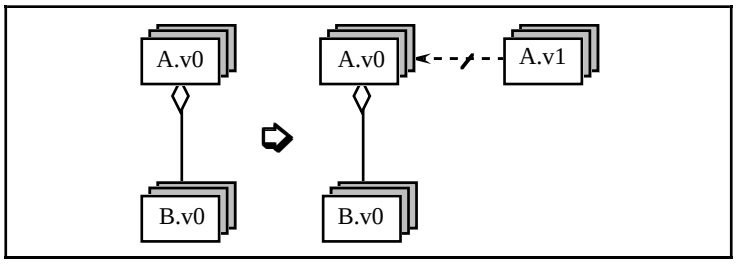


Figure II-27c : Dérivation de classe composée sans conservation de composition

Association

Pour une relation d'association, la dérivation d'une des classes peut impliquer l'évolution de la relation d'association voire de la classe liée. Plusieurs cas de figure peuvent exister selon que l'une ou l'autre des classes est dérivée ou les deux simultanément.

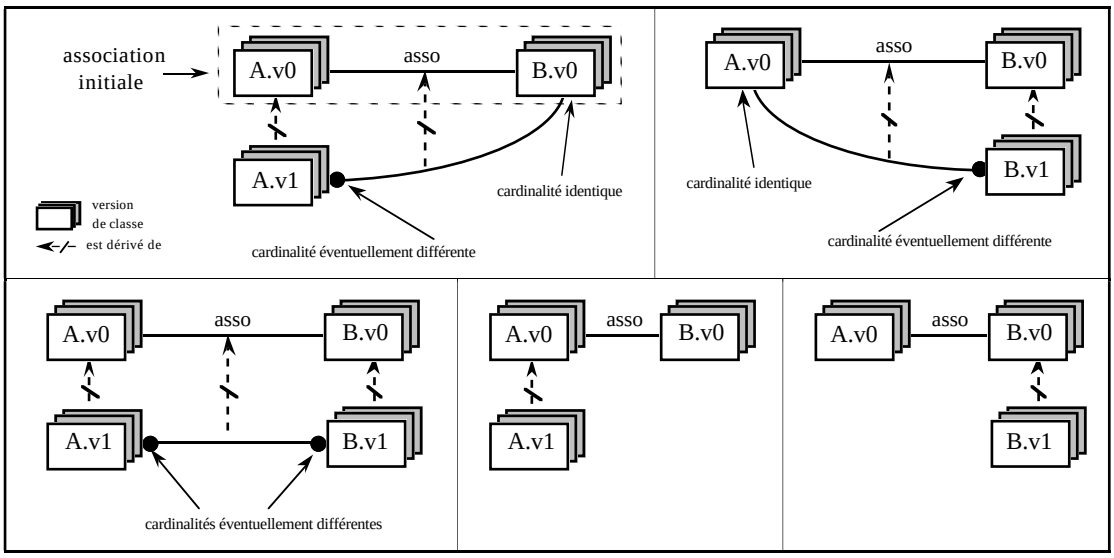


Figure II-28 : Evolutions d'association entre versions de classes

Relations entre des versions de classes et des classes simples

Lorsqu'il y a des versions de classes uniquement pour une seule des classes liées, les cas possibles d'évolution des relations pour la composition et l'association sont limités à ceux pour lesquels une seule des classes est dérivée.

Le cas d'une composition avec uniquement des versions de classe composante présente un cas particulier lorsque l'on dérive celle-ci ; une nouvelle version de classe composante est obligatoirement rattachée à la même classe composée non-versionnalisable en conservant la même composition.

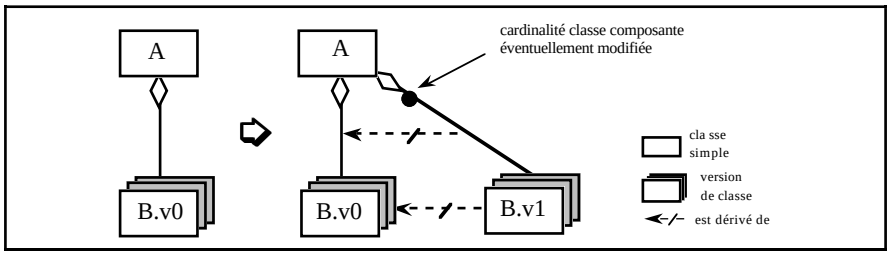


Figure II-29 : Cas particulier de dérivation d'une version de classe composante

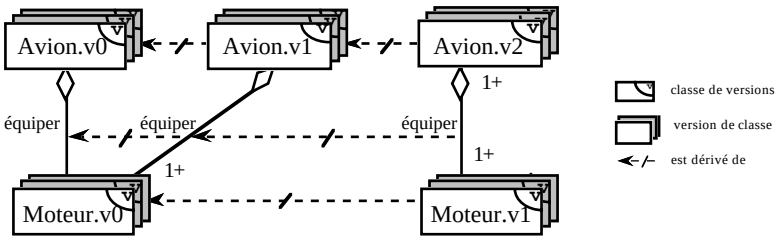
3.8.2. Conséquences au niveau des instances

La création de versions de classes par dérivation avec conservation de la même relation (de composition ou d'association) implique l'existence d'une relation entre une classe et plusieurs versions d'une autre classe. Dans ce cas, les contraintes exprimées pour les classes simples (cf. § 3.7.) restent valables en considérant :

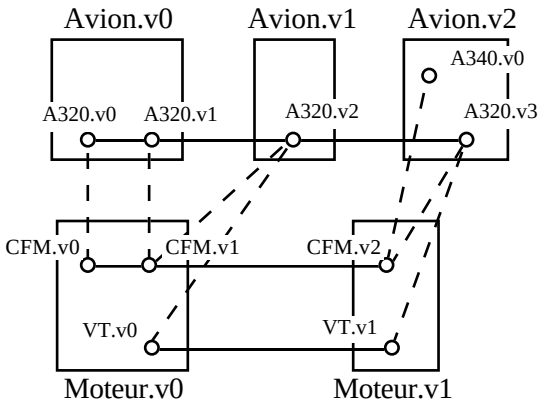
- que la dérivation d'une version peut s'effectuer dans une classe dérivée,

- que les règles concernant des versions d'objets d'une même hiérarchie et appartenant à des versions de classes différentes s'appliquent comme si elles appartenaient à une seule classe.

Exemple II-14 : considérons le cas d'une composition entre les versions de classe *Avion* et les versions de classe *Moteur* suivant :



Les instances des deux classes suivent les règles définies pour les relations entre classes simples de versions (cf. § 3.7.2.) en considérant les différentes versions de classes liées.



Par exemple, les versions d'objets instances de la version de classe Avion.v0 suivent les règles définies pour un lien de composition mono-valué (1-1). Ainsi, les versions d'objets A320.v0 et A320.v1 doivent avoir une version composante appartenant à la classe Moteur.v0 ; elles ne peuvent avoir qu'une seule version composante. De même, les versions d'objets instances des versions de classe Avion.v1 et Avion.v2 suivent les règles définies pour les liens de composition multi-valués (1-N) ; ainsi, les versions d'objets A320.v2 et A320.v3 doivent avoir au moins une version composante appartenant à la classe Moteur.v0 pour A320.v2 et appartenant à la classe Moteur.v1 pour A320.v3 ; elles peuvent avoir plusieurs versions composantes.

De même, les instances des classes composantes Moteur.v0 et Moteur.v1 suivent les règles d'exclusivité et de partage définies pour des relations entre classes simples en considérant que plusieurs versions d'une même classe composée peuvent avoir la même classe composante.

Par exemple, les instances de la classe Moteur.v0 respectent les règles définies pour les liens de composition exclusifs (1-1) en autorisant que les versions composées auxquelles elles sont rattachées puissent appartenir à plusieurs versions d'une même classe. Par exemple, la version d'objet CFM.v1 peut composer plusieurs versions de la même hiérarchie A320, ces versions appartenant à plusieurs versions de classe ; par contre, CFM.v1 ne peut composer d'autres versions appartenant à d'autres hiérarchies ni dans Avion.v0, ni dans Avion.v1. La version d'objet VT.v0 se comporte de la même façon.

La version de classe Moteur.v1 n'étant liée qu'à une seule version de classe (Avion.v2), ses instances suivent les règles définies pour deux classes simples de versions liées par un lien de composition partagé. Par exemple, la version d'objet CFM.v2 doit composer au moins une version d'objet de la classe Avion.v2 (ex. A320.v3) et peut composer plusieurs versions de plusieurs hiérarchies (ex. A340.v0).

Relations entre des versions de classes et des classes simples

Au niveau des instances, les cas de composition et d'association avec des versions seulement pour l'une des classes correspondent à des cas particuliers de liens entre versions de classes. En effet, une version de classe correspond à une classe simple ; pour les instances, les liens entre une classe simple et plusieurs versions d'une classe correspondent aux liens entre une version d'une classe et plusieurs versions d'une autre classe.

4. Le modèle dynamique intégrant les versions

Le modèle dynamique sur lequel nous nous basons est celui de la méthode OMT (Rumbaugh, 91).

Le modèle dynamique spécifie les séquences admissibles de modification des objets du modèle objet. Il indique les événements échangés entre les objets du système ; il indique également les changements d'états (valeurs attributs et liens) des objets en fonction des événements reçus. Le modèle dynamique est composé de diagrammes d'états, un pour chaque classe du modèle objet.

Contrairement aux objets, les versions d'objets n'ont pas le même comportement face à la réception d'un événement. Une version d'objet dans un état courant se comporte comme un objet tant qu'elle ne reçoit pas d'événement la faisant passer dans l'état gelé (gel ou dérivation). A l'inverse, une version d'objet dans un état gelé et qui possède des dérivées ne réagit à aucun événement la faisant changer d'état c'est-à-dire la faisant passer à l'état en cours ; dans le cas particulier d'une version

gelée qui n'a pas de version dérivée, seuls les événements la faisant passer dans un état courant (dégel) sont acceptés.

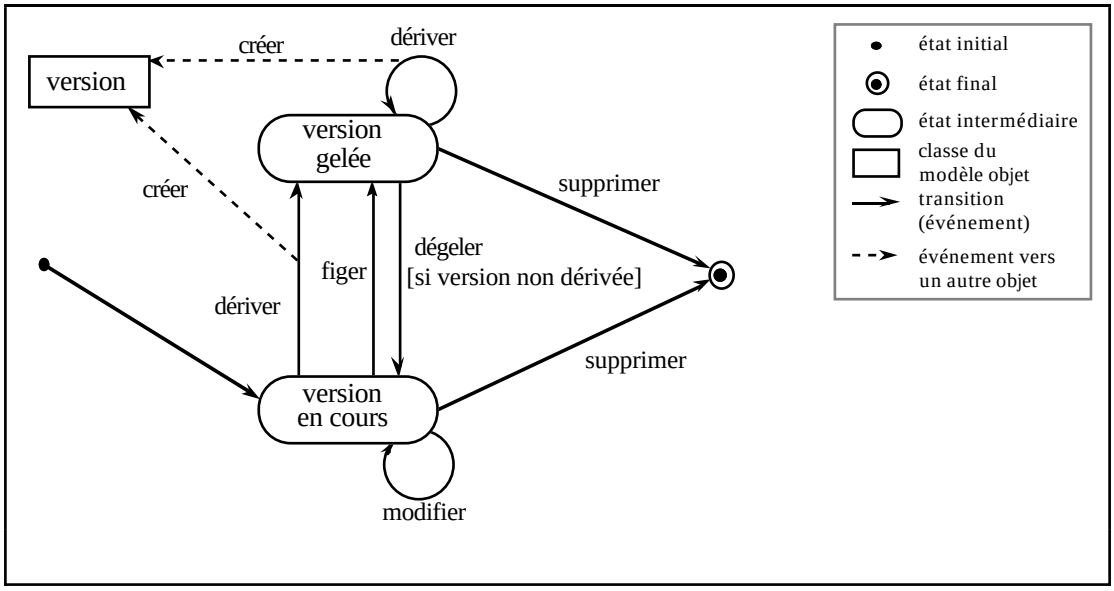
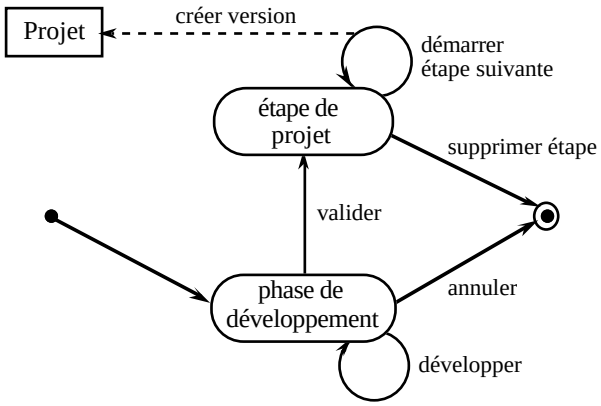


Figure II-30 : Diagramme d'état de plus haut niveau pour une classe de versions

L'état "version en cours" est une généralisation des différents états de l'entité lorsqu'elle est décrite par une version dans un état en cours (cf. § 2.1.). A l'opposé, l'état "version gelée" est une généralisation des différents états de l'entité lorsqu'elle est décrite par une version dans un état gelé (cf. § 2.1.)

Il n'est donc pas nécessaire d'étendre le modèle dynamique par de nouveaux concepts pour prendre en compte la gestion de versions au niveau des instances ; le diagramme d'états correspondant à une classe de versions au plus haut niveau de généralisation doit être analogue à celui décrit précédemment (Andonoff, 96).

Exemple II-15 : Considérons la modélisation de l'évolution des projets (pour lesquels des versions sont gérées). Différentes étapes de projet sont planifiées ; à chaque étape, l'état du projet est conservé (version). Une étape est atteinte après validation ; l'étape suivante ne peut débuter qu'après validation de l'étape précédente. Le modèle dynamique dérivant l'évolution d'un projet est le suivant :



Pour l'intégration de la gestion de versions au niveau des classes, il faut considérer l'ensemble des versions de classes correspondant à une classe générique. Les instances des versions de classes, compte tenu qu'elles décrivent les mêmes entités réelles, ont des structures et des comportements relativement proches avec néanmoins des distinctions. Ainsi, un même diagramme d'états pourra correspondre à plusieurs versions de classe ; dans d'autres cas, deux versions de classe auront deux diagrammes d'états distincts, pouvant néanmoins comporter par endroits quelques similitudes.

En revanche, lorsque l'on gère des versions de classes, les entités sont représentées par une hiérarchie de dérivation de versions s'étendant sur les différentes versions de classes. Il est donc nécessaire de préciser quand s'effectue le passage d'une version de classe à une autre. Ainsi, les différents diagrammes d'états correspondant aux versions de classes, seront regroupés au sein d'un diagramme plus général ; ce diagramme décrit les connexions entre les différents diagrammes des versions de classes, précisant ainsi les événements qui impliquent les passages d'une version de classe à une version suivante (Andonoff, 96).

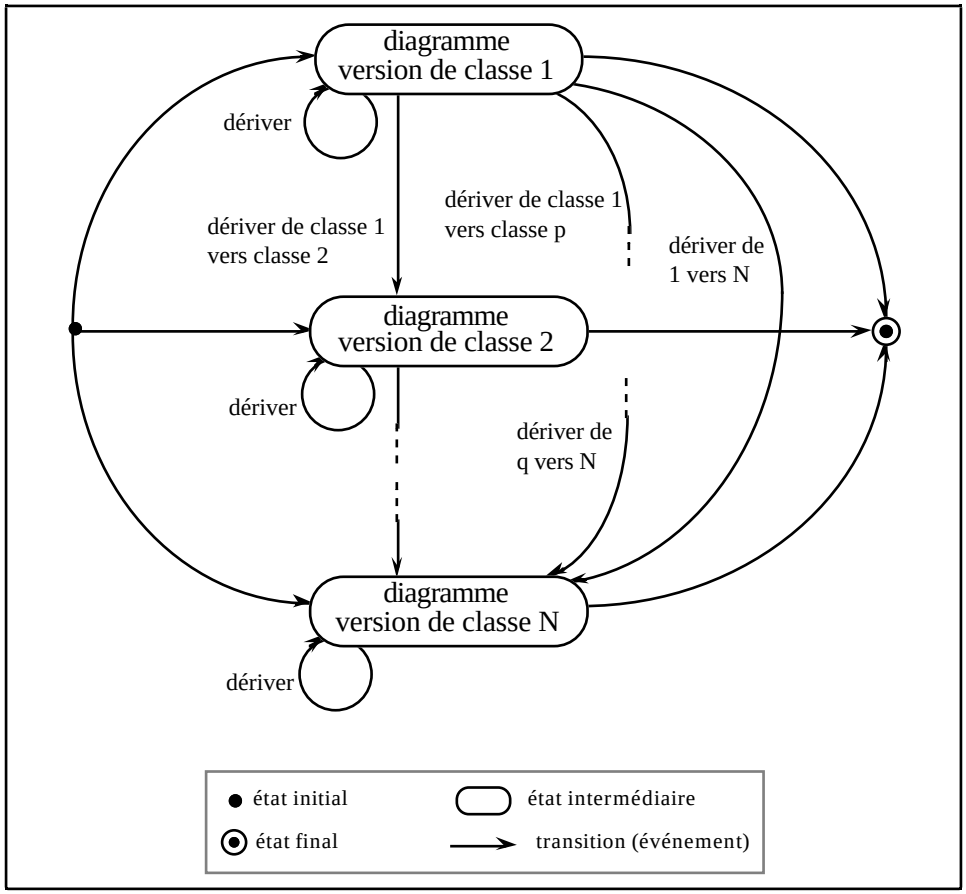
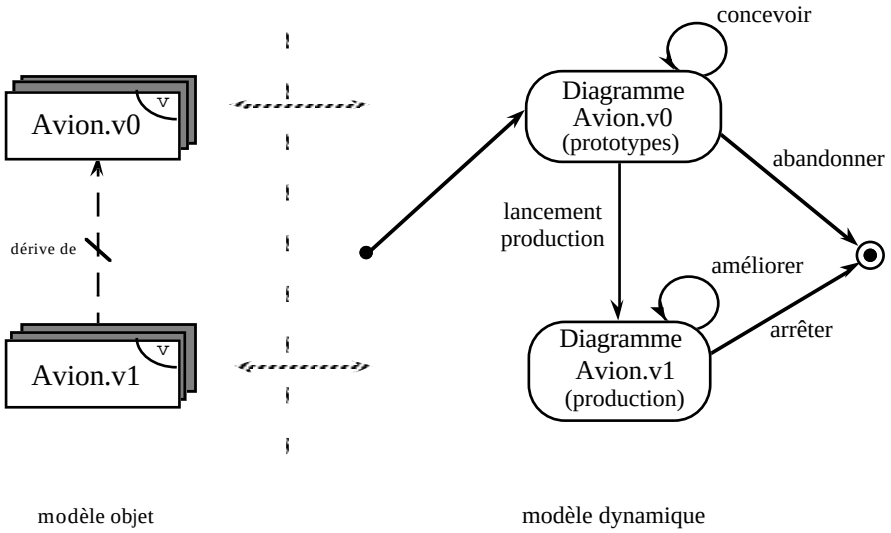


Figure II-31 : Diagramme d'état regroupant les diagrammes des versions d'une classe

Exemple II-16 : Considérons l'évolution d'avions (pour lesquels des versions d'objets et de schéma sont gérées). Deux versions de classes sont gérées : une décrivant les avions au stade de prototype, l'autre décrivant les avions ayant atteint le stade de la production (ajout au niveau du schéma d'informations supplémentaires). Un avion passe de l'état de prototype à l'état d'avion produit après décision du lancement de production. Le modèle dynamique décrivant l'évolution d'un projet est le suivant :



5. Le modèle fonctionnel intégrant les versions

Le modèle fonctionnel de base est celui de la méthode OMT. Il décrit les traitements réalisés par le système c'est-à-dire les entrées et sorties de chaque traitement, ainsi que les stockages et récupération de données.

Le modèle fonctionnel consiste en un ensemble de diagrammes à flots de données qui indiquent les relations fonctionnelles entre les valeurs calculées par un système, y compris les valeurs entrantes, les valeurs sortantes et les réservoirs de données internes. Les bases de données ont souvent un modèle fonctionnel simple, leur objectif étant de stocker et d'organiser les données, pas de les transformer (Rumbaugh, 91).

L'introduction de la gestion de versions implique quelques évolutions à prendre en compte au niveau du modèle fonctionnel ; il ne s'agit pas réellement d'ajouter de nouveaux concepts au modèle fonctionnel d'OMT mais plutôt d'étendre les concepts existants.

Au niveau des traitements, aucune évolution n'est à considérer ; dans tous les cas, les traitements utilisent des données en entrées et produisent des données en sortie.

Compte tenu de la gestion de versions, certains traitements sont identiques d'une version de classe à une autre ou ne comportent que quelques variations. Ainsi, les variations se retrouveront dans la description fonctionnelle ; une seule et même description sera bien sûr utilisée pour des traitements identiques dans plusieurs versions de classes.

Dans le modèle fonctionnel, la gestion de versions d'objets est prise en compte pour les éléments en rapport avec le modèle objet. Ainsi, on distingue les acteurs versionnalisés et les réservoirs de données versionnalisés (Andonoff, 96).

Un acteur versionnalisé est une version d'objet active qui produit ou consomme des valeurs c'est-à-dire un acteur qui est une version d'objet. Le formalisme utilisé pour le modèle objet reste valable pour les acteurs pour lesquels on gère des versions c'est-à-dire :

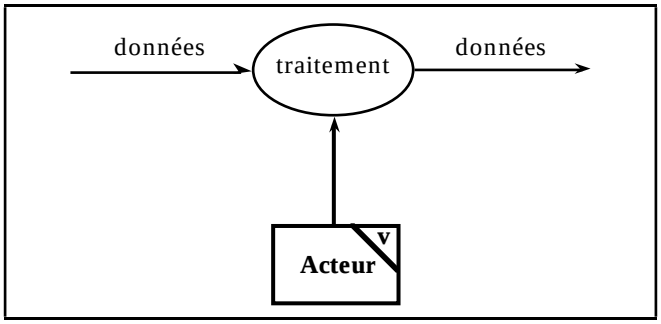


Figure II-32 : Représentation d'acteur versionnalisé

Les réservoirs de données sont des objets passifs qui assurent le stockage de valeurs en vue d'une utilisation. Un réservoir de données versionnalisé stocke des versions d'objets c'est-à-dire un réservoir de données pour versions d'objets. Pour stocker une valeur ponctuelle le réservoir sera de type traditionnel d'OMT. Pour un stockage d'ensembles de valeurs liées temporellement, le réservoir sera de type versionnalisé ; il est représenté comme suit :

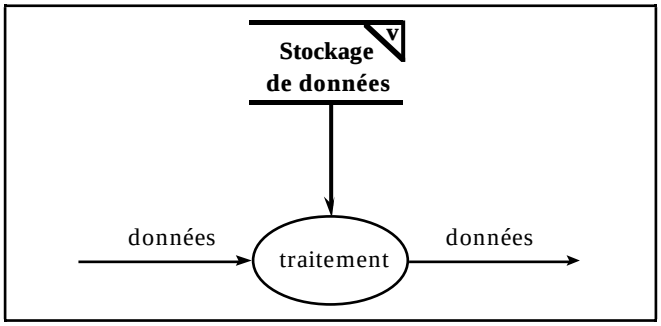
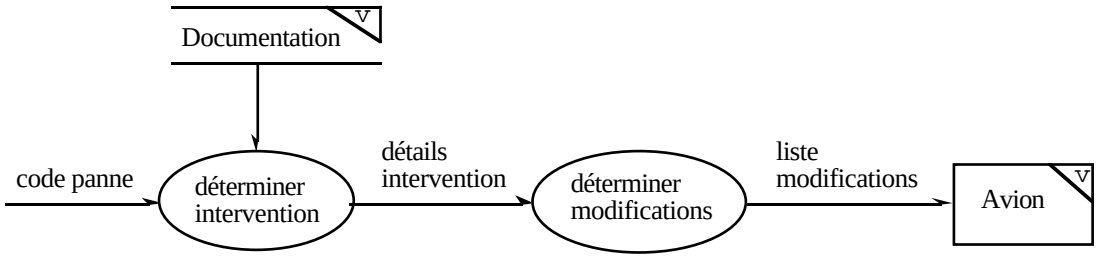


Figure II-33 : Représentation de réservoir de données versionnalisé

Exemple II-17 : Considérons la description fonctionnelle du traitement d'une panne sur un avion. A partir d'un code de panne, on détermine à l'aide de la documentation l'intervention à effectuer ; puis suivant les détails de l'intervention, on évalue les modifications à apporter à l'avion.



Les éléments du modèle objet documentation et avion étant versionnalisés, on représente un réservoir de données versionnalisé et un acteur versionnalisé.

6. Bilan

Le modèle que nous proposons dans ce chapitre est un modèle conceptuel intégrant le concept de versions.

Actuellement, certains systèmes offrent des fonctionnalités pour gérer des versions, lors de l'implantation d'une base de données. Néanmoins, ils n'offrent pas un moyen simple de décrire des bases intégrant des versions.

Notre modèle décrit les versions au niveau le plus élémentaire c'est-à-dire au niveau de chaque entité représentée dans une base de données. D'autres travaux considèrent la gestion de versions de bases entières (Gançarski, 94a) ou la gestion de parties de bases comme dans le système O2 (O2, 96). Notre modèle permet de décrire et manipuler l'évolution de chaque entité représentée, de manière indépendante.

Ce modèle s'appuie les principaux concepts liés aux versions comme le principe de dérivation. Il propose la gestion simultanée de version d'objets et de versions de classes. Il permet ainsi de modéliser différentes catégories de classes pour décrire :

- la gestion simultanée de versions de classes et de versions d'objets,
- la gestion de versions d'objets sans gestion de versions de classes,
- la gestion de versions de classes sans gestion de versions d'objets,
- le gestion d'objets traditionnelle (aucune gestion de version).

Ce modèle est fondé sur les principaux concepts des modèles conceptuels orientés objet, notamment celui de la méthode OMT. Il permet de représenter avec précision

et à l'aide de versions, l'évolution d'entités complexes telles que des documentations techniques. Le modèle définit des relations de composition et d'association entre les différentes catégories d'instances, c'est-à-dire les versions d'objets et les objets. Il est par exemple possible d'associer des classes d'objets avec des classes de versions.

La sémantique des relations est affinée à l'aide de cardinalités associées aux classes liées.

La gestion de versions au niveau chaque entité implique la définition complexe des contraintes d'intégrité inhérentes aux relations de composition et d'association et des cardinalités associées. Nous avons défini les contraintes inhérentes à la définition de liens de composition et d'association quelles que soient les instances liées (objet ou versions d'objets). Les solutions pour les liens entre versions d'objets permettent de décrire fidèlement l'évolution des entités du monde réel et de leurs relations ; les solutions permettent de tenir compte de la stratégie qui consiste à partager les versions d'objets. De plus, des solutions sont proposées pour l'expression contraintes sous-jacentes aux relations entre objets et versions ; à notre connaissance, ce cas n'a jamais été évoqué. Les solutions résolvent les difficultés pour décrire avec précision les relations entre des entités pour lesquelles on ne conserve qu'un état (objet) et des entités pour lesquelles on conserve plusieurs états (versions d'objets).

Ces contraintes sur les instances entraînent des règles suivies par les opérations sur les instances et sur les classes pour maintenir la cohérence de la base.

Néanmoins, ce modèle peut être étendu pour permettre la définition de contraintes d'intégrité supplémentaires sur les versions. Par exemple, un concepteur pourra contraindre le nombre d'alternatives qui peuvent être définies pour les versions instances d'une classe.

Notre modèle conceptuel intégrant les versions sert de base à la conception de l'interface graphique VOHQL (Le Parc, 96a) pour la manipulation de bases de données orientées objet gérant des versions ; cette interface est destinée à des utilisateurs occasionnels (présentée succinctement au Chapitre IV. § 3.4.).

CHAPITRE III.

Le langage de description et de manipulation VOSQL

Chapitre III. : Le langage de description et de manipulation VOSQL

Table des matières

1. Objectifs

Le modèle conceptuel de données défini au chapitre précédent permet de modéliser une base de données intégrant des versions. Il sert également de support pour la manipulation d'une telle base. Ce chapitre décrit le langage de description et de manipulation VOSQL (Version and Object Structured Query Language) que nous proposons.

Actuellement, les systèmes de gestion de versions ne proposent pas de réelles possibilités pour l'interrogation des versions ; par exemple, le système Presage (Talens, 94) ne fournit pas de langage d'interrogation et le système Aristote (Adiba, 89) ne permet d'interroger que d'objets. Les versions peuvent être interrogées comme de simples objets sans considérer leur sémantique spécifique c'est-à-dire sans tenir compte de leur organisation en hiérarchies de dérivation.

Nous proposons un langage qui intègre les spécificités liées aux versions d'objets et aux versions de classes. Il permet :

- la manipulation de schéma, c'est-à-dire les opérations (création, modification, ...) sur les classes et les liens ; ces opérations incluent celles qui s'appliquent aux versions de classes.
- la manipulation des instances c'est-à-dire les opérations applicables aux instances (création, suppression, ...) y compris celles spécifiques aux versions d'objets (dérivation, gel, ...), et surtout l'interrogation.

L'interrogation est de type Select From Where. Elle prend en compte la sémantique propre aux versions lors de leur interrogation. Elle permet de distinguer les différents niveaux d'abstraction liés aux versions que sont les forêts de dérivation, les arbres de dérivation et les versions d'objets. De plus, ce langage permet d'interroger de manière uniforme les objets et les versions. Il reprend les principales fonctionnalités des langages d'interrogation limités aux objets de systèmes tels que ORION et O2. Le langage est associé à une algèbre qui fait l'objet d'autres travaux au sein de notre équipe (Le Parc, 96b).

2. Manipulation de schéma

La manipulation de schéma sous-entend :

- la création des classes et des liens,
- les opérations de mise à jour d'un schéma existant.

Outre les opérations applicables à tous les types de classes (création, modification, suppression), il est nécessaire de définir les opérations relatives à la manipulation de versions de classes (dérivation, gel, dégel, ...).

2.1. Création des classes et des liens entre classes

Classes

La déclaration de création d'une classe spécifie :

- le nom de la classe créée (unique pour la base de données),
- l'évolutivité de la classe (versions de classe ou classe simple),
- le type d'instances qu'elle contient (objets ou versions),
- la position dans le graphe d'héritage (noms des super-classes),
- la structure (ensemble d'attributs),
- le comportement (ensemble de méthodes).

Le squelette de la déclaration de création d'une classe est le suivant :

opération ::= **create** [catég_classe] *nom_de_classe*
[**contains** catég_instance]
[**inherits** *classe* {**,** *classe*}]
[**attributes (** *attribut* : *domaine* {**,** *attribut* : *domaine* }**)**]
[**methods** *méthode (Signature)* {**,** *méthode (Signature)*}**)**

catég_classe ::= **simple class** | **class version**
catég_instance ::= **objects** | **versions**

Les mots-clés **class version** (option par défaut) indiquent que différentes versions de la classe pourront être définies. La déclaration spécifiée correspond à la création de la première version de la classe (implicitement la version 0). Une classe **simple** n'autorise pas la gestion de versions de la classe.

La clause **contains** précise la nature des instances de la classe. L'option **versions** indique qu'il s'agit d'une classe de versions d'objets (option par défaut). L'option **objects** limite les instances à de simples objets.

La clause **inherits** précise la position de la nouvelle classe dans le graphe d'héritage.

La structure est spécifiée, dans la clause **attributes**, par la liste des attributs.

Le comportement est précisé dans la clause **methods** par une liste de méthodes définies par leur nom et leur signature. Le corps des méthodes est défini à part, en accord avec le SGBD d'implantation.

Liens

Les liens sont définis indépendamment de la création des classes ; ils sont créés après les classes qu'ils relient.

Un **lien de composition** est défini entre une classe composée et une classe composante. La déclaration de création d'un lien de composition est la suivante :

Composition [*nom_compo*] : *classe_composée* **(**a,b**) of** *classe_composante* **(**1,c**)**

le lien de composition peut éventuellement ne pas être nommé,
la cardinalité (a,b) pour la classe composée est telle que a ∈ {0,1} et b ∈ {1,N}
la cardinalité (1,c) pour la classe composante traduit les notions d'exclusivité et de partage ; c=1 traduit l'exclusivité.

Un **lien d'association** est défini entre deux classes sans orientation. La déclaration de création d'un lien d'association est la suivante :

Relationship *nom_association* : *classe1* **(**a,b**) and** *classe2* **(**c,d**)**

le lien d'association est obligatoirement nommé,
les cardinalités pour les classes sont telles que a,c ∈ {0,1} et b,d ∈ {1,N}.

Exemple III-1 : Considérons la définition d'un schéma décrivant des documents ; un document est constitué de chapitres, chacun d'eux pouvant être à son tour constitué d'autres chapitres. On désire pouvoir conserver différentes évolutions du schéma ; on gérera donc des versions pour les deux classes. La spécification du schéma est la suivante :

create class *Document*
attributes (*désignation* : *string***)**

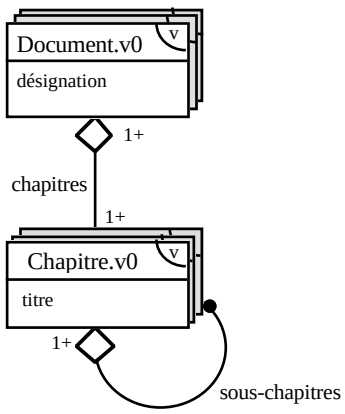
-- sans les options par défaut
create class version *Document*
contains versions
attributes (*désignation* : *string***)**

create class *Chapitre*
attributes (*titre* : *string*,
*texte : Text***)**

-- sans les options par défaut
create class version *Chapitre*
contains versions
attributes (*titre* : *string, texte : Text***)**

Composition chapitres : *Document* **(**1,N**) of** *Chapitre* **(**1,N**)**
Composition sous-chapitres : *Chapitre* **(**0,N**) of** *Chapitre* **(**1,N**)**

La déclaration fournit la base suivante :



2.2. Dérivation des classes et des liens

Classes

La dérivation d'une classe consiste à créer une version de classe à partir d'une version de classe existante dont elle reprend le schéma. Le concepteur spécifie les modifications éventuelles apportées à la nouvelle version de classe. La nouvelle version de classe est créée sans instance. La dérivation de classe ne s'applique qu'aux classes pour lesquelles on gère des versions.

La déclaration de création d'une version de classe dérivée est la suivante :

opération ::= **create class derived from** *classe*.*version*
[**inherits** [**add** *classe* {, *classe*}]
[**delete** *classe* {, *classe*}]]
[**attributes** [**add** *attribut* : *domaine* {, *attribut* : *domaine*}]
[**delete** *attribut* {, *attribut* : *domaine*}]
[**replace** *attribut* **into** *attribut* : *domaine*
{, *attribut* **into** *attribut* : *domaine*}]]
[**methods** [**add** *méthode*(signature) {, *méthode*(signature)}]
[**delete** *méthode* {, *méthode*}]]
[**replace** *méthode* **into** *méthode* (signature)
{, *méthode* **into** *méthode* (signature)}]]

La clause **derived from** indique la version de classe initiale ; le numéro de version de la nouvelle classe est automatiquement ajouté. Si le numéro de version de la classe dérivée est omis, l'opération de dérivation s'applique à la version de classe par défaut.

La clause **inherits** spécifie l'ajout ou la suppression de super-classes par rapport à la version de classe précédente. Les clauses **attributes** et **methods** indiquent respectivement les modifications de structure et de comportement.

Liens

La dérivation des liens entre classes est implicite ; elle est effectuée lorsque les classes reliées sont dérivées. Lorsqu'une classe, liée à d'autres classes, est dérivée, la nouvelle version de classe conserve les liens avec les mêmes classes. Si l'on veut dériver plusieurs classes en même temps, en retrouvant les mêmes liens entre les nouvelles versions de classes, il faut regrouper les dérivations de classes au sein d'une transaction. A la fin d'une transaction, le système évalue les liens entre les classes dérivées simultanément et il dérive ces liens pour lier les nouvelles versions de classes définies.

Exemple III-2 : En reprenant l'exemple précédent, considérons que l'on fasse évoluer la structure des documents ainsi que celle des chapitres ; les documents restent constitués de chapitres. Une nouvelle version des deux classes Document et Chapitre est dérivée. Le lien de composition est conservé entre les nouvelles versions des classes.

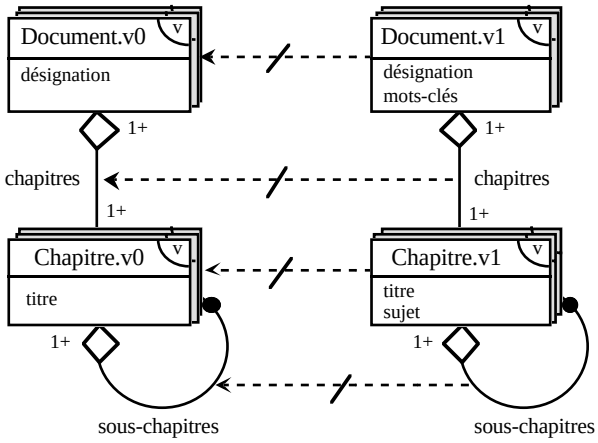
begin transaction

create class derived from *Document.v0*
attributes add *mots-clés* : *set(string)*

create class derived from *Chapitre.v0*
attributes add *sujet : string*

end transaction

Les déclarations fournissent la base suivante :



2.3. Modification et suppression des classes et des liens

Modification d'une classe

La modification du schéma d'une classe consiste en la modification :

- de l'héritage (ajout, ou suppression de super-classes) d'une classe,
- de la structure (ajout, suppression ou modification d'attribut),
- du comportement (ajout, suppression ou modification de méthode).

Pour les classes de versions d'objets, la modification ne peut être appliquée que sur des classes sans version d'objet gelée. La modification du schéma d'une classe contenant des instances implique leur adaptation au nouveau schéma de la classe.

L'opération de modification d'une classe est la suivante :

opération ::= **update class** *classe*[*.version*]
[**inherits** [**add** *classe* {**,** *classe*}]
[**delete** *classe* {**,** *classe*}]]

[**attributes** [**add** *attribut* **:** *domaine* {**,** *attribut* **:** *domaine*}]
[**delete** *attribut* {**,** *attribut* **:** *domaine*}]
[**replace** *attribut* **into** *attribut* :*domaine*
{**,** *attribut* **into** *attribut* : *domaine*}]]
[**methods** [**add** *méthode***(**signature**)** {**,** *méthode***(**signature**)**}]
[**delete** *méthode* {**,** *méthode*}]]
[**replace** *méthode* **into** *méthode* **(**signature**)**
{**,** *méthode* **into** *méthode* **(**signature**)**}]]

La clause **inherits** spécifie l'ajout ou la suppression de super-classes par rapport à la version de classe précédente. Si la classe possède déjà des instances, l'ajout de liens de composition et d'association au travers d'une super-classe est autorisé ; ceux-ci ne doivent pas induire de contrainte non respectée par les instances existantes.

La clause **attributes** concerne les modifications de structure c'est-à-dire l'ajout, la modification et la suppression d'attribut. Pour les instances déjà existantes dans la classe, les attributs ajoutés sont initialisés à la valeur nulle. La modification d'attribut consiste à renommer l'attribut, restreindre son domaine ou élargir ce dernier ; les attributs dont le domaine est restreint sont réinititalisés à la valeur nulle pour les instances dont la valeur existante n'est plus valide.

La clause **methods** concerne les modifications de comportement c'est-à-dire l'ajout, la suppression et la modification de méthode.

Suppression d'une classe

La suppression d'une classe n'est possible que lorsqu'elle ne possède pas d'instance. L'opération de suppression est applicable à tous les types de classes. Elle implique la réorganisation de la hiérarchie d'héritage, lorsque celle-ci est super-classe. Dans le cas des versions de classes, la suppression d'une version de classe implique la réorganisation de la forêt de dérivation des versions de classes à laquelle elle appartient. Elle implique de plus, la suppression des liens de composition et d'association avec les autres classes.

L'opération de suppression d'une classe est définie comme suit :

delete class *nom_de_classe*[.*version*]

Modification d'un lien

La modification d'un lien de composition ou d'association consiste en la modification des cardinalités. Pour les classes de versions possédant déjà des instances, la modification d'un lien de composition ou d'association n'est possible que si elles ne possèdent pas de version d'objet gelée. Un renforcement des

cardinalités des liens n'est possible que si les instances existantes respectent les nouvelles contraintes.

La déclaration de modification d'un lien de composition ou d'association est identique à la déclaration de création ; la redéclaration de création d'un lien déjà existant remplace la précédente déclaration.

Composition [*nom_compo*] : *classe_composée* **(**a**,**b**) of** *classe_composante* **(**1**,**c**)**

Relationship *nom_association* : *classe1* **(**a**,**b**) and** *classe2* **(**c**,**d**)**

Suppression d'un lien

La suppression d'un lien de composition ou d'association implique la disparition de tous les liens entre les instances des deux classes ; celles-ci sont conservées dans chacune des deux classes. La déclaration de suppression d'un lien de composition et de suppression d'un lien d'association sont les suivantes :

Delete Composition [*nom_compo*] : *classe_composée* **of** *classe_composante*

Delete Relationship *nom_association* : *classe1* **and** *classe2*

2.4. Gel, dégel, version de classe par défaut

Trois opérations supplémentaires sont définies uniquement pour les classes avec versions :

- le gel d'une version de classe qui entraîne l'interdiction d'appliquer l'opération de modification sur cette version de classe,

  freeze class *classe.version*

- le dégel d'une version de classe qui ne peut s'appliquer qu'à une version de classe "feuille" c'est-à-dire qui n'a pas de version de classe dérivée,

  unfreeze class *classe.version*

- la désignation de la version de classe par défaut dans une hiérarchie de dérivation de classes ; la version par défaut est celle proposée a priori à l'utilisateur pour une classe. La déclaration d'une version par défaut pour une classe annule la déclaration précédente,

  default class *classe.version*

3. Manipulation des instances

Les opérations sur les instances sont définies de manière uniforme à celles applicables aux classes. Les opérations de création, modification et suppression sont définies pour tous les types d'instances ; en revanche, les opérations de dérivation, de gel, de dégel et de désignation d'une version par défaut sont spécifiques aux

versions d'objets. De plus, l'opération de migration est définie pour les instances de versions de classes.

3.1. Création d'instance

La création d'instance consiste en :

- la création d'un objet pour les classes d'objets,
- la création d'une version racine pour les classes de versions.

Une valeur peut être attribuée à chacun des attributs. La forme de la déclaration de création d'une instance dans une classe est la suivante :

create instance *nom* **in** *classe*[**.***version*]
[**values (** *attribut* **:** *valeur*{**,** *attribut* **:** *valeur*}**)**]

Le nom sert de descripteur de l'instance. Pour les objets, il est unique pour chaque objet ; pour les versions, il est identique pour toutes les versions d'objet décrivant la même entité du monde réel. Le descripteur peut être associé à un attribut. Pour les versions d'objets, la création correspond à la définition d'une version racine décrivant l'entité ; le numéro de version est automatiquement attribué. Lorsque le nom de l'instance est déjà présent dans la classe, la déclaration correspond à la création d'une alternative de premier niveau c'est-à-dire la création d'une version racine alternative (cf. Chapitre II § 2).

Pour les instances complexes, les liens entre instances sont spécifiés après création des instances.

composition instance [*nom_compo*] **:** *nom1* **from** *classe1*
add *nom2* **from** *classe2*

relationship instance [*nom_asso*] **:** *nom1* **from** *classe1*
add *nom2* **from** *classe2*

La vérification des contraintes sur les instances est a priori effectuée après chaque création d'instance. Or, pour respecter certaines contraintes inhérentes à la composition ou à l'association, il faut pouvoir créer simultanément les instances liées (cf. Chapitre II. § 3.7.2.2.2. et 3.7.3.2.). Pour cela, on regroupe les créations simultanées au sein d'une transaction (délimitée par **begin transaction** et **end transaction**). La vérification des contraintes est suspendue pendant une transaction ; elle ne s'effectue qu'en fin de la transaction.

Exemple III-3 : Considérons la classe de version Document.v0 définie dans l'exemple III-1 comme suit :

create class *Document*
attributes (*désignation* : *string***)**

create class *Chapitre*
attributes (*titre* : *string, texte : Text***)**

Composition chapitres : *Document* **(**1**,**N**) of** *Chapitre* **(**1**,**N**)**
Composition sous-chapitres : *Chapitre* **(**0**,**N**) of** *Chapitre* **(**1**,**N**)**

Considérons la création suivante d'un document constitué de trois chapitres dont un est lui-même constitué de deux sous-chapitres :

begin transaction

-- création de la version v0 de Doc dans la classe Document
create instance *Doc* **in** *Document*
values (*désignation* : "Documentation")

-- création de la version v0 de Intro dans la classe Chapitre
create instance *Intro* in *Chapitre*
values (*titre* : "introduction", *texte* : intro.text)

-- création de la version v0 de Desc dans la classe Chapitre
create instance *Desc* **in** *Chapitre*
values (*titre* : "description", *texte* : desc.text)

-- création de la version v0 de Bilan dans la classe Chapitre
create instance *Bilan* in *Chapitre*
values (*titre* : "bilan", *texte* : bilan.text)

composition instance chapitres : *Doc* **from** *Document*
add *Intro.v0, Desc.v0, Bilan.v0* **from** *Chapitre*

end transaction

Compte tenu des cardinalités fixées pour les liens de composition, les créations des chapitres *Intro*, *Desc* et *Bilan* doivent être réalisées durant la même transaction que la création du document *Doc*.

La cardinalité minimum de 0 pour la composition de chapitres en sous-chapitres autorise à effectuer les créations de sous-chapitres dans des transactions distinctes.

begin transaction

-- création de la version v0 de Part1 dans la classe Chapitre
create instance *Part1* **in** *Chapitre*
values (*titre* : "partie1", *texte* : part1.text)

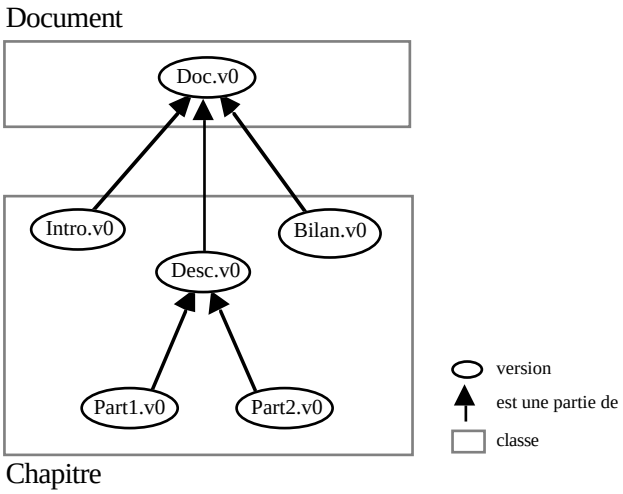
-- création de la version v0 de Part2 dans la classe Chapitre
create instance *Part2* **in** *Chapitre*
values (*titre* : "partie2", *texte* : part2.text)

composition instance sous-chapitres : *Desc* **from** *Chapitre*
add *Part1.v0, Part2.v0* **from** *Chapitre*

end transaction

Cette deuxième partie peut néanmoins être intégrée dans la première déclaration.

Ces déclarations fournissent :



3.2. Modification et suppression d'instance

Modification d'instance

La modification d'instance concerne la mise à jour de la valeur d'un ou plusieurs attributs d'un objet ou d'une version d'objet ; chaque attribut modifié doit être spécifié avec la nouvelle valeur correspondante. Dans le cas des versions d'objets, seules les versions non gelées peuvent être modifiées.

La déclaration de modification d'une instance est la suivante :

update instance *nom*[**.***version*] **in** *classe*[**.***version*]
[**values (** *attribut* **:** *valeur* {**,** *attribut* **:** *valeur*}**)**]

Suppression d'instance

La suppression d'instance peut concerner un objet, une version ou toute une forêt de dérivation. Pour les versions, la suppression d'une version implique la réorganisation de la forêt de dérivation. Lorsqu'aucun numéro de version n'est spécifiée, toute la forêt de dérivation est supprimée.

delete instance *nom.*[*version*] **in** *classe*[**.***version*]

Les liens (composition et association) entre instances sont modifiés comme suit :

composition instance [*nom_compo*] **:** *nom1*[*.version*] **from** *classe1*[*.version*]
add *nom2*[*.version*] **from** *classe2*[*.version*]
replace *nom2*[*.version*] **with** *nom3*[*.version*] **from** *classe2*[*.version*]
delete *nom2*[*.version*] **from** *classe2*[*.version*]

relationship instance [*nom_asso*] **:** *nom1*[*.version*] **from** *classe1*[*.version*]
add *nom2*[*.version*] **from** *classe2*[*.version*]
replace *nom2*[*.version*] **with** *nom3*[*.version*] **from** *classe2*[*.version*]
delete *nom2*[*.version*] **from** *classe2*[*.version*]

3.3. Dérivation et migration d'instance

Dérivation d'instance

La dérivation consiste à définir une nouvelle version d'objet à partir d'une version d'objet déjà existante ; les deux versions décrivent la même entité. Une différence de valeur entre les deux versions est généralement spécifiée mais n'est pas obligatoire. La dérivation d'instance ne concerne que les classes de versions d'objets. La déclaration de versions alternatives est implicite ; deux versions créées par dérivation de la même version d'objet existante sont des alternatives.

La déclaration d'une version d'objet dérivée est la suivante :
create instance in *classe*
derived from *nom*[**.***version*]
[**values (** *attribut* **:** *valeur* {**,** *attribut* **:** *valeur*} **)**]

La nouvelle version conserve les liens avec les mêmes instances que la version précédente. Les modifications de liens sont effectuées comme défini précédemment :
composition instance [*nom_compo*] **:** *nom1*[*.version*] **from** *classe1*[*.version*]
add *nom2*[*.version*] **from** *classe2*[*.version*]
replace *nom2*[*.version*] **with** *nom3*[*.version*] **from** *classe2*[*.version*]
delete *nom2*[*.version*] **from** *classe2*[*.version*]

relationship instance [*nom_asso*] **:** *nom1*[*.version*] **from** *classe1*[*.version*]
add *nom2*[*.version*] **from** *classe2*[*.version*]
replace *nom2*[*.version*] **with** *nom3*[*.version*] **from** *classe2*[*.version*]
delete *nom2*[*.version*] **from** *classe2*[*.version*]

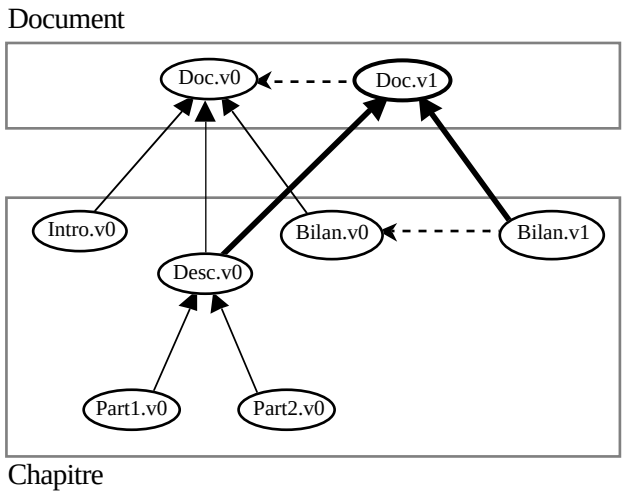
La vérification des contraintes et ses conséquences (comme la propagation des dérivations) est effectuée après chaque dérivation. Lorsque l'on veut dériver simultanément des versions liées (ex. une version composée et une version composante), il faut regrouper les dérivations dans une transaction. La vérification des contraintes est suspendue pendant la transaction et reprend à la fin de celle-ci. Lorsque deux versions liées sont dérivées dans une transaction (ex. une version composée dérivée avec une version composante), leur lien est automatiquement répercuté entre les 2 nouvelles versions définies (ex. la nouvelle version composée est liée à la nouvelle version composante) ; les liens avec d'autres versions liées non

dérivées dans la transaction sont également conservés (ex. la nouvelle version composée conserve les liens avec les composants non dérivés).

Exemple III-4 : Reprenons les classes Document.v0 et Chapitre.v0 et la version Doc.v0 de document, définies dans les exemples III-1 et III-3. Considérons la création d'une nouvelle version de document à partir de la version de document Doc.v0 ; la nouvelle version du document diffère de la précédente au niveau d'un chapitre dont une nouvelle version est également créée.

```
begin transaction

-- création de Doc.v1 dérivée de Doc.v0 dans la classe Document
create instance in Document
derived from Doc.v0

-- création de Bilan.v1 dérivée de Bilan.v0 dans la classe Chapitre
create instance in Chapitre
derived from Bilan.v0

composition instance chapitres : Doc.v1 in Document
      delete Intro.v0 from Chapitre.v0

end transaction
```

Cette déclaration fournit le résultat suivant :



Migration d'instance

La migration d'instance consiste à changer une instance de classe ; l'instance est "adaptée" au nouveau schéma de classe, c'est-à-dire que les valeurs des attributs dans l'ancienne classe valides dans le nouveau schéma sont conservées.

Nous limitons la migration d'instance aux versions de classes d'une même hiérarchie. En effet, une instance ne peut passer d'une classe à une autre classe qui ne décrirait pas les mêmes entités ; par exemple, une Personne ne peut pas devenir une Documentation. Nous considérons qu'un objet ou une version d'objet peut changer de version de classe mais en restant dans le même arbre de dérivation ; une

Personne reste une Personne mais avec une représentation différente. De plus, la migration d'une instance ne peut s'effectuer que vers une version de classe directement ou indirectement dérivée ; la migration doit suivre l'évolution d'une entité c'est-à-dire qu'elle doit suivre le sens de dérivation des versions de classes.

La déclaration de migration d'une instance est la suivante :

move instance *nom*[**.***version*]
from *classe.version*$_i$ **to** *classe.version*$_j$
values (*attribut* **:** *valeur* {**,** *attribut* **:** *valeur*} **)**

avec *version*$_i$ < *version*$_j$

3.4. Gel, dégel et version d'objet par défaut

Dans le cas des versions d'objets, trois opérations supplémentaires sont applicables : les opérations de gel et dégel de versions et la désignation de versions par défaut. Ces opérations sont analogues à celles définies pour les classes ; elles s'appliquent à une version d'objet donnée appartenant à une classe donnée.

La déclaration de ces opérations est la suivante :

freeze instance *nom.version* **in** *classe*[**.***version*]

unfreeze instance *nom.version* **in** *classe*[**.***version*]

default instance *nom.version* **in** *classe*[**.***version*]

4. Un langage d'interrogation intégrant les versions

Le langage d'interrogation est de type **Select From Where**. Il étend les langages de requêtes SQL objets permettant à la fois l'interrogation d'objets et l'interrogation de versions. Les systèmes de gestion de versions qui proposent un langage de requêtes, comme ORION (Kim, 89b) et Aristote (Fauvet, 92) permettent seulement l'interrogation d'objets (les versions sont interrogées comme de simples objets sans considérer leurs spécificités).

4.1. Principe

Une requête est de la forme **Select From Where**. La clause **Select** définit la forme du résultat. La clause **From** décrit la portée de la requête c'est-à-dire l'ensemble interrogé. La clause **Where** précise les prédicats de la requête.

Selon notre modèle, le résultat et la portée d'une requête peuvent être un ensemble de forêts de versions, un ensemble d'arbres de versions ou un ensemble de versions (cf. Chapitre II § 2.1.). Ces versions d'objets peuvent appartenir à :

- une seule classe pour des classes simples,
- une seule version de classe ou un ensemble de versions de classe d'une hiérarchie de dérivation, lorsqu'il y a gestion de versions de classes.

Les prédicats doivent être vérifiés selon les cas, par une forêt de versions, un arbre de versions ou une version. Pour un prédicat concernant une forêt ou un arbre de versions, les quantificateurs **Exist** et **Forall** sont utilisés ; ils permettent d'obtenir les forêts (ou les arbres) dont au moins un arbre (ou une version) vérifie une condition donnée ou dont tous les arbres (ou toutes les versions) vérifient une condition donnée.

Un ensemble de fonctions peut être utilisé dans l'expression des prédicats, notamment les fonctions associées aux critères internes automatiquement ajoutés pour les classes de versions. La fonction **Entity** peut être utilisée pour une forêt, un arbre ou une version ; elle retourne le descripteur de l'entité décrite par la forêt, l'arbre ou la version. Les fonctions **Date**, **Creator**, **Number**, **Ancestors**, **Descendants**,..., peuvent être utilisées pour une version ; elles retournent respectivement la date de création, le nom du créateur, le numéro, l'ensemble des versions ancêtres et l'ensemble des versions dérivées de la version considérée. Les fonctions booléennes **Frozen** et **Default** sont utilisables pour une version ; elles indiquent respectivement si une version est gelée et si une version est version par défaut. Enfin les attributs, les méthodes, les liens de composition et les liens d'association peuvent également être utilisés pour les prédicats concernant une version.

Les fonctions des langages de requêtes de bases de données (**Min**, **Max**, **Count**, **Distinct**, ...) sont également disponibles. Les fonctions **Forest**, **Unforest**, **Tree** et **Untree** sont définies pour restructurer les ensembles manipulés (ensemble de forêts de versions, ensemble d'arbres de versions ou ensemble de versions).

4.2. Interrogation de classes simples

L'interrogation de classes de versions prend en compte toute la sémantique des versions d'objets. L'interrogation des classes de versions étend l'interrogation de classes d'objets ; elle reprend les principales fonctionnalités des langages d'interrogation des systèmes de gestion de bases de données orientées objet traditionnels tels que ORION et O2.

4.2.1. Base exemple de classes simples

Pour illustrer les différentes requêtes qu'il est possible d'exprimer, nous utiliserons l'exemple de base suivant :

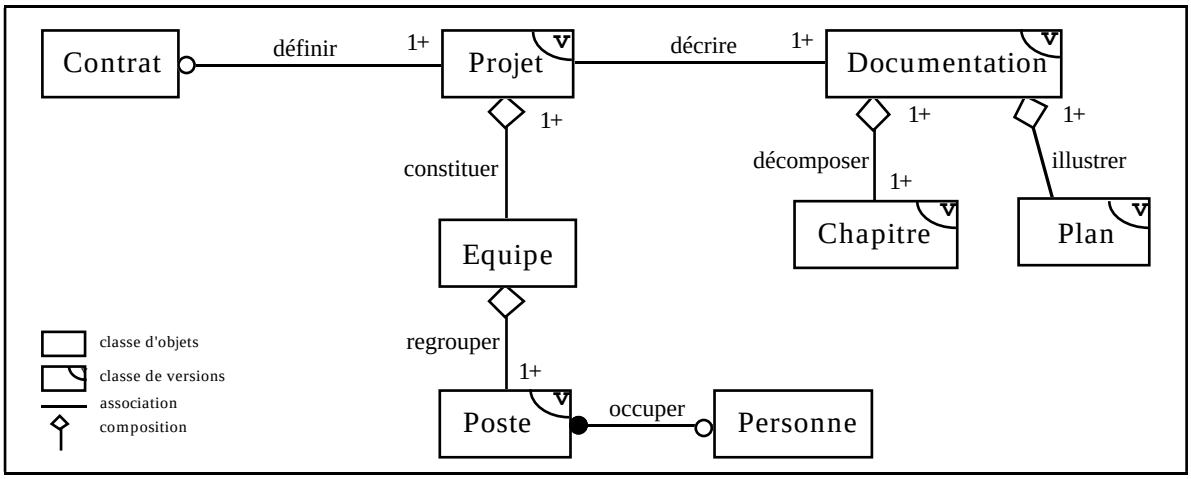


Figure III-1 : Base exemple de classes simples

La spécification de ce schéma dans notre langage est la suivante:

create simple class Projet
contains versions
attributes (code : string,
descriptif : texte,
durée : integer)

create simple class Equipe
contains objects
attributes (numéro : integer,
secteur : string,
activité : string)

create simple class Personne
contains objects
attributes (nom : string,
prénom : string,
date_nais : date,
diplômes : set (string))
methods age() : integer;

create simple class Poste
contains versions
attributes (code : string,
fonction : string,
niveau : integer)

create simple class Documentation
contains versions
attributes (titre : string,
mots-clés : set (string))
methods Compter_mots() : integer;

create simple class Chapitre
contains versions
attributes (titre : string,
contenu : texte)

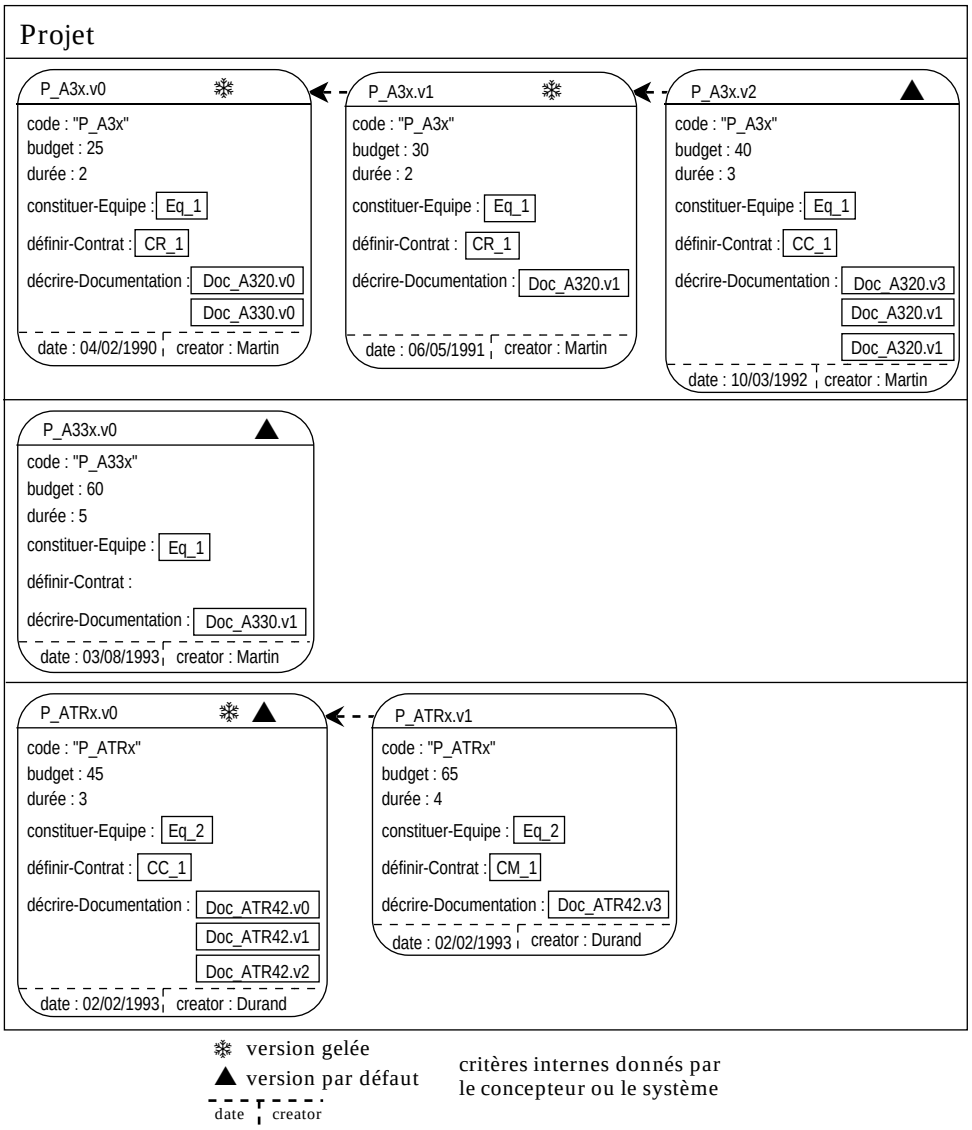
create simple class Contrat
contains objects
attributes (numéro : integer,
catégorie : string)

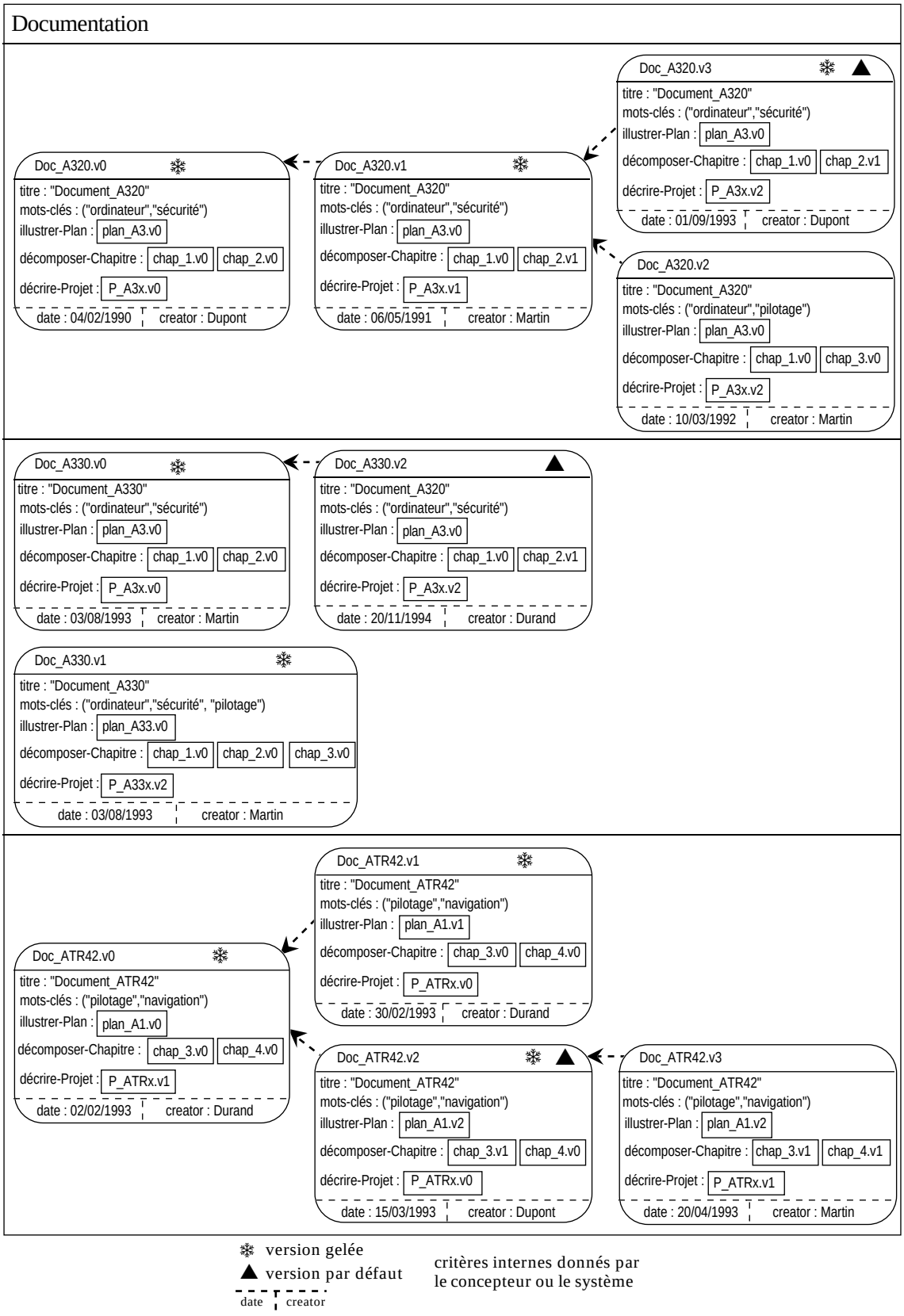
create simple class Plan
contains versions
attributes (nom : string,
descriptif : string,
image : bitmap)

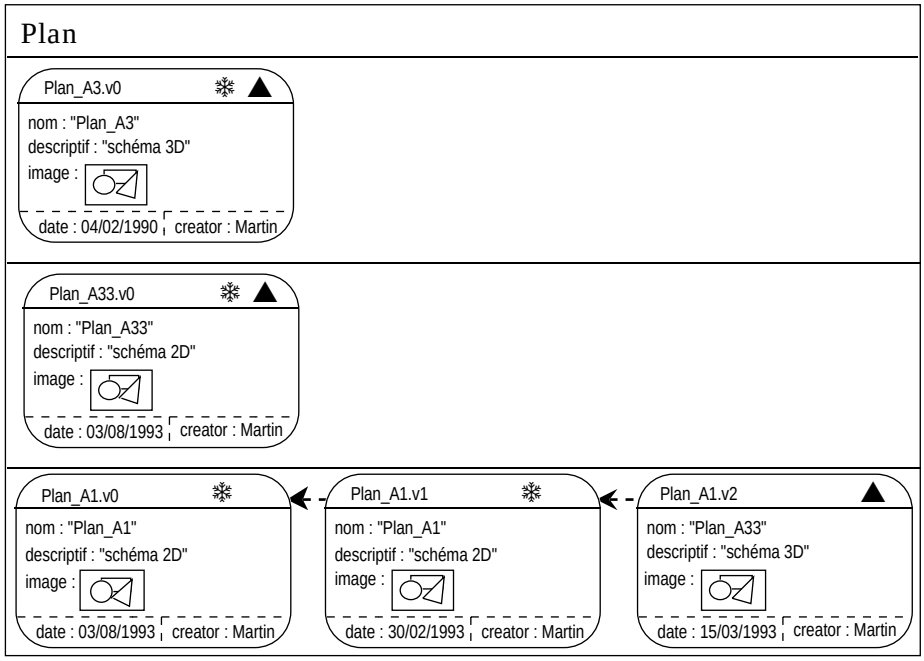
Composition constituer : Projet (1,1) **of** Equipe (1,N)
Composition regrouper : Equipe (1,N) **of** Poste (1,1)
Composition décomposer : Documentation (1,N) **of** Chapitre (1,N)
Composition illustrer : Documentation (1,1) **of** Plan (1,N)

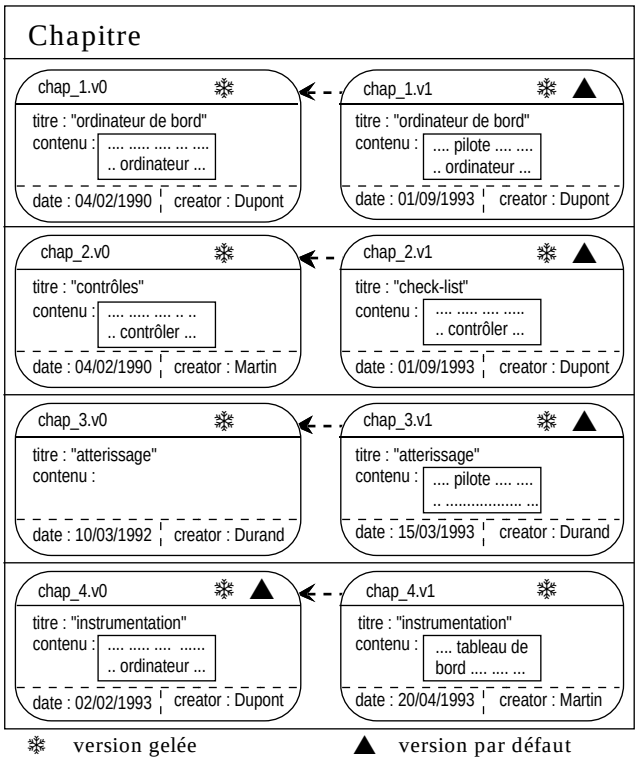
Relationship décrire : Projet (1,N) **and** Documentation (1,1)
Relationship définir : Projet (0,1) **and** Contrat (1,N)
Relationship occuper : Personne (0,1) **and** Poste (0,1)

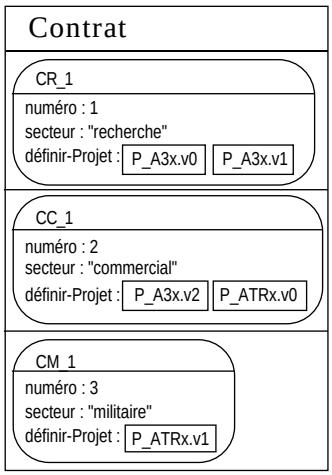
Les instances des classes sont les suivantes :

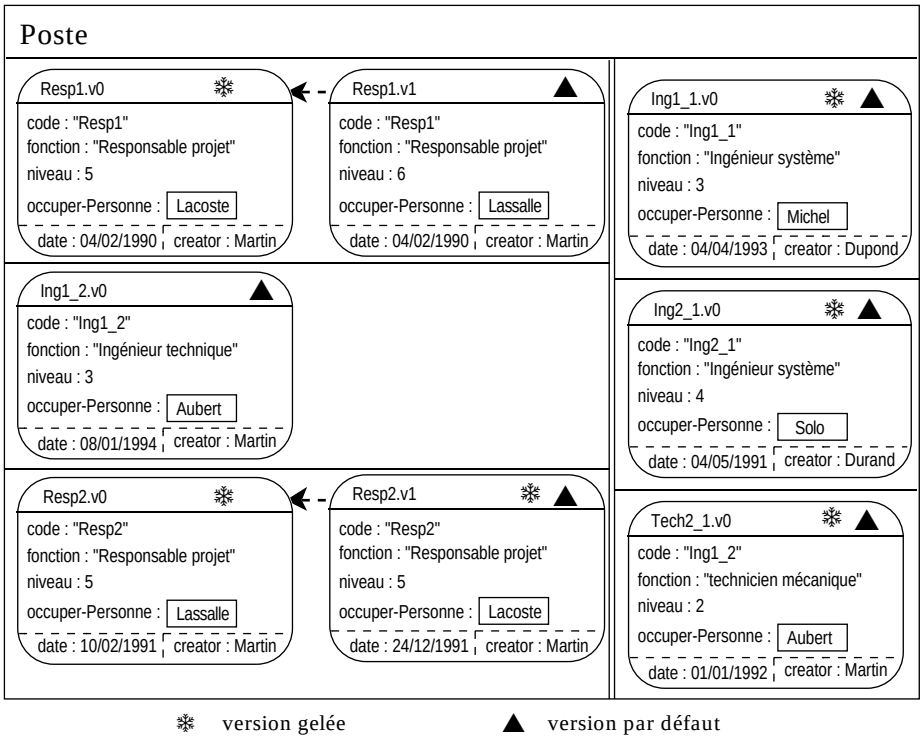


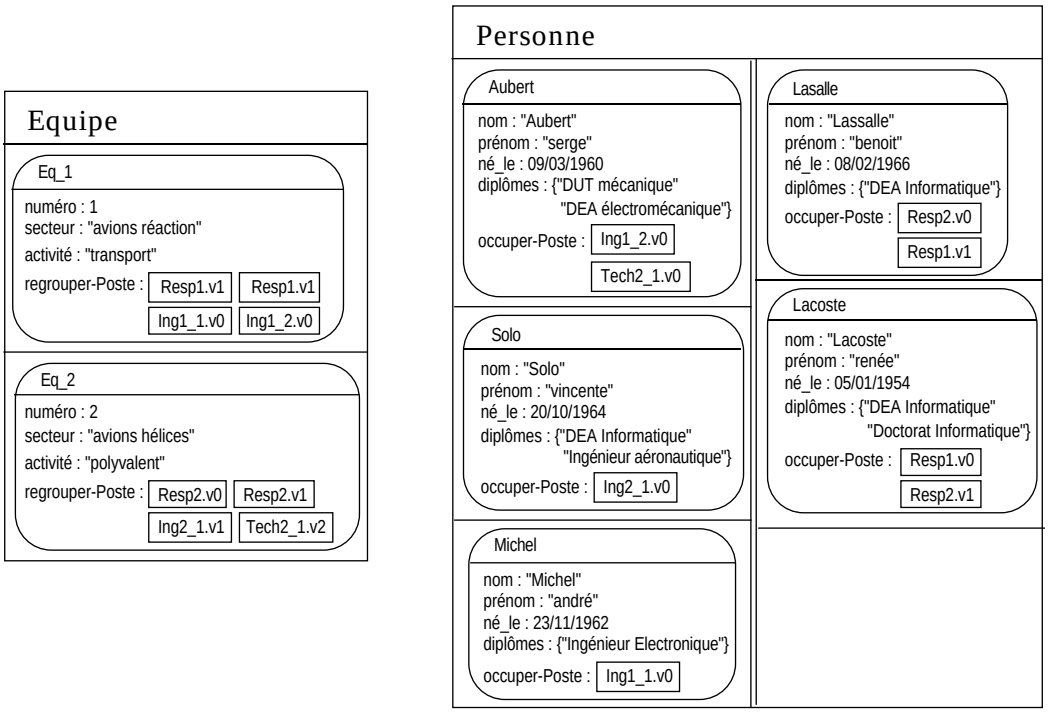

Documentation
Doc_A320.v0
titre : "Document_A320"
mots-clés : ("ordinateur","sécurité")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_2.v0
décrire-Projet : P_A3x.v0
date : 04/02/1990 creator : Dupont
Doc_A320.v1
titre : "Document_A320"
mots-clés : ("ordinateur","sécurité")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_2.v1
décrire-Projet : P_A3x.v1
date : 06/05/1991 creator : Martin
Doc_A320.v3
titre : "Document_A320"
mots-clés : ("ordinateur","sécurité")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_2.v1
décrire-Projet : P_A3x.v2
date : 01/09/1993 creator : Dupont
Doc_A320.v2
titre : "Document_A320"
mots-clés : ("ordinateur","pilotage")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_3.v0
décrire-Projet : P_A3x.v2
date : 10/03/1992 creator : Martin
Doc_A330.v0
titre : "Document_A330"
mots-clés : ("ordinateur","sécurité")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_2.v0
décrire-Projet : P_A3x.v0
date : 03/08/1993 creator : Martin
Doc_A330.v2
titre : "Document_A320"
mots-clés : ("ordinateur","sécurité")
illustrer-Plan : plan_A3.v0
décomposer-Chapitre : chap_1.v0 chap_2.v1
décrire-Projet : P_A3x.v2
date : 20/11/1994 creator : Durand
Doc_A330.v1
titre : "Document_A330"
mots-clés : ("ordinateur","sécurité", "pilotage")
illustrer-Plan : plan_A33.v0
décomposer-Chapitre : chap_1.v0 chap_2.v0 chap_3.v0
décrire-Projet : P_A33x.v2
date : 03/08/1993 creator : Martin
Doc_ATR42.v1
titre : "Document_ATR42"
mots-clés : ("pilotage","navigation")
illustrer-Plan : plan_A1.v1
décomposer-Chapitre : chap_3.v0 chap_4.v0
décrire-Projet : P_ATRx.v0
date : 30/02/1993 creator : Durand
Doc_ATR42.v0
titre : "Document_ATR42"
mots-clés : ("pilotage","navigation")
illustrer-Plan : plan_A1.v0
décomposer-Chapitre : chap_3.v0 chap_4.v0
décrire-Projet : P_ATRx.v1
date : 02/02/1993 creator : Durand
Doc_ATR42.v2
titre : "Document_ATR42"
mots-clés : ("pilotage","navigation")
illustrer-Plan : plan_A1.v2
décomposer-Chapitre : chap_3.v1 chap_4.v0
décrire-Projet : P_ATRx.v0
date : 15/03/1993 creator : Dupont
Doc_ATR42.v3
titre : "Document_ATR42"
mots-clés : ("pilotage","navigation")
illustrer-Plan : plan_A1.v2
décomposer-Chapitre : chap_3.v1 chap_4.v1
décrire-Projet : P_ATRx.v1
date : 20/04/1993 creator : Martin
version gelée
version par défaut
date creator
critères internes donnés par
le concepteur ou le système


Plan
Plan_A3.v0
nom : "Plan_A3"
descriptif : "schéma 3D"
image :
date : 04/02/1990
creator : Martin
Plan_A33.v0
nom : "Plan_A33"
descriptif : "schéma 2D"
image :
date : 03/08/1993
creator : Martin
Plan_A1.v0
nom : "Plan_A1"
descriptif : "schéma 2D"
image :
date : 03/08/1993
creator : Martin
Plan_A1.v1
nom : "Plan_A1"
descriptif : "schéma 2D"
image :
date : 30/02/1993
creator : Martin
Plan_A1.v2
nom : "Plan_A33"
descriptif : "schéma 3D"
image :
date : 15/03/1993
creator : Martin
version gelée
version par défaut


Chapitre
chap_1.v0
titre : "ordinateur de bord"
contenu : ....
.. ordinateur ...
date : 04/02/1990
creator : Dupont
chap_1.v1
titre : "ordinateur de bord"
contenu : .... pilote ....
.. ordinateur ...
date : 01/09/1993
creator : Dupont
chap_2.v0
titre : "contrôles"
contenu : ....
.. contrôler ...
date : 04/02/1990
creator : Martin
chap_2.v1
titre : "check-list"
contenu : ....
.. contrôler ...
date : 01/09/1993
creator : Dupont
chap_3.v0
titre : "atterissage"
contenu :
date : 10/03/1992
creator : Durand
chap_3.v1
titre : "atterissage"
contenu : .... pilote ....
.. ..
date : 15/03/1993
creator : Durand
chap_4.v0
titre : "instrumentation"
contenu : ....
.. ordinateur ...
date : 02/02/1993
creator : Dupont
chap_4.v1
titre : "instrumentation"
contenu : .... tableau de
bord ....
date : 20/04/1993
creator : Martin
version gelée
version par défaut


Contrat
CR_1
numéro : 1
secteur : "recherche"
définir-Projet : P_A3x.v0 P_A3x.v1
CC_1
numéro : 2
secteur : "commercial"
définir-Projet : P_A3x.v2 P_ATRx.v0
CM_1
numéro : 3
secteur : "militaire"
définir-Projet : P_ATRx.v1





4.2.2. Interrogation de classes d'objets

La syntaxe de notre langage s'inspire de celle du langage OQL du système O2 (O2, 96). Les ensembles interrogés sont les classes d'objets. Le résultat est une nouvelle classe d'objets. Les prédicats concernent un objet. Ils peuvent utiliser les attributs, les méthodes, les liens de composition et les liens d'association. Pour des

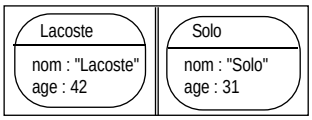
attributs et des liens multi-valués, les quantificateurs sont utilisés dans les prédicats ; les opérateurs de comparaison d'ensembles (égalité, inclusion, ...) peuvent également être utilisés.

Exemple III-9 : Considérons une requête avec des prédicats portant sur des attributs ; une condition sur un attribut mono-valué et une condition quantifiée sur un attribut multi-valué.

question : "Obtenir les nom et âge des personnes de plus de 30 ans, possédant le diplôme de DEA Informatique".

requête : **Select** p→nom, p→age **From** p **in** Personne
Where p→age > 30
And Exist q **in** p→diplômes : q = "DEA Informatique"

Pour la base exemple, la requête fournit le résultat suivant :



Les fonctions standards des langages de requêtes des systèmes de gestion de bases de données telles que **Max()**, **Count()**, ... sont également utilisables au niveau des prédicats et du résultat. De plus, les opérateurs ensemblistes tels que **Union**, **Intersect**, ... peuvent être appliqués notamment pour les résultats de plusieurs Select From Where.

4.2.3. Interrogation de classes de versions

L'expression d'une requête sur des classes de versions est basée sur celle décrite précédemment pour les classes d'objets. La portée d'une requête, c'est-à-dire l'ensemble interrogé, est par défaut un ensemble de forêts de versions lorsque l'interrogation porte sur une classe. Des fonctions **Unforest**, **Untree**, **Forest** et **Tree** permettent des transformations d'ensembles ; l'utilisation des fonctions **Unforest** et **Untree** permet notamment d'interroger un ensemble d'arbres de versions ou un ensemble de versions. Le résultat d'une requête est par défaut du même type que l'ensemble interrogé. Suivant le type de résultat obtenu, les fonctions **Untree**, **Tree**, **Unforest**, **Forest** permettent de restructurer le résultat.

Les prédicats s'appliquent à une forêt de versions, un arbre de versions ou une version ; plusieurs prédicats, éventuellement de niveaux différents (ex. niveau forêt et niveau version), peuvent évidemment être combinés avec les opérateurs logiques **et** et **ou**. Les conditions peuvent concerner les critères internes en utilisant les fonctions correspondantes (**Date**, **Creator**, **Number**, ...), les attributs, les méthodes, les liens de composition et d'association. Les fonctions **Max()**, **Count()**, ... sont toujours

utilisables ; les opérateurs ensemblistes peuvent également être utilisés notamment pour combiner les résultats de plusieurs requêtes.

4.2.3.1. Les niveaux d'abstraction

Le concept de forêt de versions d'objets pour décrire une entité du monde réel permet de décrire différentes solutions de conception dès la première représentation ; autrement dit, il permet plusieurs représentations d'une entité dès le départ.

Cependant, l'utilisation du concept de forêt est également importante lors de l'interrogation des classes de versions. En effet, lorsque l'on considère qu'une entité est représentée par un arbre de dérivation de versions, le résultat d'une sélection peut être d'une nature différente. Ainsi, le résultat d'une sélection sur un arbre de versions peut être un ensemble de versions appartenant à des branches différentes de l'arbre et qui n'ont pas d'ancêtre commun.

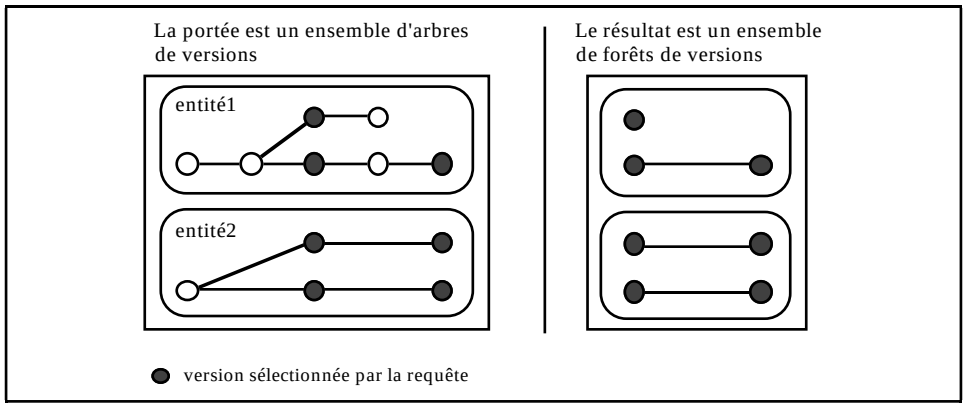


Figure III-2 : Problème liés à la représentation sous forme d'arbres de versions

Si le résultat doit être mis en forme, il est seulement possible de construire des forêts de versions. Le résultat d'une requête et sa portée sont alors de types différents. Les systèmes permettant de modéliser uniquement des arbres de versions comme les systèmes ORION, Aristote, OVM ne peuvent résoudre ce problème. Ces systèmes ne considèrent les versions qu'indépendamment les unes des autres c'est-à-dire comme de simples objets. La hiérarchie de dérivation des versions n'est pas considérée dans l'ensemble interrogé ; elle ne se retrouve donc pas non plus au niveau du résultat. Notre approche, qui permet de construire des forêts de versions, permet d'obtenir des résultats de requêtes de même type que les ensembles interrogés.

4.2.3.2. Les transformations d'ensembles

Lorsque l'on gère des versions d'objets, trois types d'éléments peuvent être manipulés : les forêts de versions, les arbres de versions et les versions. Par défaut, des ensembles de forêts de versions sont manipulés.

Des fonctions sont définies pour permettre la transformation d'ensembles suivant les trois niveaux d'organisation des versions. Les fonctions **Unforest**, **Untree**, **Forest** et **Tree** s'inspirent des opérateurs **Set** et **Flatten** du système O2 pour la transformation d'ensembles d'objets ainsi que des opérateurs **Nest** et **Unnest** du modèle NF^2(Scheck, 82).

La fonction **Unforest** permet de transformer un ensemble de forêts de versions en un ensemble d'arbres de versions ; les arbres ne sont alors plus regroupés suivant les entités du monde réel qu'ils décrivent. Inversement, la fonction **Forest** transforme un ensemble d'arbres de versions en un ensemble de forêts ; les arbres sont regroupés suivant les entités du monde réel qu'ils décrivent. **Forest** et **Unforest** sont des fonctions inverses, c'est-à-dire **Forest**(**Unforest**(ens_forêts))=ens_forêts et **Unforest**(**Forest**(ens_arbres))=ens_arbres.

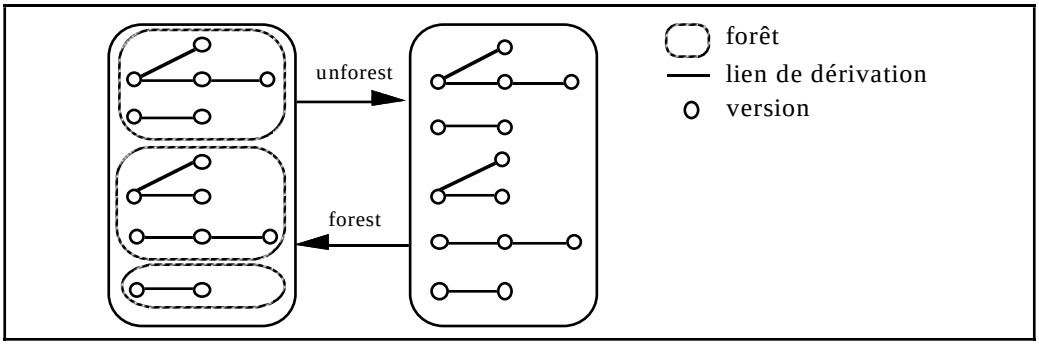


Figure III-3 : Opérateurs Forest et Unforest

De même, la fonction **Untree** permet de transformer un ensemble d'arbres de versions en un ensemble de versions ; les versions ne sont plus organisées suivant la relation de dérivation. Inversement, la fonction **Tree** permet de transformer un ensemble de versions en un ensemble d'arbres ; les versions sont organisées suivant les liens de dérivation. **Untree** et **Tree** sont des fonctions inverses ; ainsi, **Untree**(**Tree**(ens_versions))=ens_versions et **Tree**(**Untree**(ens_arbres)=ens_arbres.

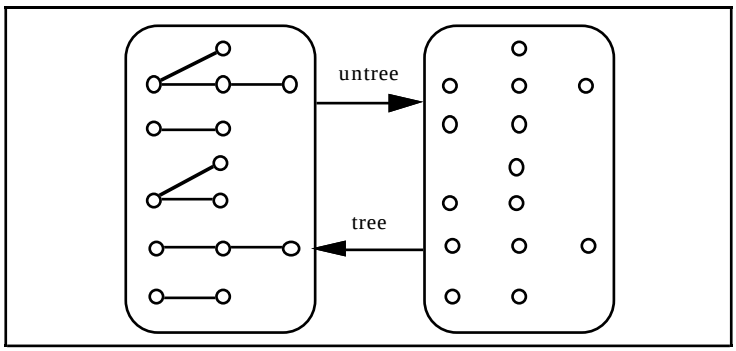


Figure III-4 : Opérateurs Tree et Untree

Les fonctions peuvent évidemment être combinées pour passer d'un ensemble de forêts de versions à un ensemble de versions et inversement. Elles peuvent être

utilisées dans chacune des clauses des requêtes (select, from et where) notamment pour transformer l'ensemble constituant la portée d'une requête ou pour restructurer le résultat d'une requête.

4.2.3.3. Les critères internes

Les critères internes sont des informations associées à chaque version (cf. Chapitre II § 3.2.) ; ce sont notamment des informations relatives à la création d'une version. Les critères internes sont les suivants :

- la date de création de la version,
- le nom du créateur de la version,
- le numéro de la version,
- l'état de la version (gelée ou en cours),
- si la version est une version par défaut ou non.

Une fonction est définie pour chaque critère interne ; la fonction appliquée à une version renvoie la valeur du critère interne correspondant. Les fonctions correspondant aux critères internes énoncés sont respectivement :

- **Date()** qui renvoie une date ; des opérateurs de comparaisons de dates et d'intervalles de dates (**during**, **before**, **overlap**, ...) sont définis en se basant sur ceux définis dans les langages des bases de données temporelles notamment le langage TSQL2 (Snodgrass, 94).
- **Creator()** qui renvoie une chaîne de caractères correspondant au nom du concepteur de la version ; les opérateurs de comparaisons de chaînes de caractères ainsi que les caractères de substitution (*, ?, ...) sont définis.
- **Number()** qui renvoie un numéro de version.
- **Frozen()** qui est une fonction booléenne qui renvoie vrai lorsque la version à laquelle elle s'applique est gelée et faux lorsque celle-ci est en cours.
- **Default()** qui est une fonction booléenne qui renvoie vrai lorsque la version à laquelle elle s'applique est une version par défaut.

Ces fonctions permettent notamment de définir des prédicats de sélection sur des critères internes ou peuvent également être utilisées dans le résultat d'une requête.

Exemple III-10 : considérons la définition d'un prédicat sur le critère interne correspondant à la date de création des versions.

Date(v**) during** [01/01/1993, 31/12/1993]

Ce prédicat permet de sélectionner uniquement les versions créées en 1993.

4.2.3.4. Interrogation de forêts de versions

Une classe simple de versions regroupe des forêts de dérivation de versions d'objets, chaque forêt décrivant une entité du monde réel. Par défaut, lorsque la portée d'une requête est une classe de versions, l'interrogation porte sur un ensemble de forêts de versions d'objets. Le résultat de la requête, par défaut de même nature que la portée de la requête, est également un ensemble de forêts de versions. La construction d'une requête sur les classes de versions est donc uniforme avec celle sur les classes d'objets.

Les prédicats portent sur :

- une hiérarchie décrivant une entité en utilisant la fonction **Entity()** ou les opérateurs de comparaison ensemblistes,
- les critères internes (en utilisant les fonctions pré-définies **Date()**, **Creator()**, ...), les attributs, les méthodes ou les liens de composition ou d'association.

• ***Interrogation de forêts décrivant des entités données***

Pour obtenir la forêt de versions d'objets décrivant une entité donnée (identifiée par son descripteur), la fonction **Entity()** est utilisée. Cette fonction, appliquée à une forêt de dérivation versions, renvoie le descripteur (chaîne de caractère, nombre, ...) de l'entité qu'elle décrit.

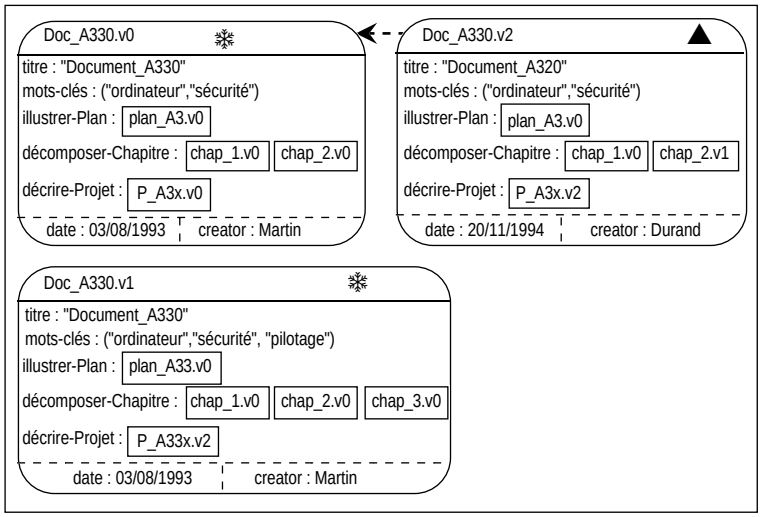
Exemple III-11 : Considérons la requête permettant d'obtenir la forêt de versions décrivant l'entité "Doc_A330" dans la classe *Documentation*.

question : "Obtenir la documentation Doc_A330"

principe : On recherche une forêt de dérivation complète ; l'ensemble interrogé est donc la classe Documentation (qui contient les forêts décrivant des documentations). Le prédicat porte sur le descripteur de forêt ou d'entité (unique pour chaque forêt décrivant une entité) ; la forêt (entité) recherchée est celle ayant pour descripteur Doc_A330.

requête :
```
Select h
From h in Documentation
Where Entity(h) = "Doc_A330"
```

Pour la classe *Documentation* choisie comme exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



- ***Interrogation de forêts avec quantificateur***

Deux types d'interrogations de forêts de versions peuvent être distinguées :

- soit on recherche les forêts dont toutes les versions vérifient une condition donnée. Ce cas permet notamment d'effectuer une recherche de forêts par rapport à une caractéristique invariante pour toutes les versions (ex. les projets dont le budget a toujours dépassé les 60 Gf). Dans ce cas, le quantificateur universel **Forall** est utilisé.
- soit on recherche les forêts dont au moins une des versions vérifie une condition donnée. Ce cas permet notamment d'obtenir des forêts décrivant des entités lorsque seule une information concernant une version est connue (ex. les projets dont le budget a déjà dépassé les 60 Gf). Dans ce cas, le quantificateur existentiel **Exist** est utilisé.

Exemple III-12 : considérons une requête sur la classe *Documentation* avec quantificateur existentiel ; la condition vérifiée par au moins une version concerne l'attribut *mots-clés*.

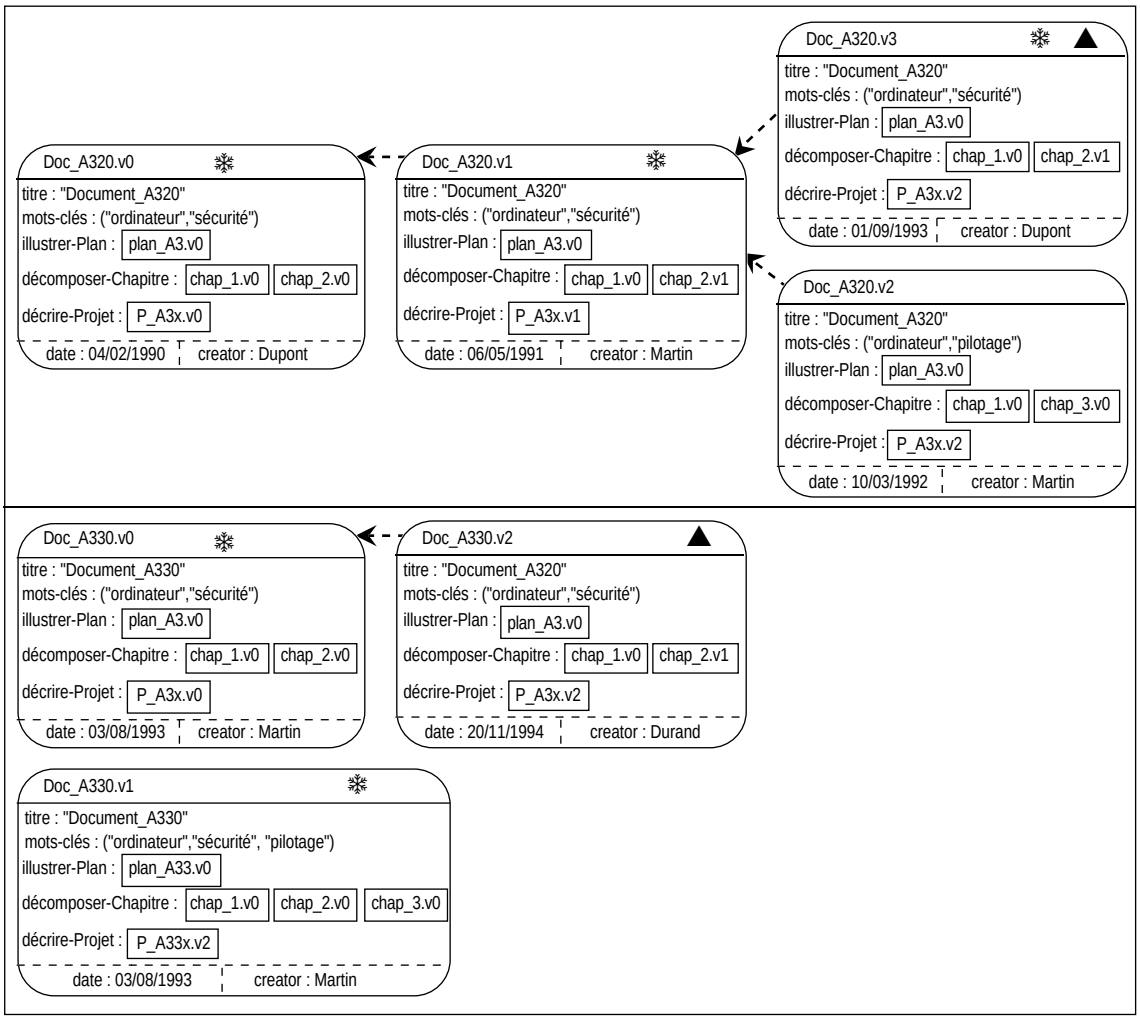
question : "Obtenir les documentations dont une des versions a traité de sécurité".

principe : On recherche des forêts de dérivation complètes ; l'ensemble interrogé est donc une classe (Documentation). Le prédicat porte sur l'ensemble des versions constituant chaque forêt ; on utilise le quantificateur existentiel pour spécifier qu'au moins une version de chaque forêt doit satisfaire la condition (inversement on utilise le quantificateur existentiel si toutes les versions de chaque forêt doivent satisfaire la condition).

requête : **Select** h
From h **in** Documentation
Where Exist v **in Untree(**h**) :** "sécurité" in v→mots-clés

h est une forêt de dérivation de versions c'est-à-dire un ensemble d'arbres de versions. Le quantificateur appliqué à h ferait intervenir les arbres constituant h. Or le quantificateur existentiel s'applique aux versions, il faut donc transformer l'ensemble d'arbres h en ensemble de versions, en utilisant la fonction **Untree()**. La requête recherche ainsi chaque forêt pour laquelle on trouve une version ayant 'sécurité' pour mot-clé.

Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



4.2.3.5. Interrogation aux niveaux arbres et versions

Une classe est un ensemble de forêts de versions. Lorsque la portée d'une requête est une classe, l'interrogation porte sur des forêts de versions. Néanmoins, il est possible d'effectuer des requêtes sur des ensembles d'arbres de versions ou des ensembles de versions. Pour cela, les fonctions **Untree** et **Unforest** doivent être utilisées.

Ainsi, puisqu'une classe est un ensemble de forêts de versions, en appliquant la fonction **Unforest** à la classe, la portée d'une requête devient un ensemble d'arbres de versions. De même, le résultat de la requête, par défaut de même nature que la portée, devient également un ensemble d'arbres de versions. Ce type d'interrogation permet d'obtenir l'arbre de dérivation correspondant à une alternative de conception donnée (ex. l'arbre de dérivation de versions décrivant la solution de M. Dupont).

Exemple III-13 : Considérons une requête utilisant la fonction **Unforest** au niveau de la portée ; la portée et le résultat de la requête sont donc des ensembles d'arbres de versions.

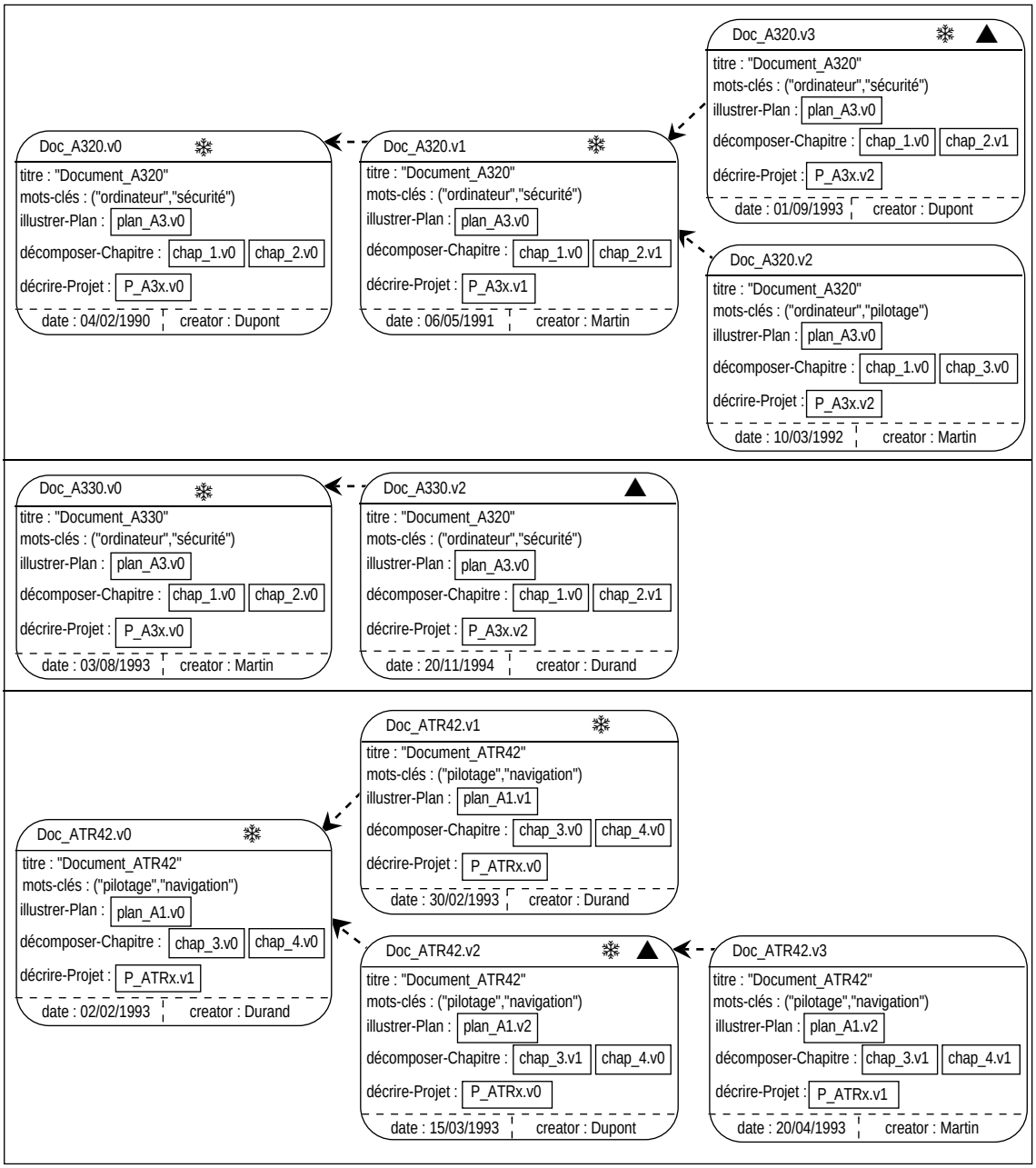
question : "Obtenir, pour toute documentation, l'évolution par défaut" ; il s'agit de conserver les arbres de dérivation de versions auxquels appartient la version par défaut de chaque documentation.

principe : On recherche des arbres de dérivation ; l'ensemble interrogé est donc un ensemble d'arbres c'est-à-dire une classe (ensemble d'ensembles d'arbres) transformée en ensembles d'arbres à l'aide de la fonction **Unforest**. Le prédicat porte sur l'ensemble des versions constituant chaque arbre ; on utilise le quantificateur existentiel pour spécifier qu'au moins une version de chaque arbre doit satisfaire la condition (ici être version par défaut).

requête : **Select** a
From a **in Unforest(**Documentation**)**
Where Exist v **in** a **: Default(**v**)**

La requête recherche donc chaque arbre de dérivation pour lequel on trouve une version qui soit version par défaut.

Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



De même, en appliquant successivement les fonctions **Unforest** et **Untree** à une classe, la portée d'une requête devient un ensemble de versions. Le résultat est alors également un ensemble de versions. Ce type d'interrogation permet d'obtenir des états précis des entités décrites c'est-à-dire seulement les versions vérifiant une condition donnée (ex. pour chaque documentation, uniquement les versions créées par M. Durand).

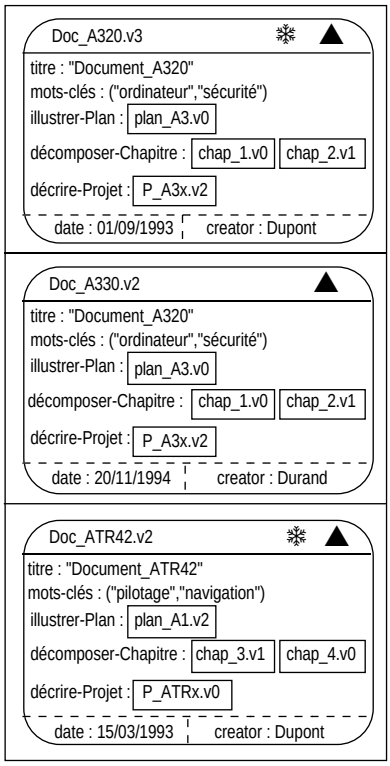
Exemple III-14 : Considérons une requête utilisant les fonctions **Unforest** et **Untree** au niveau de la portée ; la portée et le résultat de la requête sont donc des ensembles de versions.

question : "Obtenir la version significative de chaque documentation" ; seule la version par défaut de chaque entité documentation représentée doit être conservée.

principe : On recherche des versions ; l'ensemble interrogé est donc un ensemble de versions c'est-à-dire une classe (ensemble d'ensembles d'ensembles de versions) transformée en ensemble de versions à l'aide des fonctions **Unforest** puis **Untree**. Le prédicat porte sur chaque version (ici être version par défaut).

requête : **Select** v
From v **in Untree(Unforest(**Documentation**))**
Where Default(v**)**

La requête recherche donc chaque version qui soit version par défaut. Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



4.2.3.6. Restructuration du résultat

Par défaut, le résultat et la portée d'une requête sont des ensembles de même type. Il est possible de changer le type d'ensemble résultat en appliquant les fonctions **Forest**, **Tree**, **Unforest** et **Untree** selon le type du résultat.

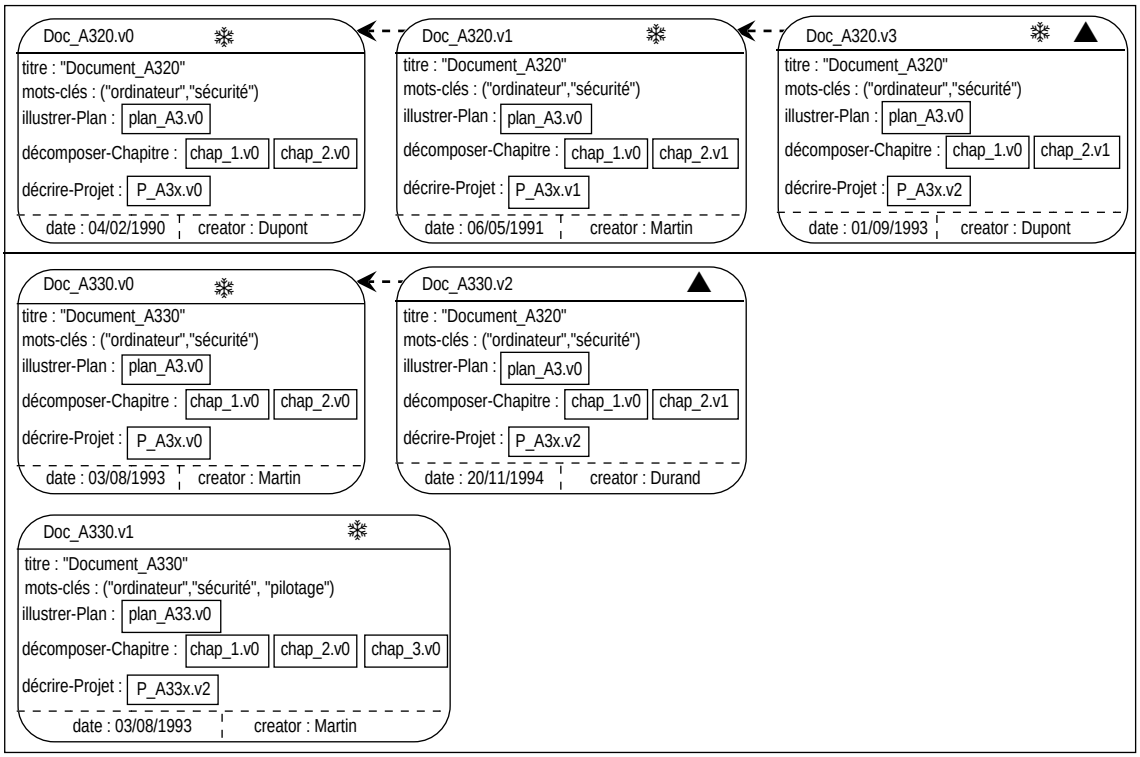
Exemple III-15 : considérons une requête dont la portée est un ensemble de versions et dont le résultat est restructuré en ensemble de forêts de versions.

question : "Obtenir, pour toute documentation, l'évolution des versions comportant 'sécurité' pour mot-clé"

principe : On recherche tout d'abord des versions ; la requête de base est donc une interrogation de versions (Cf. Exemple III-14). Il faut ensuite reconstruire le résultat sous forme de forêts de dérivation (regroupement sous forme d'arbres à l'aide de la fonction **Tree**, puis regroupement par entité (forêt) à l'aide de la fonction **Forest**).

requête : **Forest(Tree(Select** v
From v **in Untree(Unforest(**Documentation**))**
Where "sécurité" in v→mots-clés **))**

Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



4.2.3.7. Combinaisons de prédicats à différents niveaux

Pour une requête sur une classe simple de versions, différents prédicats spécifiés à différents niveaux (forêt, arbre ou version) peuvent être combinés en utilisant les opérateurs logiques **et** et **ou**. Ainsi, il est notamment possible d'obtenir des versions particulières appartenant à des forêts particulières.

Exemple III-16 : considérons une requête permettant d'obtenir des versions vérifiant une condition, ces versions appartenant à des forêts vérifiant elles-mêmes une certaine condition.

question : "Obtenir les versions, créées en 1993, des documentations dont toutes les versions ont traité de sécurité".

principe : On recherche tout d'abord des forêts complètes répondant à un prédicat ; la requête de base est donc une interrogation de forêts (Cf. Exemple III-11). Pour les forêts obtenues, on recherche ensuite les versions satisfaisant une condition ; une deuxième requête qui est une interrogation de versions (Cf. Exemple III-14) utilise le résultat de la première requête.

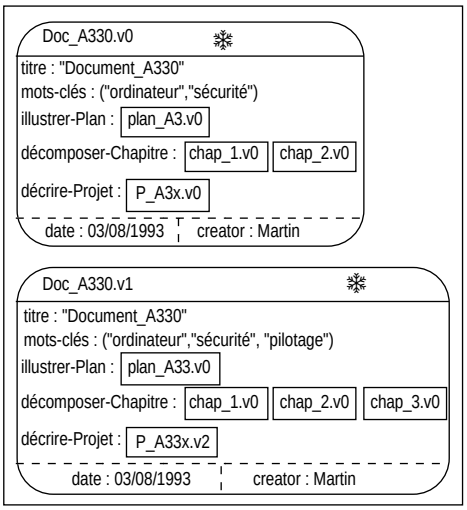
requête : **Select** v
From v **in** Untree(Unforest(**Select** h **from** h in Documentation
Where Forall x **in Untree(**h**) :**
"sécurité" in x->mots-clés))
Where Date(v**) during** [1/1/1993, 31/12/1993]

La première requête recherche chaque forêt pour laquelle toutes les versions ont 'sécurité' pour mot-clé ; Dans l'ensemble des versions des forêts obtenu, la deuxième requête recherche les versions créées durant 1993.

Une autre solution est une auto-jointure.

requête : **Select** v **From** h **in** Documentation, v **in Untree(**h**)**
Where Forall x **in Untree(**h**) :** "sécurité" in x->mots-clés
And Date(v**) during** [1/1/1993, 31/12/1993]

Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



4.2.3.8. Liens entre classes simples de versions

A l'image des classes d'objets, l'interrogation sur les classes de versions peut être effectuée suivant les liens de composition et d'association qui sont définis entre les classes. L'interrogation tient compte des contraintes inhérentes à la définition des liens. L'interrogation suivant un lien de composition ne peut s'effectuer que pour les instances d'une classe composée.

Pour un lien de composition mono-valué (cardinalité de type α-1 avec $\alpha \in \{0,1\}$), une version composée est liée à une seule version composante (éventuellement aucune lorsque la cardinalité minimum est 0 (cf. Chapitre II 3.7.2.2.1.)). Les opérateurs de comparaison de versions sont utilisés : égalité (égalité des valeurs), identité (égalité des identifiants), différence. La comparaison avec la valeur nulle est possible. Il est également possible de poser un prédicat sur les attributs de la classe composante.

Exemple III-17 : considérons une requête portant sur le lien de composition mono-valué entre la classe *Documentation* et la classe *Plan*.

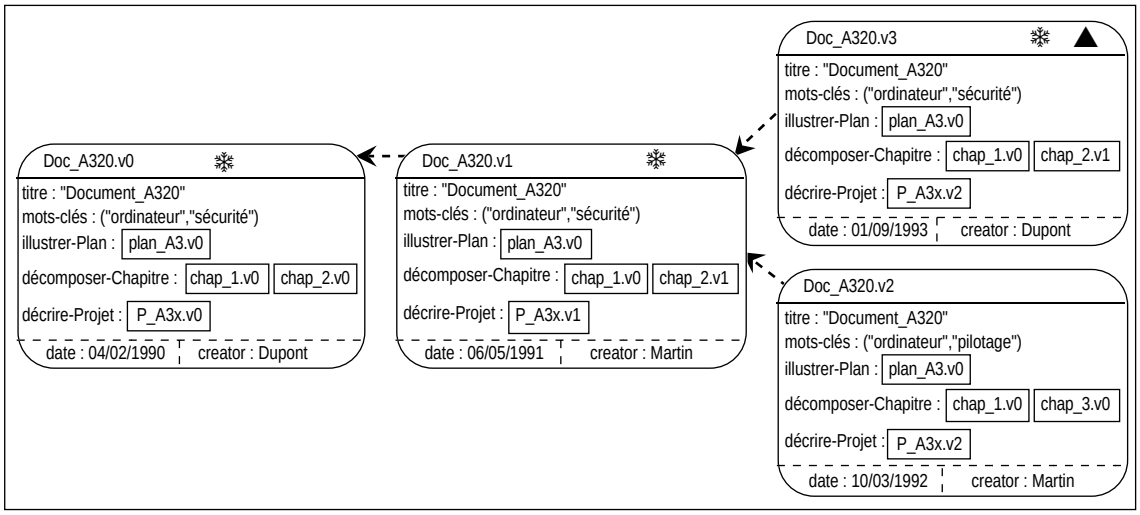
question : "Obtenir les documentations dont toutes les versions ont comporté le plan plan_A3".

principe : Il s'agit d'obtenir les forêts de dérivation décrivant des documentations pour lesquelles toutes les versions sont composées (lien *illustrer*) de versions appartenant à la forêt décrivant le plan plan_A3. La base de la requête est une interrogation de forêts (Cf. Exemple III-11). La différence est au niveau du prédicat, sur chaque version, qui porte sur un lien. Chaque version est ici liée à une seule instance ; on peut donc utiliser un prédicat de comparaison simple.

requête :
```
Select h
From h in Documentation
Where Forall v in Untree(h) : v->illustrer-Plan->nom = "plan_A3"
```

La requête retrouve donc chaque forêt *documentation* pour laquelle toute version est liée à une version *plan* porte le nom 'plan_A3'. Lorsqu'il n'y a pas ambiguïté, il n'est pas nécessaire d'indiquer le nom de la classe associée au lien.

Pour la classe *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



Pour un lien de composition multi-valué, une version composée est liée à un ensemble de versions composantes ; l'ensemble est éventuellement vide lorsque la cardinalité minimum est nulle (cf. Chapitre II 3.7.2.2.1.). L'opérateur d'appartenance d'un élément à un ensemble est utilisable ainsi que les opérateurs de comparaison d'ensembles : égalité, inclusion, différence. La comparaison avec l'ensemble vide est possible.

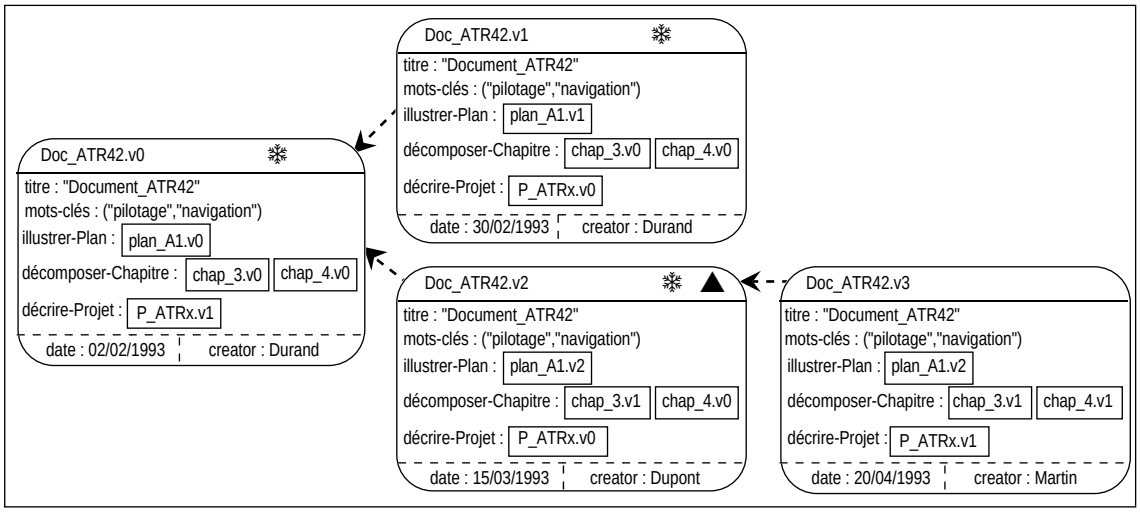
Exemple III-18 : Interrogation sur la classe *Documentation* portant sur le lien de composition multi-valué *décomposer* avec la classe composante *Chapitre*.

question : "Obtenir les documentations dont au moins une version a comporté un chapitre intitulé instrumentation".

principe : Il s'agit d'une interrogation sur des forêts (Cf. Exemple III-11). La différence est ici au niveau du prédicat, sur chaque version, chaque version est ici liée à un ensemble d'instances ; on utilise un prédicat avec un quantificateur (existentiel ou universel suivant les cas).

requête :
```
Select d
From d in Documentation
Where Exist v in Untree(d) : Exist c in v->décomposer-Chapitre :
                                        c→titre = "instrumentation"
```

Pour les classes *Documentation* et *Chapitre* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



Pour les liens d'association, une version est toujours liée à un ensemble de versions ; selon la cardinalité, ces versions appartiennent à une seule hiérarchie ou à plusieurs (cf. Chapitre II § 3.7.2.3.1.). Ainsi, l'interrogation est analogue à celle définie pour les liens de composition multi-valués (Exemple III-18).

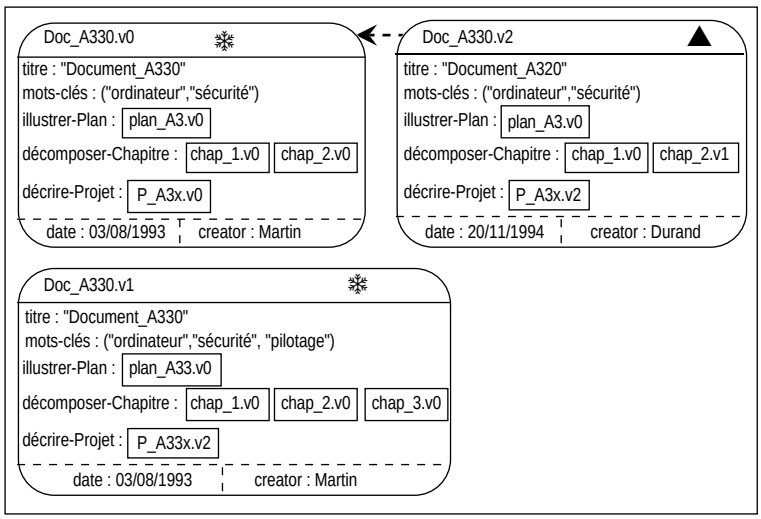
Exemple III-19 : considérons une requête sur le lien d'association *décrire* entre les classes *Projet* et *Documentation*.

question : "Obtenir les documentations liées au projet P_A33x".

principe : Seules sont conservées les forêts de versions de *Documentation* dont au moins une version est en rapport avec le Projet P_A33x. Le principe est le même que pour l'exemple précédent (Cf. Exemple III-18).

requête : **Select** h **From** h **in** Documentation
Where Exist v **in Untree(**d**):**
Exist p **in** v→décrire-Projet: **Entity(**p**)** = "P_A33x"

Pour les classes *Projet* et *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :



4.2.4. Interrogations entre classes d'objets et de versions

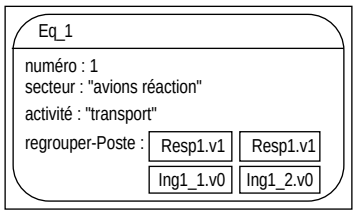
L'interrogation suivant un lien entre classes d'objets et classes de versions prend en compte les contraintes inhérentes aux liens dans ce cas de figure (cf. Chapitre II § 3.7.). Selon les cas, une version (resp. un objet) est liée à un seul objet (resp. une version) ou à un ensemble d'objets (resp. un ensemble de versions). L'interrogation s'apparente alors à des cas de liens entre instances de même type (objets ou versions) décrit précédemment (cf. § 4.2.3.).

Exemple III-20 : considérons une requête sur la classe d'objets composée *Equipe*, suivant le lien de composition *regrouper* multi-valué avec la classe de versions *Poste*. Un objet de *Equipe* est lié à un ensemble de versions de *Poste* ; la multi-valuation autorise que ces versions appartiennent plusieurs hiérarchies (cf. Chapitre II § 3.7.3.). Les prédicats sont donc basés sur ceux définis pour un lien de composition multi-valué entre deux classes simples d'objets.

question : "Obtenir les équipes qui ont toujours eu des postes de niveau supérieur à 2".

requête : **Select** o
From o **in** Equipe
Where Forall v in o->regrouper-Poste **:** v->niveau > 2

Pour les classes Projet et Documentation de la base exemple (cf. § 4.2.1.), le résultat fourni par la requête est le suivant :

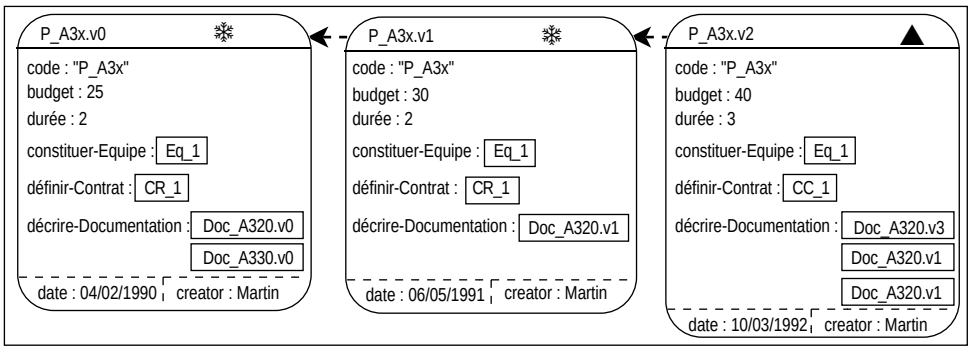


Exemple III-21 : considérons une requête sur la classe de versions *Projet* suivant le lien d'association *définir* avec la classe d'objets *Contrat*. Une version de *Projet* est liée à un seul objet *Contrat* puisque le lien est mono-valué (elle serait liée à un ensemble d'objets si le lien était multi-valué). La requête est analogue à une requête sur une classe de versions composées suivant un lien de composition mono-valué vers une classe de versions composantes (cf. Exemple III-17).

question : "Obtenir les projets qui ont été liés à un contrat de recherche".

requête : **Select** h
From h **in** Projet
Where Exist v **in Untree(**h**) :** v->définir-Contrat->secteur="recherche"

Pour les classes *Projet* et *Documentation* de la base exemple (cf. § 4.2.1.), le résultat fournit par la requête est le suivant :



4.3. Interrogation de versions de classes

Une hiérarchie de versions de classes est un ensemble de versions de classe, liées par un lien de dérivation. Les versions de classe d'une même hiérarchie décrivent les mêmes entités du monde réel. Il est possible d'interroger une ou plusieurs des versions de classe d'une hiérarchie. L'interrogation d'une version de classe est analogue à l'interrogation d'une classe simple ; l'interrogation d'un ensemble de versions d'une classe utilise le concept de super-classe commune aux versions de classes considérées.

4.3.1. Base exemple avec versions de classes

Pour illustrer les différentes requêtes qu'il est possible de définir, nous utiliserons l'exemple de base suivant :

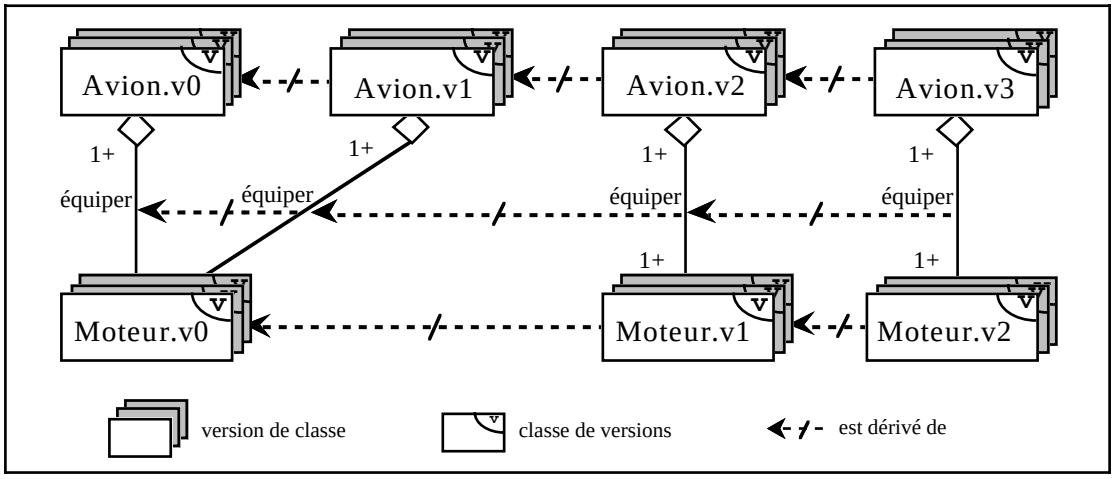


Figure III-5 : Base exemple avec versions de classe

La définition du schéma dans notre langage est la suivante :

create class version Avion
contains versions
attributes (désignation : string, catégorie : string, secteur : char**)**

create class version Moteur
contains versions
attributes (nom : string, type : char, puissance : integer**)**

Composition équiper : Avion.v0 (1,1) **of** Moteur.v0 (1,N)

create class derived from Avion.v0
attributes add capacité : integer

begin transaction

create class derived from Avion.v1
attributes delete secteur

create class derived from Moteur.v0
attributes add marque : string

Composition équiper : Avion.v2 (1,N) **of** Moteur.v0

end transaction

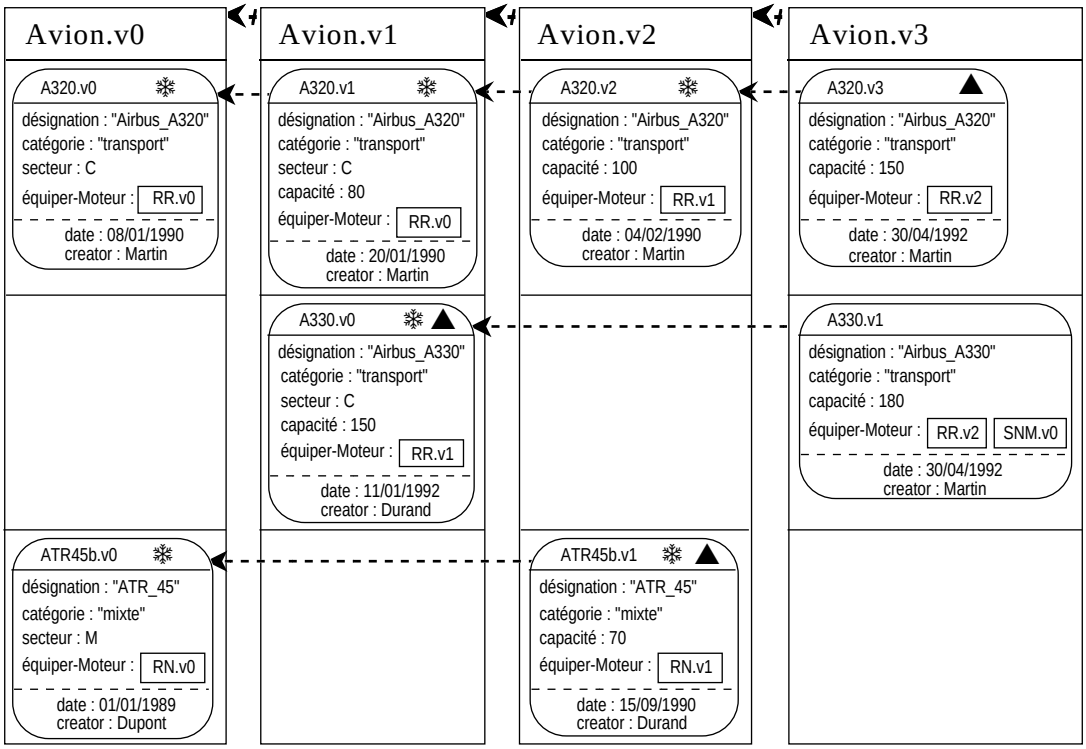
begin transaction

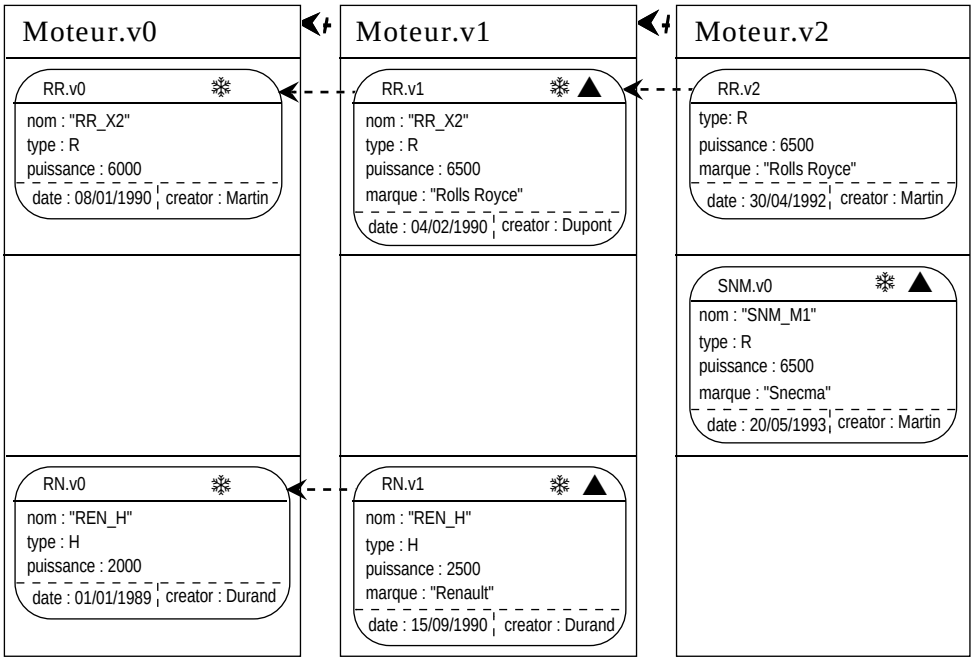
create class derived from Avion.v2

create class derived from Moteur.v1
attributes delete nom

end transaction

Les instances pour chacune des classes, utilisées lors des exemples de requêtes sont les suivantes :





4.3.2. Interrogation d'une version de classe

Une version de classe est une classe simple ; elle est liée à d'autres versions de la classe par un lien de dérivation. L'interrogation est donc analogue à celle des classes simples (cf. § 4.3.). Une requête s'appliquant à une version de classe n'examine que les instances de cette version de classe ; les instances des versions de classes précédentes et suivantes ne sont pas considérées par la requête. Le résultat de la requête ne comporte donc que des informations relatives aux instances de la version de classe considérée.

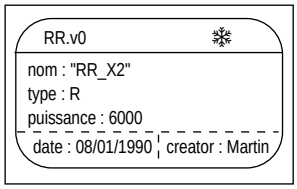
Exemple III-22 : Considérons une requête sur la version v0 de la classe *Moteur*, avec un prédicat sur l'attribut *puissance*.

question : "Obtenir, dans le schéma initial, les moteurs dont la puissance a déjà atteint 5000"

principe : Le principe est le même que pour l'interrogation de classes simples (Cf. Exemple III-11) ; il suffit de préciser le numéro de version de classe.

requête : **Select** h
From h **in** Moteur.v0
Where Exist v **in Untree(**h**) :** v->puissance≥5000

Le résultat de la requête pour la base exemple est le suivant :



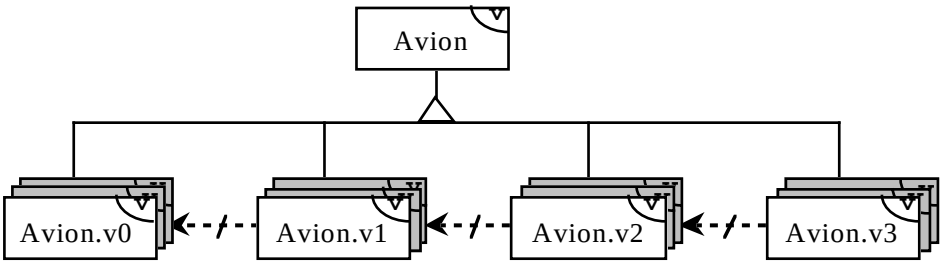
4.3.3. Super-classe commune à plusieurs versions de classe

Les différentes versions de classe d'une même hiérarchie possèdent des propriétés communes (attributs, méthodes, liens de composition et d'association) ; au minimum, la partie commune à plusieurs versions d'une classe se limite aux critères internes. Une classe commune peut être déduite des versions d'une classe. Cette classe est la super-classe commune aux versions de classe considérées ; elle possède les attributs, les méthodes, les liens de composition et d'association communes aux versions de classe ainsi que les critères internes.

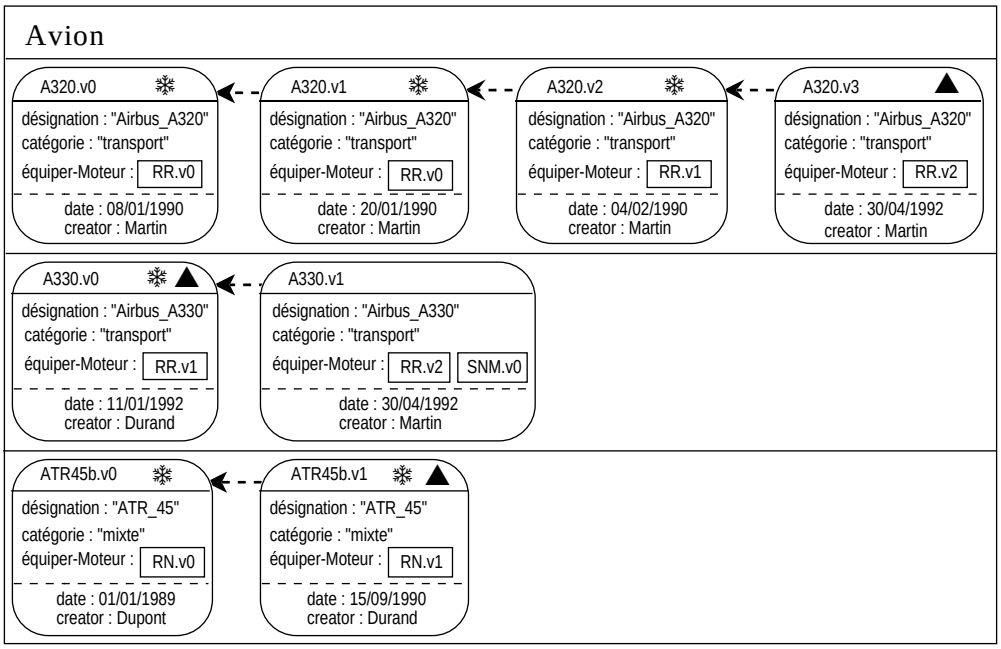
La super-classe regroupe les instances des versions de classe dont elle est super-classe ; le schéma de ces instances est limité à celui de la super-classe.

Exemple III-23 : La super-classe commune aux versions de la classe *Avion* possède le schéma suivant :

contains versions
attributes (désignation : string, catégorie : string)
avec le lien de composition : Composition équiper : Avion (1,N) of Moteur (1,N)



L'extension de la classe Avion pour la base exemple donnée est la suivante :



La notion de super-classe commune à plusieurs versions d'une même classe peut notamment être utilisée lors de l'interrogation de l'ensemble ou d'un sous-ensemble des versions de la classe.

4.3.4. Interrogation de plusieurs versions de classe

L'interrogation de plusieurs versions d'une classe est divisée en :

- l'interrogation de l'ensemble des versions de la classe,
- l'interrogation d'un sous-ensemble des versions d'une classe.

Les deux principes d'interrogation se différencient lors de la spécification de la portée d'une requête. Dans les deux cas, la notion de super-classe commune aux versions de classe interrogées est utilisée. Lorsqu'un ensemble de versions d'une classe est spécifié comme portée d'une requête, l'interrogation porte en fait sur la super-classe commune aux versions de classe considérées.

Les prédicats de requête concernent uniquement les propriétés de la super-classe (attributs, méthodes, ...) c'est-à-dire celles communes aux versions interrogées de la classe. L'interrogation de l'ensemble des versions de classe d'une même hiérarchie considère toutes les versions d'objets pour chaque entité représentée dans la classe tandis que l'interrogation d'un sous-ensemble des versions de la classe, ne considère que les versions d'objets instances des versions de classe interrogées. L'interrogation de l'ensemble des versions d'une classe est indiquée en spécifiant dans la portée uniquement le nom de la classe sans préciser de numéro de version.

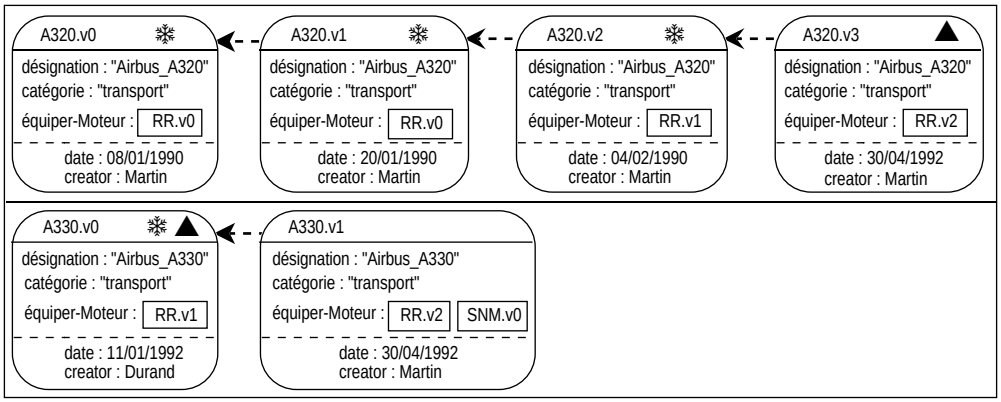
Exemple III-24 : Requête sur l'ensemble des versions de classe *Avion*.

question : "Obtenir les avions qui ont toujours été dans la catégorie transport".

principe : Le principe est le même que pour l'interrogation de classes simples (Cf. Exemple III-11). L'absence de numéro de version de classe indique que l'on interroge l'ensemble des versions de classe via la super-classe commune (Cf. 4.3.3.); seuls les attributs communs peuvent être utilisés dans le prédicat.

requête :
```
Select h
From h in Avion
Where Forall v in Untree(h) : v->catégorie="transport"
```

Le résultat de cette requête pour la base exemple est alors le suivant :



Dans le cas d'une interrogation d'un sous-ensemble de versions d'une classe, les versions de classe considérées sont spécifiées sous forme d'ensemble de numéros de versions ($[v_i, ..., v_j]$) ou d'intervalle de numéro de versions ($[v_i .. v_j]$).

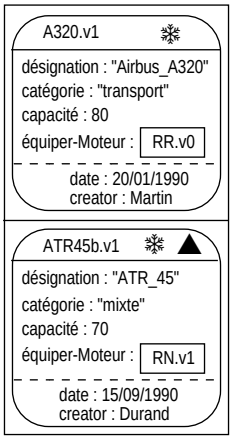
Exemple III-25 : Requête sur les versions v1 à v2 de la classe *Avion*.

question : "Obtenir, pour les schémas 1 à 2, toutes les versions d'avions ayant une capacité inférieure à 100".

principe : Le principe est le même que pour l'interrogation de l'ensemble des versions de classe (Cf. Exemple III-24). On précise simplement les numéros de version de classe interrogées.

requête : **Select** v
From v **in Untree(Unforest(**Avion.[v1 .. v2]**))**
Where v->capacité<100

Le résultat de cette requête pour la base exemple est alors le suivant :



4.3.5. Liens et versions de classes

L'interrogation suivant des liens de composition et d'association regroupe :

- l'interrogation d'une version d'une classe sur un lien avec une version de la classe liée ou sur les liens avec un ensemble de versions d'une classe,
- l'interrogation d'un ensemble de versions d'une classe sur un lien avec une version de la classe liée ou sur les liens avec un ensemble de versions de la classe liée.

Les principes établis pour spécifier un prédicat portant sur un lien sont les mêmes que pour les classes simples c'est-à-dire v->lien-classe ; les versions de classe liées qui sont considérées, sont précisées comme pour la portée c'est-à-dire .v_i pour une seule version, .[v_i, .., v_j] ou .[v_i .. v_j] pour un ensemble de versions de classe, pas de numéro pour l'ensemble des versions (cf. § 4.3.4.).

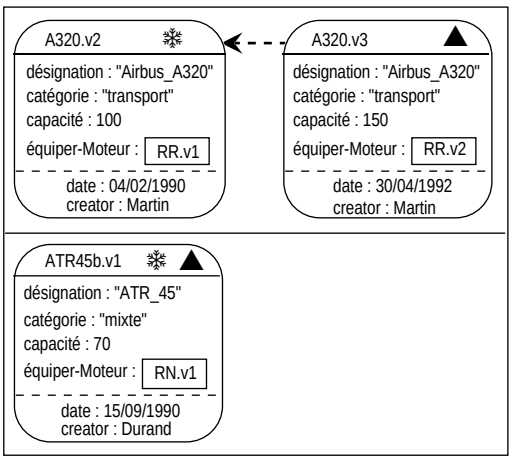
Exemple III-26 : Considérons une requête entre les versions v2 et v3 de la classe *Avion* et les versions v0 et v1 de *Moteur*.

question : "Obtenir, pour les schémas 2 à 3, les avions dont toutes les versions ont été équipées d'un seul type de moteur".

principe : Le principe est celui de l'interrogation sur des liens entre de classes simples (Cf. Exemples III-17 et III-18). On précise simplement dans la portée et les prédicats, les numéros des versions de classes concernées.

requête :
```
Select ha
From ha in Avion.[v2 .. v3]
Where Forall v in Untree(ha) : Count( Select distinct Entity(p)
                                      from p in v->équiper-Moteur.[v1, v2] ) = 1
```

Le résultat de cette requête pour la base exemple choisie est le suivant :



L'apparition d'un conflit de cardinalité pour un même lien entre différentes versions de classes (lien mono-valué entre certaines versions et multi-valué entre

d'autres versions) est résolu en considérant uniquement le cas général de multi-valuation ; la mono-valuation n'est qu'un cas particulier inclus dans le cas général.

4.4. Interrogation entre classes simples et versions de classes

Considérons les classes définies dans les bases exemples choisies précédemment et supposons qu'un lien d'association soit défini entre les versions de classe *Avion* et la classe simple *Projet*.

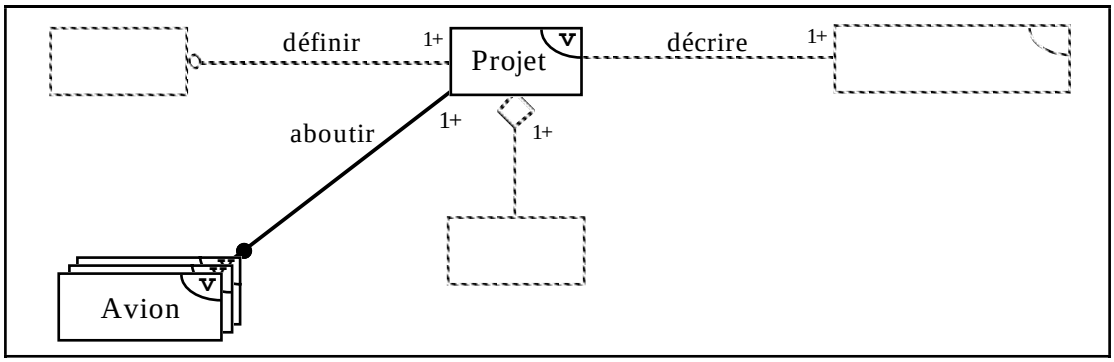


Figure III-6 : Liens entre bases exemples

L'interrogation d'une classe simple suivant un lien avec des versions d'une classe est similaire à l'interrogation d'une seule version d'une classe suivant un lien avec plusieurs versions d'une classe liée (cf. paragraphe précédent).

Exemple III-27 : Considérons une requête sur la classe simple *Projet* suivant le lien d'association *aboutir* avec l'ensemble des versions de classe *Avion*.

question : "Obtenir les Projets dont toutes les versions sont associées à un avion" ; il s'agit de conserver les forêts de Projet dont toutes les versions sont liées à des versions d'Avions.

requête : **Select** hp
From hp **in** Projet
Where Forall v **in Untree(**hp**):** v->aboutir-Avion ≠ set()

L'interrogation de l'ensemble des versions d'une classe suivant un lien avec une classe simple est analogue de l'interrogation l'ensemble des versions d'une classe suivant un lien avec une seule version d'une classe liée (cf. paragraphe précédent).

Exemple III-28 : Considérons une requête sur les versions de classe *Avion* suivant le lien d'association *aboutir* avec la classe *Projet*.

question : "Obtenir les avions qui ont déjà été liés au projet P_A3x".

requête : **Select** ha
From ha **in** Avion
Where Exist v **in Untree(**ha**) :**
Exist p **in** v->aboutir-Projet **: Entity(** p**)** = "P_A3x"

5. Bilan

Actuellement, les systèmes de gestion de versions ne proposent pas réelles possibilités pour la manipulation des versions. Ils ne fournissent en général qu'un langage d'interrogation d'objets. Les versions sont interrogées comme de simples objets sans considérer leur sémantique spécifique, c'est-à-dire sans tenir compte de leur organisation en hiérarchies de dérivation.

Le langage de description et de manipulation, que nous proposons, regroupe la définition et la manipulation des classes et des liens suivant les concepts de notre modèle ; il regroupe également les opérations applicables aux instances et l'interrogation des instances. Notre langage tient compte des spécificités liées à la gestion de versions d'objets et de classes. Ce langage uniformise les opérations applicables aux classes et aux instances ; il respecte ainsi l'utilisation unifiée du concept de version au niveau des classes et des instances.

Nous proposons une interrogation structurée de type Select From Where. L'interrogation est uniforme pour les objets et les versions d'objets. Elle intègre les spécificités liées aux notions de versions d'objets et de versions de classes. Elle permet de prendre en compte les niveaux d'abstraction liés aux versions c'est-à-dire les concepts de forêt de dérivation de versions, d'arbre de versions et de versions d'objet. Notre langage permet notamment de définir des prédicats de sélection suivant ces niveaux d'abstraction. L'interrogation peut également s'effectuer suivant les critères "internes" propres aux versions d'objets, notamment pour exprimer des requêtes suivant le temps, comme pour les langages pour bases de données temporelles. De plus, il est possible d'effectuer une interrogation sur l'ensemble ou une partie des versions d'une classe ; on utilise alors le concept de super-classe commune aux versions de classes interrogées.

Notre langage exploite ainsi les spécificités des versions, particulièrement en matière d'interrogation. Il peut être étendu à des fonctionnalités supplémentaires ; par exemple, il pourrait exploiter la différence entre les versions (ex. obtenir les versions les versions qui diffèrent suivant moins de deux attributs d'une version donnée c'est-à-dire obtenir les versions proches d'une version donnée).

CHAPITRE IV.

Réalisation

Chapitre IV. : Réalisation

Table des matières

1. Introduction

Le modèle de données et le langage de manipulation définis dans les chapitres précédents ont été expérimentés au travers d'un prototype. Nous avons participé au développement d'une interface graphique pour la manipulation de bases de données orientées objet intégrant des versions. Cette interface repose sur notre modèle et son langage.

Les systèmes existants tels que ORION (Kim, 89b) et Presage (Talens, 93) sont des SGBD intégrant des fonctionnalités de gestion de versions. Au lieu de construire un SGBD spécifique, nous proposons de définir un outil, qui appliqué à un SGBD, lui permet de gérer des bases de données intégrant des versions d'objets et de classes. Cet outil est l'ensemble des fonctionnalités nécessaires à l'intégration de la gestion de versions dans un SGBD. La solution proposée n'est pas liée au SGBDOO choisi.

Le modèle de données, la gestion des contraintes d'intégrité inhérentes aux liens et le langage, sont mis en oeuvre comme des modules indépendants. Ceci permet une plus grande extensibilité pour les développements futurs. Dans ce chapitre, nous proposons tout d'abord des solutions pour la mise en oeuvre de ces modules.

L'interface VOHQL (Version and Object Hypertext Query Language) est basée sur une représentation graphique d'une base définie suivant notre modèle. Elle est destinée à des utilisateurs occasionnels. Les requêtes sont construites directement en utilisant la représentation graphique de la base. Elles correspondent à des requêtes exprimées dans le langage structuré VOSQL. Ce chapitre présente également les principes de manipulation du prototype VOHQL et leur correspondance en langage textuel.

2. Principe

Le principe, que nous préconisons pour mettre en oeuvre la gestion de versions d'objets et de classes dans une base de données, est la définition d'un ensemble minimal de fonctionnalités ; ce dernier permettra d'étendre un SGBDOO à la gestion de versions. L'objectif est d'utiliser les éléments communs à tous les SGBD, c'est-à-dire à la fois les principes objet (les classes, l'héritage, ...), et les fonctionnalités générales des SGBD notamment le langage de requête.

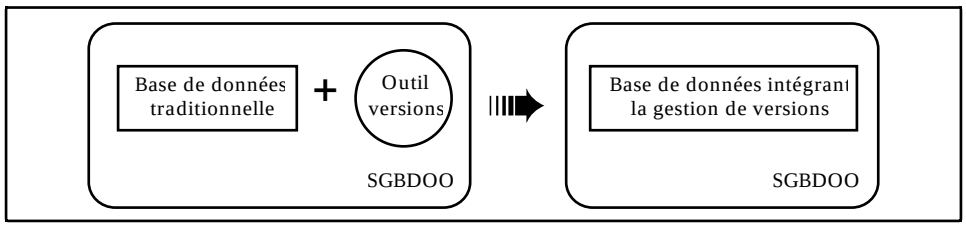


Figure IV-1 : Principe d'implantation

2.1. Spécification d'un méta-modèle

La définition d'un outil générique passe par la spécification d'un modèle décrivant les concepts nécessaires. Nous avons défini, par une description conceptuelle, un méta-modèle pour l'implantation de notre modèle dans un SGBDOO hôte générique.

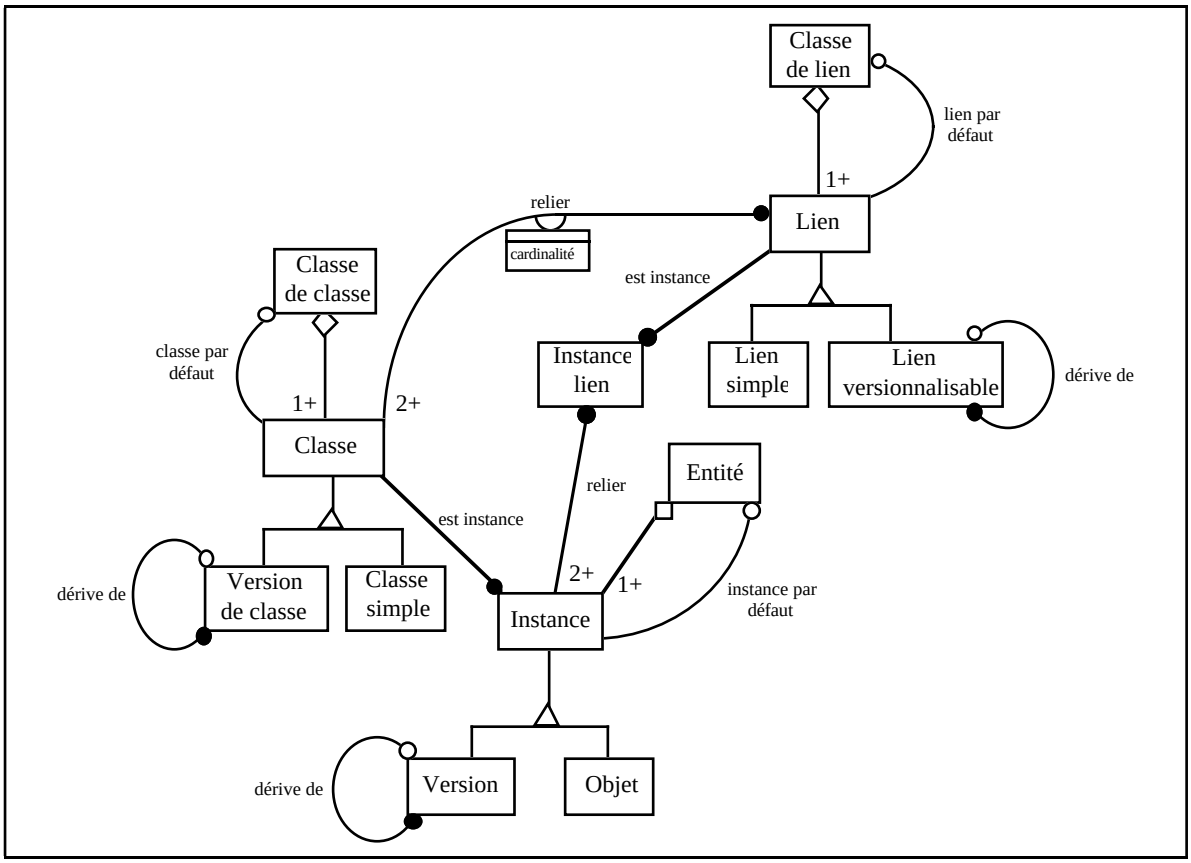


Figure IV-2 : Le méta-modèle

D'une part, la classe *Classe de classe* décrit les différentes classes d'entités représentées dans la base de données. Chaque classe d'entités est en fait composée d'une seule classe pour les classes simples (ex. la classe d'entité Projet regroupe une seule classe simple de même nom Projet) ou de plusieurs classes pour les versions de classe (ex. la classe d'entité Avions regroupe les versions de classe Avions.v0 et Avions.v1) ; ceci est traduit par un lien de composition entre la classe *Classe de classe* et la classe *Classe* (ayant deux sous-classes *Version de classe* et *Classe simple*). Les classes *Version de classe* et *Classe simple* décrivent les classes d'instances (d'objets ou de versions d'objets) définies par l'utilisateur.

D'autre part, la classe *Entité* décrit les différentes entités représentées dans la base de données. Chaque entité est en fait composée d'une instance (pour les objets) ou de plusieurs instances (pour les versions d'objets) ; ceci est traduit par un lien de composition entre la classe *Entité* et la classe *Instance* (ayant deux sous-classes *Version* et *Objet*). Les classes définies par l'utilisateur seront des sous-classes des

classes *Version* et *Objet*. La relation de dérivation pour les versions d'objets est représentée par le lien d'association sur la classe *Version*. Le lien d'association *est_instance* entre les classes *Instance* et *Classe*, à partir d'une instance, permet de retrouver l'objet qui décrit sa classe.

Du point de vue dynamique, les opérations définies pour les classes et les instances sont modélisées par des méthodes au niveau des classes correspondantes.

Les différents groupes de liens définis dans la base de données sont décrits au niveau de la classe *Classe de lien*. Un lien est en réalité composé de plusieurs versions de lien si le lien est versionnalisable ou d'un seul lien pour un lien simple ; ceci est traduit par le lien de composition entre la classe *Classe de lien* et la classe *Lien* (ayant deux sous-classes, *Lien simple* pour les liens non versionnalisables et *Lien versionnalisable* s'il y a des versions de lien).

Chaque lien entre classes est décrit au niveau de la classe *Lien* ; il relie des classes, avec une cardinalité pour chaque classe (traduit par l'association *relier* entre la classe *Lien* et la classe *Classe*).

Chaque instance de lien lie des instances (traduit par l'association entre les classes *Instance lien* et la classe *Instance*) ; on retrouve les informations sur le lien (cardinalités, classes liées) via l'association entre les classes *Instance lien* et *Lien*.

Le modèle ainsi défini décrit les concepts minimaux permettant la gestion de versions d'objets et de classes en plus de la gestion traditionnelle. D'autres types de classes ou d'instances peuvent être définis par spécialisation des classes existantes (*Classe*, *Instance*, ...) permettant d'étendre le SGBDOO hôte avec des concepts supplémentaires.

Pour définir une base de données intégrant la gestion de versions, les classes du méta-modèle sont instanciées comme suit :

- les objets instances de la classe *Classe de classe* décrivent les classes d'entités représentées dans la base,
- les objets instances des classes *Classe simple* et *Version de Classe* décrivent les classes spécifiées par l'utilisateur comme des objets,
- les classes définies par l'utilisateur sont considérées comme des sous-classes des classes *Objet* et *Version* suivant le type d'instance choisi,
- pour chaque classe définie par l'utilisateur une classe est définie comme sous-classe de la classe *Entité* ; les objets de chaque sous-classe décrivent les entités représentées dans la classe utilisateur correspondante.

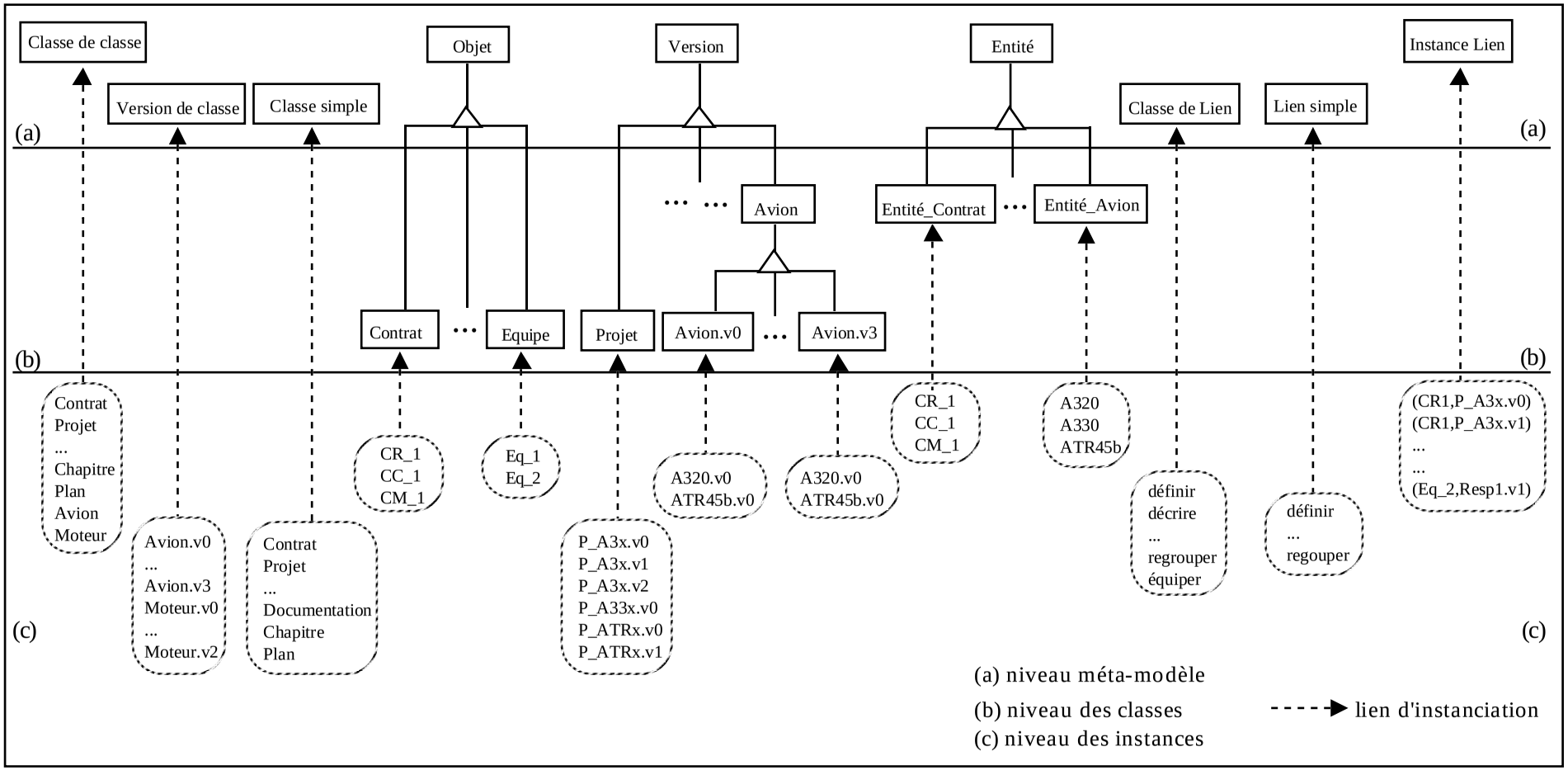


Figure IV-3 : Instanciation du méta-modèle

2.2. Le langage de manipulation

Le langage est mis en oeuvre au sein d'un module indépendant comme un outil supplémentaire fourni à l'utilisateur ; en effet, le méta-modèle suffit pour gérer les versions et les manipuler à l'aide des méthodes spécifiées. Le module langage constitue une interface utilisateur pour la gestion et surtout l'interrogation des versions en tenant compte de leurs spécificités.

Le langage est défini par une grammaire non contextuelle (cf. Annexe II.) ; ceci permet de mettre en oeuvre un analyseur lexical et un analyseur syntaxique à l'aide de générateurs tels que LEX et YACC. La requête soumise est vérifiée et ensuite traduite dans le langage du SGBD hôte en fonction des classes implantées. Les commandes sont ensuite exécutées par le SGBDOO hôte. Les résultats sont interprétés pour correspondre au schéma manipulé par l'utilisateur c'est-à-dire spécifié dans notre modèle intégrant les versions.

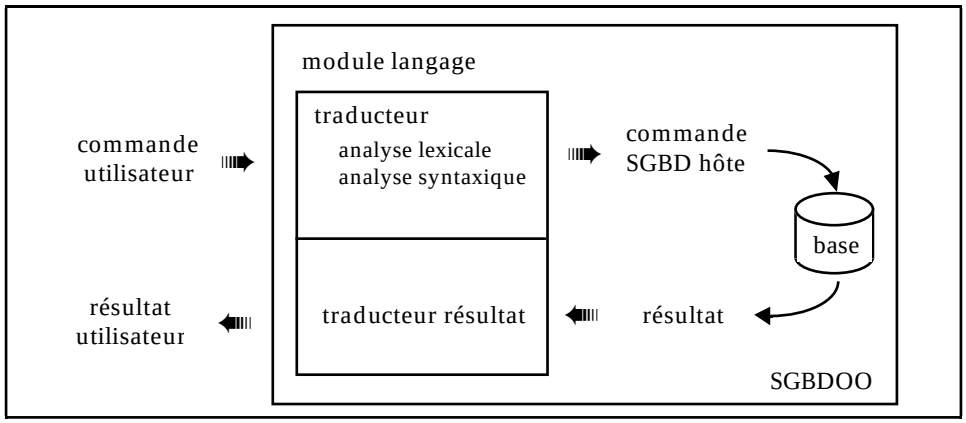


Figure IV-4 : Implantation du module langage

2.3. La mise en oeuvre des contraintes

Pour la mise en oeuvre des contraintes inhérentes à notre modèle (cf. Chapitre II) nous avons opté pour les possibilités intéressantes offertes par les bases de données actives. Ce type d'approche est utilisé dans d'autres systèmes tels que ARIS (Amghar, 94)(Bouaziz, 95) et SHOOD (Bounaas, 95). La gestion des contraintes est réalisée dans un module indépendant ; les contraintes sont exprimées sous forme de règles ECA (Evénement Condition Action). Le système ainsi défini permet en plus une plus grande extensibilité. Nous intégrons les apports des bases de données actives en utilisant les travaux réalisés au sein de notre équipe avec l'outil Urdos basé sur le concept de schéma actif (Tchounikine, 93a).

La plupart des travaux réalisés dans le domaine des bases de données actives consistent à développer une couche active intégrée dans un SGBD particulier, comme par exemple les systèmes EXACT (Diaz, 91) ou HiPAC (Dayal, 88). L'approche

de Urdos est différente. Au lieu de construire un système de règles pour un SGBD spécifique, il consiste à définir un outil. Un SGBD passif utilisant cet outil, peut de gérer des bases de données actives. L'outil correspond à un ensemble minimal de fonctionnalités nécessaires à l'intégration de la composante active dans une BD. La solution se veut universelle (Tchounikine, 93a).

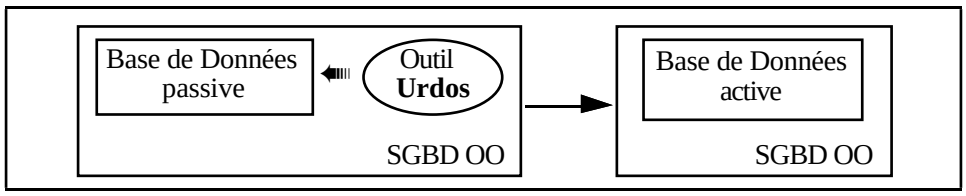


Figure IV-5 : Principe de l'outil URDOS

La définition de l'outil Urdos respecte les quatre points suivants :

- utilisation exclusive des concepts et mécanismes communs aux SGBDOO,
- respect de l'intégrité des objets ; l'introduction de règles n'affecte pas les objets contrairement à la plupart des SGOA qui modifient les objets pour y intégrer l'activité,
- prise en compte de la dynamique de la base ; l'objectif est de permettre une flexibilité optimale des objets actifs et sans rendre immuable le caractère actif ou passif des objets comme c'est le cas dans la plupart des SGOA,
- définition simple des règles ; l'utilisateur ne doit pas avoir à programmer les règles mais plutôt les définir de manière homogène aux autres objets de la base.

Le schéma Urdos est construit comme une hiérarchie de classes qui sera importée dans le schéma de l'application pour laquelle l'activité est désirée. Les règles sont considérées comme des objets : leur structure est décrite par un schéma ; les règles sont créées par instanciation d'attributs. De plus, les règles sont considérées comme des objets complexes, c'est-à-dire des objets composés d'autres objets correspondant aux événements ; les événements sont des objets.

La modélisation de l'activité est réalisée au travers d'une classe pour les règles ECA liées par un lien de composition à une classe d'événements. Des méthodes sont ensuite associées aux classes définies pour gérer règles et événements. L'interface utilisateur ainsi que les mécanismes d'inférences sur les règles sont également réalisés à l'aide de méthodes.

Ainsi, les règles de comportement décrivant l'activité des objets sont décrites à l'aide de classes auxquelles sont associées des méthodes pour gérer l'activité, c'est-à-dire créer les règles et réaliser l'inférence. C'est ce schéma de classes qui est appelé Schéma Actif (Tchounikine 93b). Ce schéma peut être vu comme un méta-modèle, qui sera intégré dans le méta-schéma d'une application comme les classes racines des hiérarchies d'héritage.

L'utilisation du Schéma Actif ne modifie pas la structure des autres objets ; il fonctionne de manière autonome, c'est-à-dire sans nécessiter d'autre mécanisme que ceux qu'il contient déjà en lui-même. L'utilisateur crée ensuite les instances de règles nécessaires pour son application ; il utilise pour cela les méthodes de création qui lui sont fournies avec le schéma.

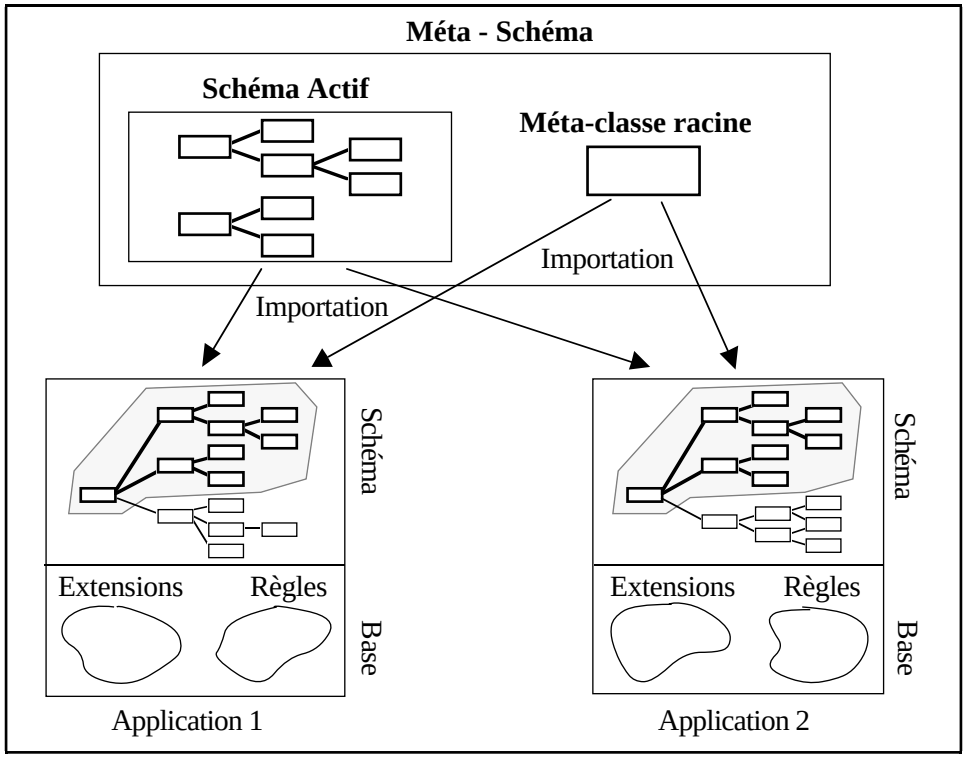


Figure IV-6 : Architecture générale d'Urdos

3. Expérimentation

Nous avons expérimenté nos solutions d'implantation à l'aide du SGBD O2 fournissant les concepts communs aux SGBD orienté objet. Ceci nous a permis d'étudier la faisabilité des solutions que nous proposons pour l'intégration de la gestion de versions dans un SGBD. Cette expérimentation est réalisé au travers d'une interface graphique, baptisée VOHQL, destinée à des utilisateurs occasionnels ; cette interface permet notamment de définir graphiquement des requêtes.

3.1. Le SGBD hôte

Le SGBD hôte choisi est le SGBD O2 (Bancilhon, , 92) commercialisé par la société O2 Technology, projet initié par le Groupement d'Intérêt Général Altaïr en 1986. La version utilisée pour l'expérimentation est la version 4.5 multi-utilisateurs.

Le SGBD propose un langage de programmation, appelé O2C, qui repose sur le langage C étendu aux notions de classes, d'objets ... La gestion de la persistance est réalisée au travers d'objets nommés contenant les ensembles d'instances. Le SGBD O2 propose également un langage d'interrogation, OQL précédemment nommé O2SQL dont la syntaxe s'inspire de SQL ; de plus, le langage OQL est le langage de requêtes du standard ODMG-93 (Atwood, 94).

Le SGBD O2 propose toutes les caractéristiques d'un SGBD orienté objet (classe, objets, méthodes, héritage, ...) ainsi qu'un langage de requêtes de type SQL ; l'utilisation est simple et conviviale. De plus, plusieurs développements ont déjà été réalisés au sein de notre équipe à l'aide du SGBD O2 ; c'est le cas notamment de l'outil Urdos, avec lequel nous gérons les contraintes d'intégrité.

3.2. L'architecture du système VOHQL

Le prototype que nous avons réalisé, baptisé VOHQL (Version and Object Hypertext Query Language) suit les principes d'implantation fixés précédemment (cf. § 2.). L'architecture logicielle est la suivante :

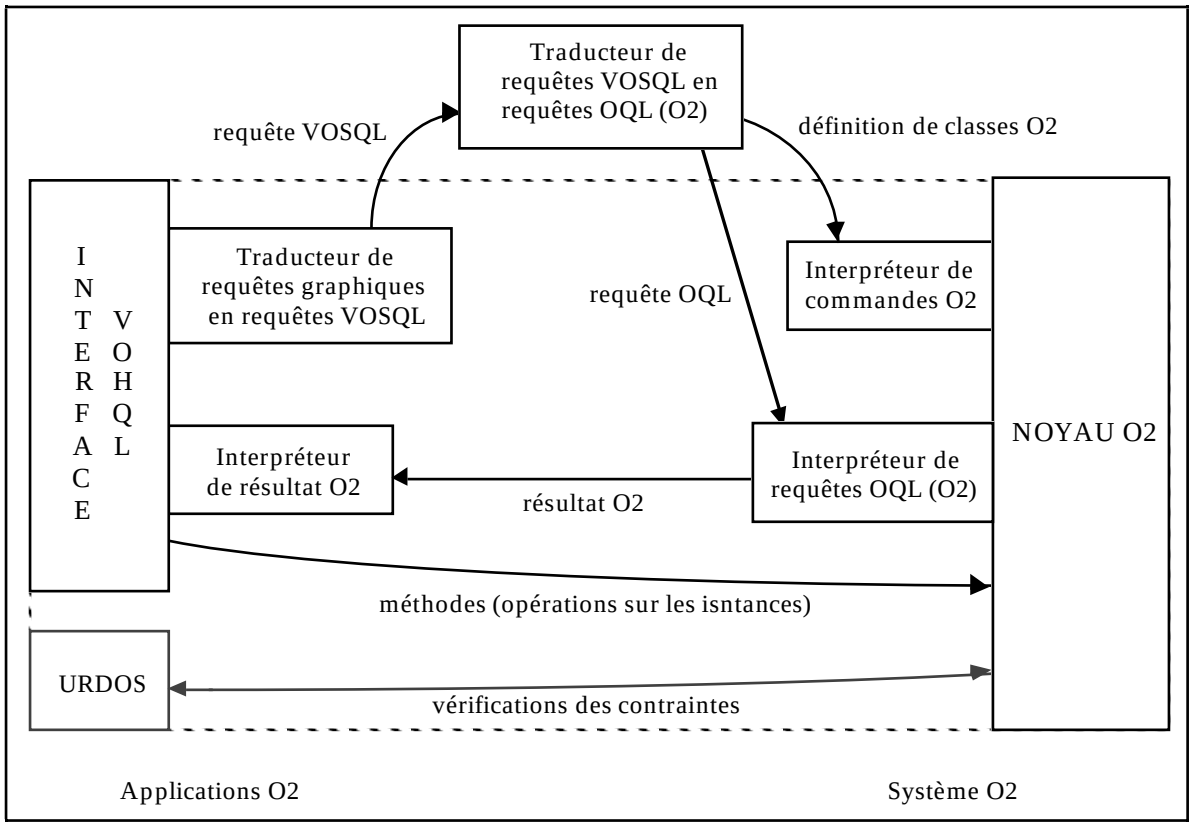


Figure IV-7 : Architecture générale du système VOHQL

L'interface VOHQL est définie comme une application O2. Elle comprend deux modules pour l'exécution des requêtes graphiques :

- un module qui génère la requête VOSQL correspondant à la requête graphique exprimée au niveau de l'interface,
- un module qui interprète le résultat fournit par le SGBD O2.

L'interface utilise un module de traduction d'une requête VOSQL en requête OQL interprétable par le système O2. Ce module, étant réalisé avec Lex et Yacc, est externe aux applications O2. Le module actif URDOS, pour la gestion des contraintes, est également une application O2 ; son intégration dans le prototype est actuellement en cours.

3.3. Le méta-schéma

Les classes du méta-schéma définies dans le SGBD hôte O2 correspondent aux classes d'entités du méta-modèle (cf. § 3.). Le SGBD O2 n'offre pas explicitement les concepts de liens de composition et d'association ; ceux-ci sont définis à l'aide d'attributs dans les classes liées.

```
schema Gestion_Version;
class Classe_d_entité  public type tuple (
                nom : string,
                nb_version : integer,
                pr_version : integer,
                typ_défaut : char)
        method public ajouter_classe : integer;
end
…
class Version  inherit Instance
        public type tuple (
                créateur : string,
                num_version : integer,
                état : char,
                date : Date,
                dérive_de : Version)
        method    dériver : integer
                  …
                  élaguer (num_vers1, num_vers2 : integer) : integer
        end
…
```

Le schéma ainsi créé sous O2 est utilisé suivant le principe d'importation de schéma. Les classes du schéma *Gestion_Version* sont importées dans le schéma utilisateur créé ; ainsi importées, elles sont directement instanciables ou sont utilisées en tant que super-classe de classes créées.

3.4. L'exécution des requêtes

Le traitement d'une requête, définie graphiquement au niveau de l'interface et générée sous forme de requête VOSQL, peut être schématisée comme suit :

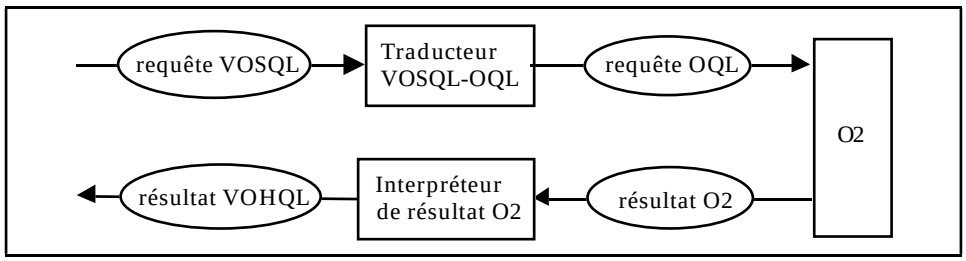


Figure IV-8 : Exécution d'une requête VOHQL

La validité de la requête VOSQL est vérifiée par le compilateur défini à l'aide d'une grammaire non contextuelle Lex & Yacc (cf. Annexe II), selon la méthode d'analyse syntaxique ascendante LALR (Aho, 77). La traduction de la requête VOSQL en requête OQL(O2) est également réalisée par ce même compilateur à l'aide de règles de production associées à la grammaire Lex & Yacc.

La traduction d'une requête sur une classe de versions (ex. Projet) utilise la partie du méta-schéma suivante (cf. § 2.1.) :

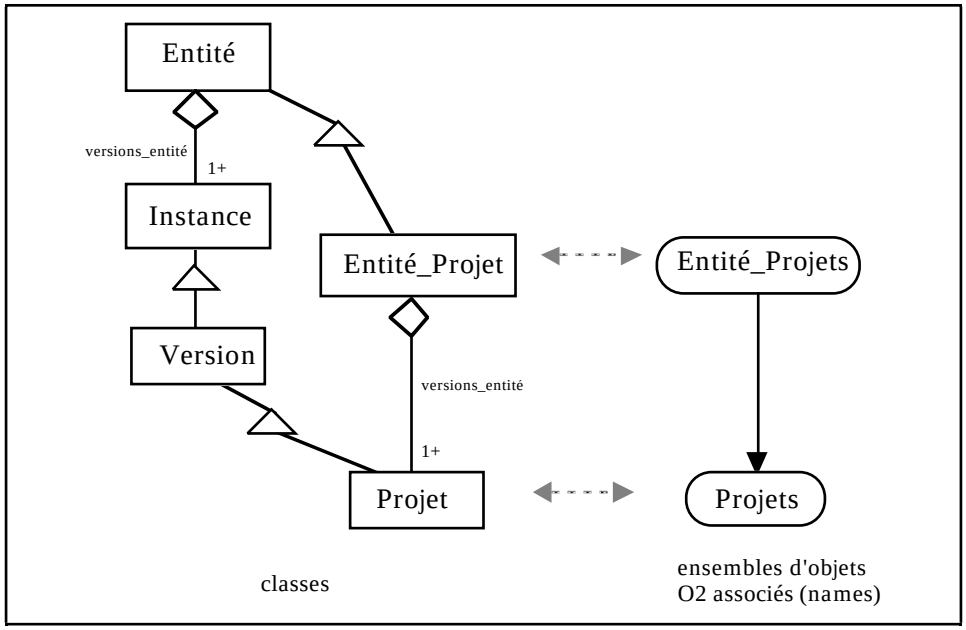


Figure IV-9 : Classes utilisées pour la traduction de requêtes

Chaque entité projet décrite est composée d'un ensemble de versions (lien *versions_entité*). Les entités projets sont décrites dans une classe *Entité_Projet* sous-classe de la classe *Entité* du méta-schéma. Les versions des entités projets sont décrites dans la classe *Projet* sous-classe de la classe *Version*, elle-même sous-classe de *Instance*, du méta-schéma. Dans le système O2, les instances des classes sont conservées dans des ensembles d'objets associés aux classes (ex. l'ensemble

Entité_Projets est associé à la classe *Entité_Projet*, l'ensemble *Projets* est associé à la classe *Projet*).

Le principe de traduction est le suivant :

- pour des requêtes portant sur des versions indépendantes (c'est-à-dire sans considérer les liens de dérivation), de la forme :

```
requête VOSQL :  select v
                 from v in Untree(Unforest(Projet))
                 where v->durée = 2
```

L'interrogation s'effectue dans O2 sur l'ensemble d'objets qui contient les versions (*Projets* associé à la classe implantée Projet de même nom que la classe interrogée). Le prédicat de sélection reste le même. La requête OQL soumise à O2, est la suivante :

```
requête OQL :  select v
               from v in Projets
               where v->durée = 2
```

Pour interpréter le résultat, il suffit d'afficher chaque élément de l'ensemble de versions obtenu.

- pour des requêtes portant sur des hiérarchies de versions, de la forme :

```
requête VOSQL :  select h
                 from h in Projet
                 where Forall v in Untree(h) : v->budget > 60
```

L'interrogation s'effectue dans O2 sur l'ensemble d'objets qui contient les entités représentées dans la classe interrogée (*Entité_Projets* associé à la classe *Entité_Projet* décrivant les entités représentées dans la classe interrogée). L'ensemble des versions composant chaque entité (correspondant à Untree(h)) est obtenu par le lien de composition nommé versions_entité (cf. Figure IV-9, lien de composition entre la classe *Entité* et la classe *Instance* spécialisé en lien de composition entre la classe *Entité_Projet* sous-classe de *Entité*, et la classe *Projet* de sous-classe de *Version* et *Instance*). Le prédicat de sélection est pratiquement le même (on remplace Untree(h) par h->versions_entité).

La requête OQL correspondante, soumise à O2, est la suivante :

```
requête OQL :  select h
               from h in Entité_Projets
               where for all v in h->versions_entité : v->budget > 60
```

Pour interpréter le résultat, il faut :

pour chaque élément ***a*** de l'ensemble d'objets (entités) obtenu

constituer la hiérarchie de dérivation, à partir des versions de l'entité obtenues par le lien de composition ***a***->versions_entité, en exploitant le lien de dérivation entre versions (cf. méta-modèle § 2.1. lien *dérive_de* sur la classe *Version*).

afficher la hiérarchie de dérivation.

Au niveau du prototype, la traduction des requêtes VOSQL couvre l'ensemble des requêtes graphiques exprimables (interrogation de versions, interrogation de hiérarchies de versions, interrogation suivant les liens de composition et d'association, interrogation sur plusieurs classes (jointure explicite), interrogation sur les critères internes (une partie des opérateurs sur les dates)). Il reste à réaliser la traduction de requêtes plus complexes (requêtes imbriquées c'est-à-dire des requêtes utilisant le résultat de sous-requêtes, des requêtes sur plusieurs versions de classe) et étendre le nombre d'opérateurs sur les dates (cf. Chapitre III. § 4.2.3.4.).

3.5. L'implantation du schéma actif Urdos

Le schéma actif Urdos est implanté à l'aide de classes (Tchounikine, 93a) décrivant :

- les classes d'événements,
- les classes de règles,
- les extensions des classes précédentes.

L'inférence est réalisée à l'aide de méthodes associées aux classes définies (*evt_detecte* définie sur la classe *Ext_Evt*, *activer_regle* définie sur la classe *Ext_regle*, ...) et d'une fonction *inference*.

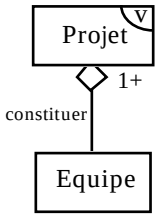
```
schema Urdos;

class Evenement  public type tuple (
                    public identifiant : string,
                    public source : integer,
                    public cond_occ : string,
                    public occurrence : set(Object))
end;

class  Ext_evt
       public type tuple (public instances : unique set (Evenement))
       method public evt_detecte (s:Object, N:integer) : set(Evenement)
end;

name evenements : Ext_evt;
```

```
class Regle  public type tuple (
                public nom : string,
                public evt : Evenement,
                public condition : string,
                public action : string)
        method public executer(R:list(Object), O:Object
        end;

class Ext_regle  public type tuple (public instances : unique set (Regle))
        method public activer_regle (E:Evenement, OA:Object, N:integer)
end;

name regles : Ext_regle;

function inference (om:Object);
```

Le schéma ainsi créé sous O2 est utilisé selon le principe d'importation de classe. Les classes du schéma Urdos sont importées dans le schéma utilisateur créé. Chaque règle est créée comme instance de la classe *Regle*. La vérification des contraintes sous-jacentes à notre modèle de données définies au chapitre 2 sont définies sous la forme de règles instances de la classe *Regle*.

L'intégration du schéma actif au prototype est en cours de développement. Les règles, traduisant les contraintes d'intégrité, seront générées automatiquement lors de la création du schéma utilisateur. Les classes du schéma actif seront instanciées lors de la phase de génération des classes définies par l'utilisateur ; chaque règle sera une instance.

Exemple IV-1 : Considérons le lien de composition entre la classe de versions *Projet* composée de la classe d'objets *Equipe* suivant :



Une règle vérifiant les contraintes relatives à la cardinalité 1-1 du lien de composition pour la classe *Projet* est définie comme suit :

nom : integrite_compo
evt : **identifiant** : creation_modif
source : Projet
cond_occ :
condition : count($1->constituer-Equipe) = 1
action : $1->erreur

Cette règle est appliquée lors de la création ou la modification d'une version composée.

La génération d'une telle règle dans le prototype URDOS sous O2 s'effectue de la manière suivante, en créant des nouvelles instances des classes Evenement et Regle (Cf. schéma URDOS précédent) :

```
run body
    {
        o2 Evenement evt_reg;
        o2 Regle nouv_regle;

        -- définition de l'événement déclencheur de la règle
        evt_reg = new(Evenement);
        evt_reg.identifiant = "définition composé"
        evt_reg.source = "Projet"        -- classe concernée

        -- ajout de l'événement dans l'ensemble des événements gérés
        Ext_evt->instances += evt_reg;

        -- définition de la nouvelle règle
        nouv_regle = new(Regle);
        nouv_regle.nom = "creation_modif";
        nouv_regle.evt = evt_reg;
        nouv_regle.condition = "count($1->constituer-Equipe) = 1"
        nouv_regle.action = "$1->erreur"

        -- ajout de la nouvelle règle dans l'ensemble des règles
        Ext_regle->instances += nouv_regle;
    }
```

3.6. L'interface graphique VOHQL

L'interface graphique VOHQL (Le Parc, 96a) est une interface pour bases de données orientées objet intégrant des versions et destinée à des utilisateurs occasionnels. L'interface VOHQL est basée sur notre modèle et son langage. L'interface permet de créer et de manipuler graphiquement une BDOO intégrant des versions. L'interface inclut également le module de langage textuel VOSQL de base.

L'interface VOHQL reprend les principes de manipulation d'interfaces graphiques développées dans notre équipe. Il s'agit d'interfaces pour la manipulation de BDOO comme OHQL(Canillac, 91)(Andonoff, 93a)(Andonoff, 94b) et CHOLQ(Sallaberry, 92) (Andonoff, 93b)(Andonoff, 94a), et pour la manipulation de BDOO réparties comme D_OHQL(Hubert, 94).

3.6.1. La création d'une base intégrant les versions

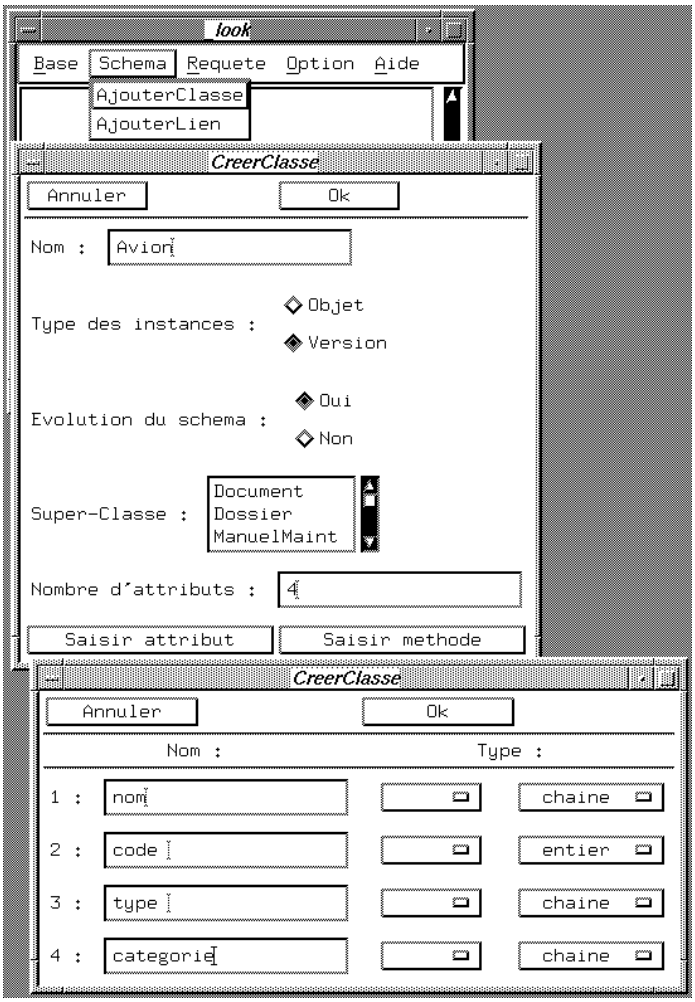
Deux possibilités sont offertes à l'utilisateur pour créer une BDOO intégrant des versions :

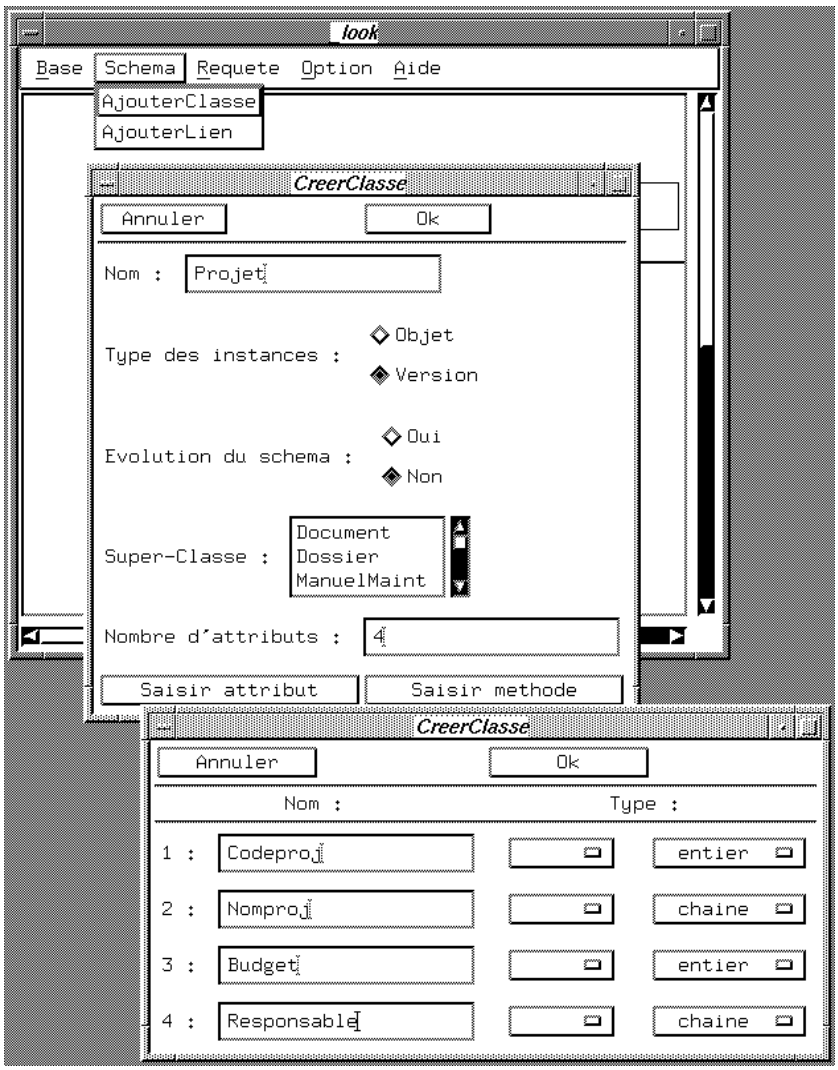
- en utilisant le langage VOSQL suivant la syntaxe définie au chapitre III.
- intéractivement en créant chaque classe et chaque relation entre classes

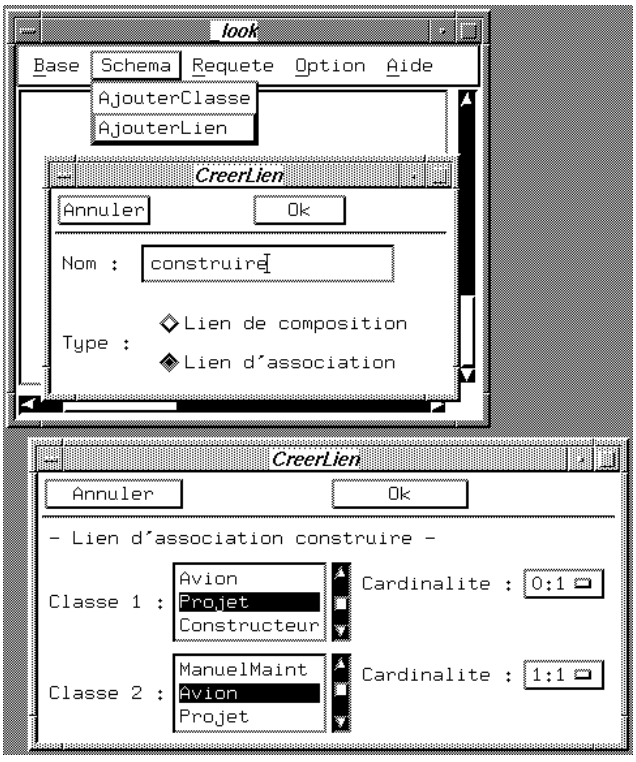
Une fenêtre de dialogue permet de saisir les différentes caractéristiques d'une classe : le nom de la classe, si les instances sont des versions ou des objets, si l'on autorise les versions de classe, les super-classes, le nombre d'attributs. Deux fenêtres permettent ensuite de spécifier les attributs (nom et domaine) et les méthodes.

Chaque lien de composition et d'association est défini dans une autre fenêtre de dialogue. Les classes liées sont indiquées avec leurs cardinalités pour le lien.

Exemple IV-2 : Considérons la création de la classe *Avion* pour laquelle on gère des versions au niveau du schéma et des instances, et de la classe simple de versions *Projet* ; les deux classes sont reliées par le lien d'association *construire*.




look
Base
Schema
Requete
Option
Aide
AjouterClasse
AjouterLien
CreerClasse
Annuler
Ok
Nom :
Projet
Type des instances :
Objet
Version
Evolution du schema :
Oui
Non
Super-Classe :
Document
Dossier
ManuelMaint
Nombre d'attributs :
4
Saisir attribut
Saisir methode
CreerClasse
Annuler
Ok
Nom :
Type :
1 :
Codeproj
entier
2 :
Nomproj
chaine
3 :
Budget
entier
4 :
Responsable
chaine


look
Base
Schema
Requete
Option
Aide
AjouterClasse
AjouterLien
CreerLien
Annuler
Ok
Nom :
construire
Type :
Lien de composition
Lien d'association
CreerLien
Annuler
Ok
- Lien d'association construire -
Classe 1 :
Avion
Projet
Constructeur
Cardinalite :
0:1
Classe 2 :
ManuelMaint
Avion
Projet
Cardinalite :
1:1

Cette création graphique de classes et de liens correspond à la déclaration textuelle suivante :

create simple class Projet
contains versions
attributes (Codeproj : integer, Nomproj : string,
Budget : integer, Responsable : string)

create class version Avion
contains versions
attributes (nom : string, code : integer, type : string, categorie : string)

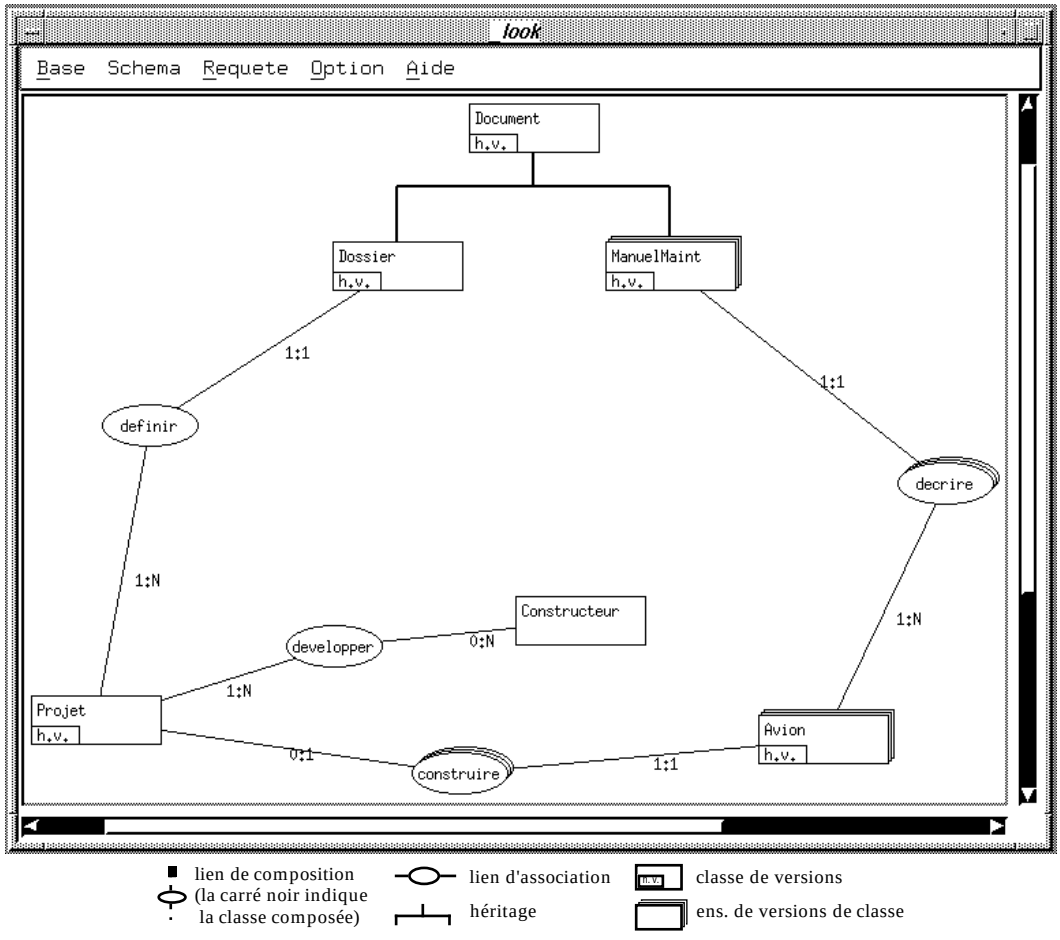
relationship construire **:** Projet (0:1) **and** Avion (1:1)

3.6.2. La représentation du schéma de la base

Une base de données est représentée sous la forme d'un graphe :

- les noeuds décrivent les classes,
- les liens décrivent les relations entre classes.

Exemple IV-3 : La représentation graphique de la base exemple Construction_Avion est la suivante dans l'interface VOHQL :



Le graphe constitue le point de départ de toute manipulation. Une interface est associée à chaque noeud sous la forme d'un menu proposant les opérations applicables.

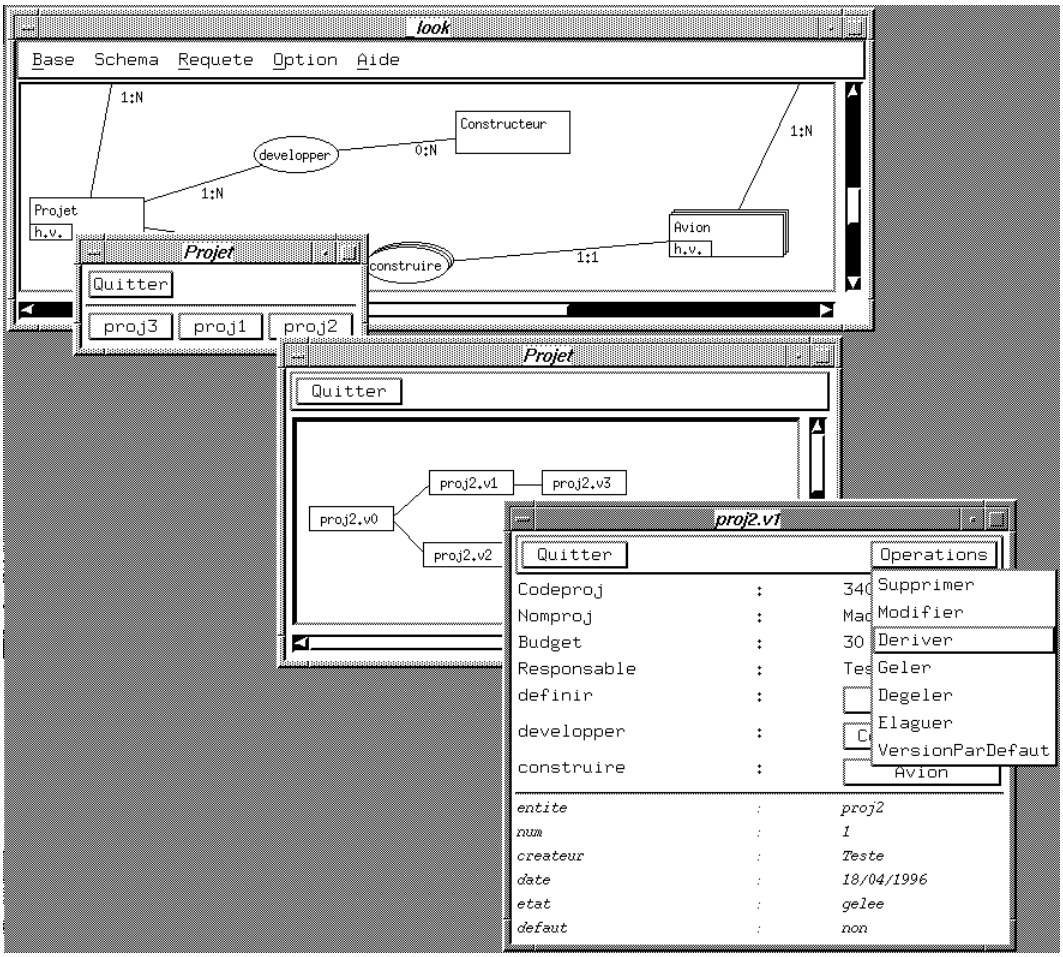
3.6.3. La création des instances

La création d'une instance est réalisée graphiquement en appliquant l'opération de création dans l'interface des classes (noeuds). La création est effectuée lorsque l'utilisateur la valide (bouton Valider). Les contraintes sur les instances sont alors vérifiées (fin de transaction).

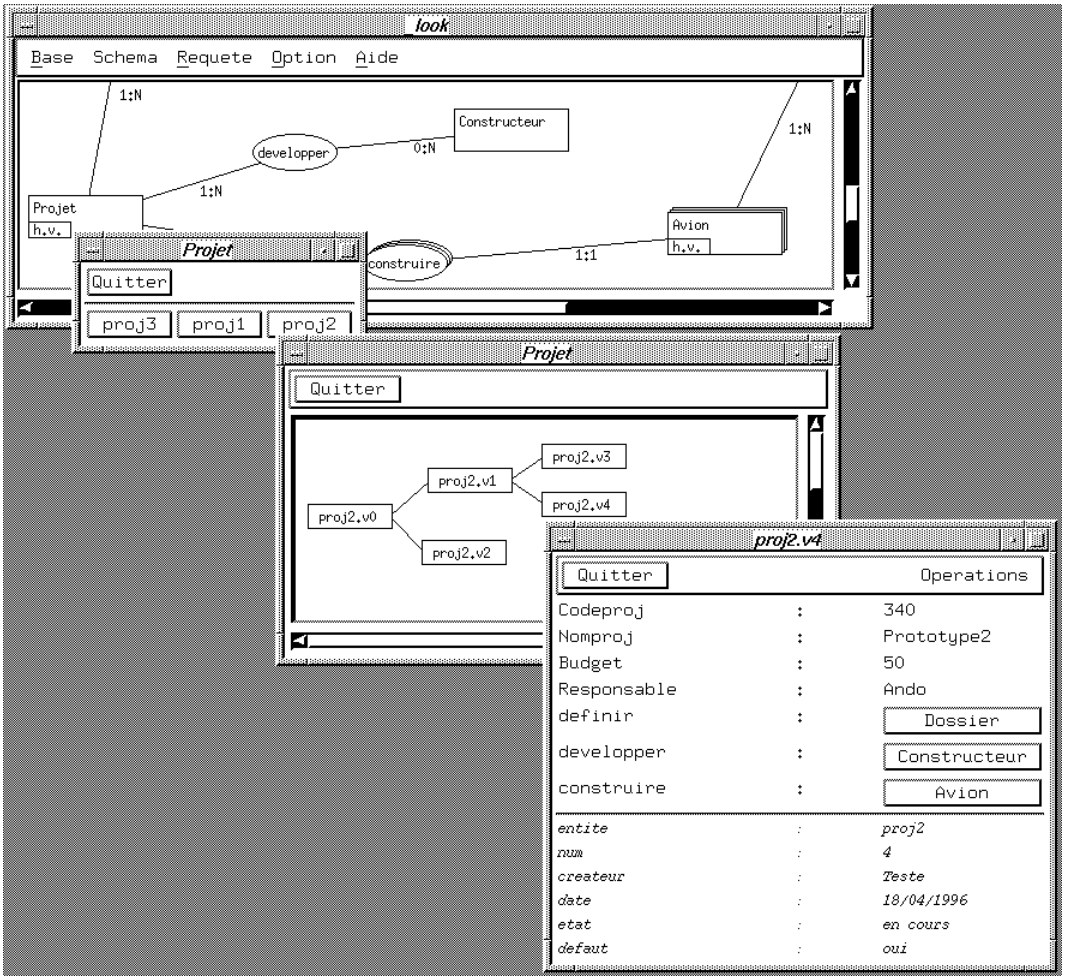
La création graphique d'une version par dérivation d'une version existante est réalisée en deux étapes :

- l'utilisateur se positionne sur la version à dériver,
- il exécute l'opération de dérivation disponible dans l'interface de l'instance.

Exemple IV-4 : Considérons la création, dans la classe Projet, de la version 4 du projet 2 par dérivation de la version 1.



Le déclenchement de l'opération de dérivation crée une nouvelle version en reprenant les valeurs de la version dérivée. L'utilisateur modifie ensuite les valeurs des attributs pour la nouvelle version.



Cette dérivation correspond à la déclaration suivante dans le langage VOSQL :

create instance in *Projet*
derived from *proj2.v1*
values (*Nomproj* : Prototype2, *Budget* : 50, *Responsable* : Ando **)**

La nouvelle version reste liée aux mêmes instances des autres classes.

3.6.4. L'interrogation d'une base

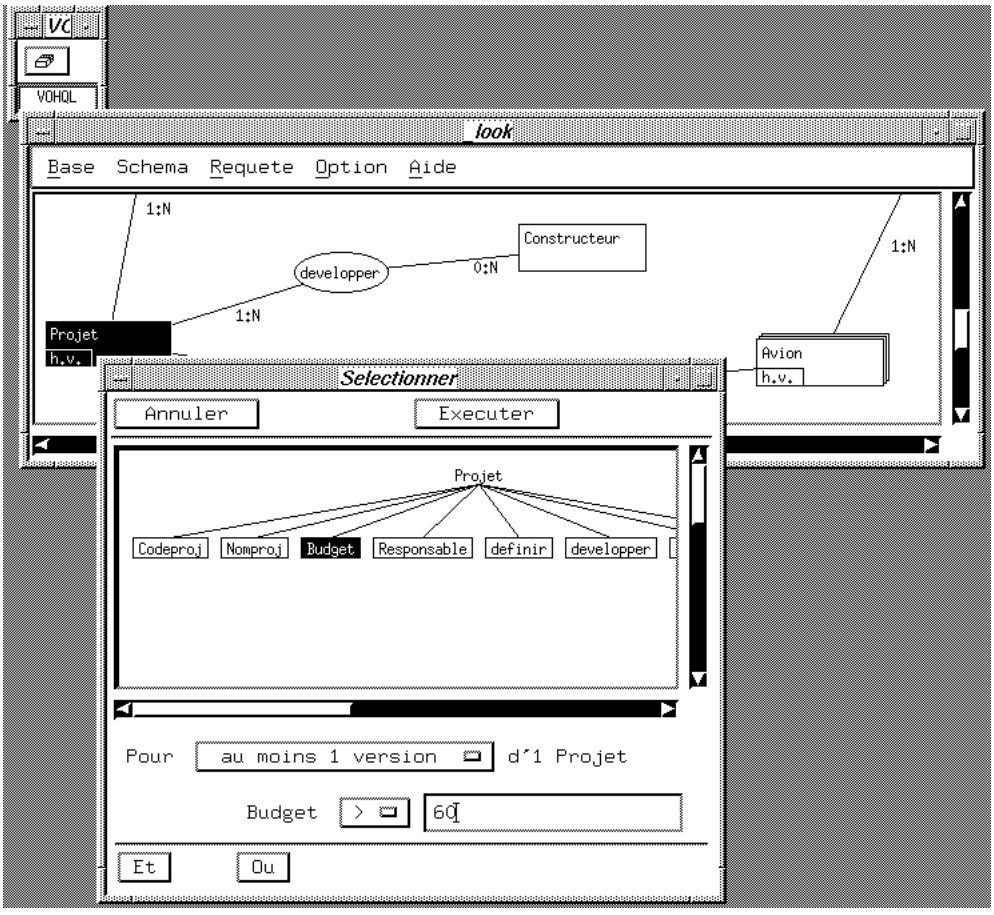
La construction d'une requête graphique s'effectue directement sur le graphe représentant la base. L'utilisateur fixe les prédicats de la requête en déclenchant l'opération de Sélection dans l'interface de chaque classe. Le résultat d'une requête est visualisé au travers des classes du graphe ; seules les instances conservées par la requête apparaissent dans leur classe respective.

3.6.4.1. Interrogation de forêts de dérivation

La définition d'un prédicat s'effectue en plusieurs étapes :

- l'utilisateur choisit l'attribut dans un arbre représentant la structure de la classe,
- il précise le quantificateur éventuel,
- il choisit l'opérateur de comparaison,
- il indique la valeur de comparaison,
- il choisit éventuellement un opérateur logique (ET ou OU) pour définir une autre condition suivant le même processus,
- il valide la sélection (bouton Exécuter)

Exemple IV-5 : Considérons la construction d'une requête permettant d'obtenir les projets dont le budget a déjà dépassé les 60 Gf.



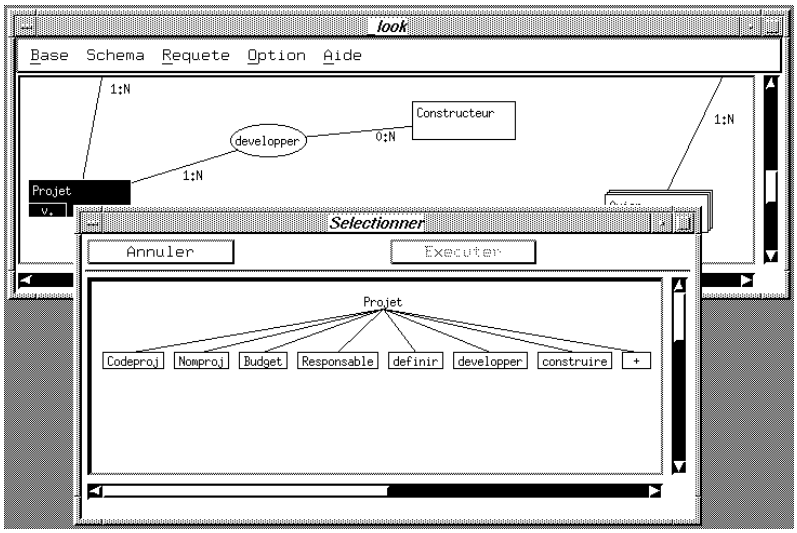
Cette requête correspond à la requête VOSQL suivante :

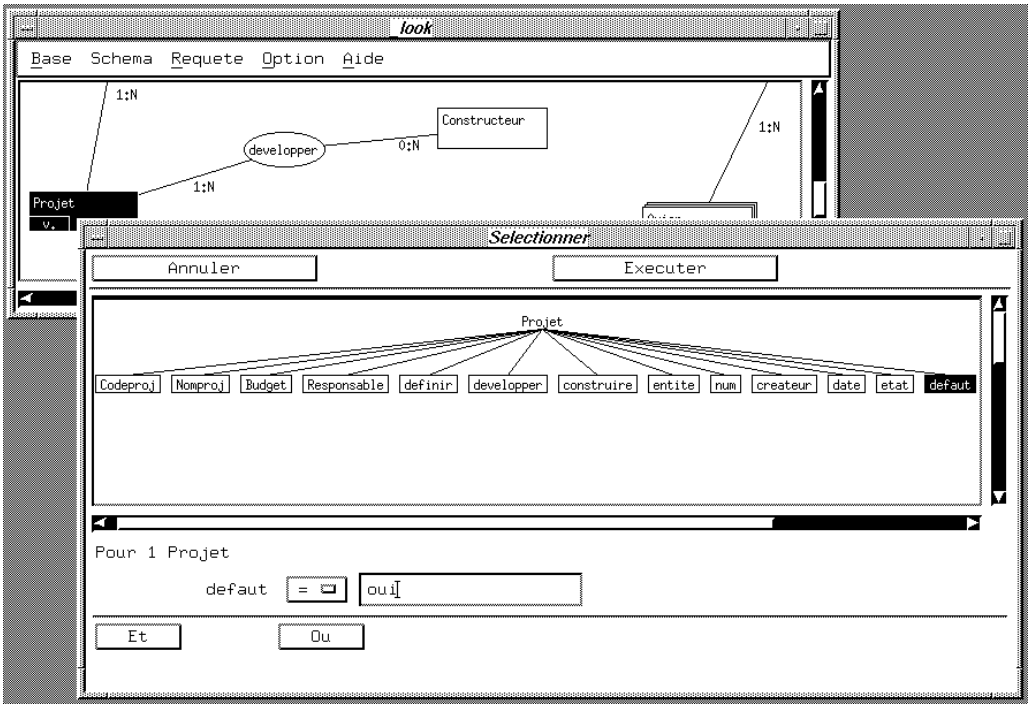
select h
from h **in** Projet
where Exist v **in Untree(**h**)** : v->Budget > 60

3.6.4.2. Interrogation de versions

Pour obtenir des versions (et non des hiérarchies entières comme précédemment) vérifiant une condition, il faut déstructurer l'ensemble interrogé. Pour cela, l'utilisateur précise l'organisation des versions à l'aide d'un bouton au niveau du noeud représentant la classe concernée : le bouton en position **v.** indique que l'on interroge les versions indépendamment les unes des autres, tandis que la position par défaut **h.v.** (exemple précédent) indique que l'on interroge des hiérarchies. La définition du prédicat de sélection reste basée sur le même principe.

Exemple IV-6 : Considérons la construction d'une requête permettant d'obtenir les versions projets par défaut. L'utilisateur affiche pour cela les attributs internes (cf. Chapitre II § 4.2.3.3.) dans l'arbre représentant la structure (bouton **+**).





Cette requête correspond à la requête VOSQL suivante, qui peut être directement saisie dans le module textuel :

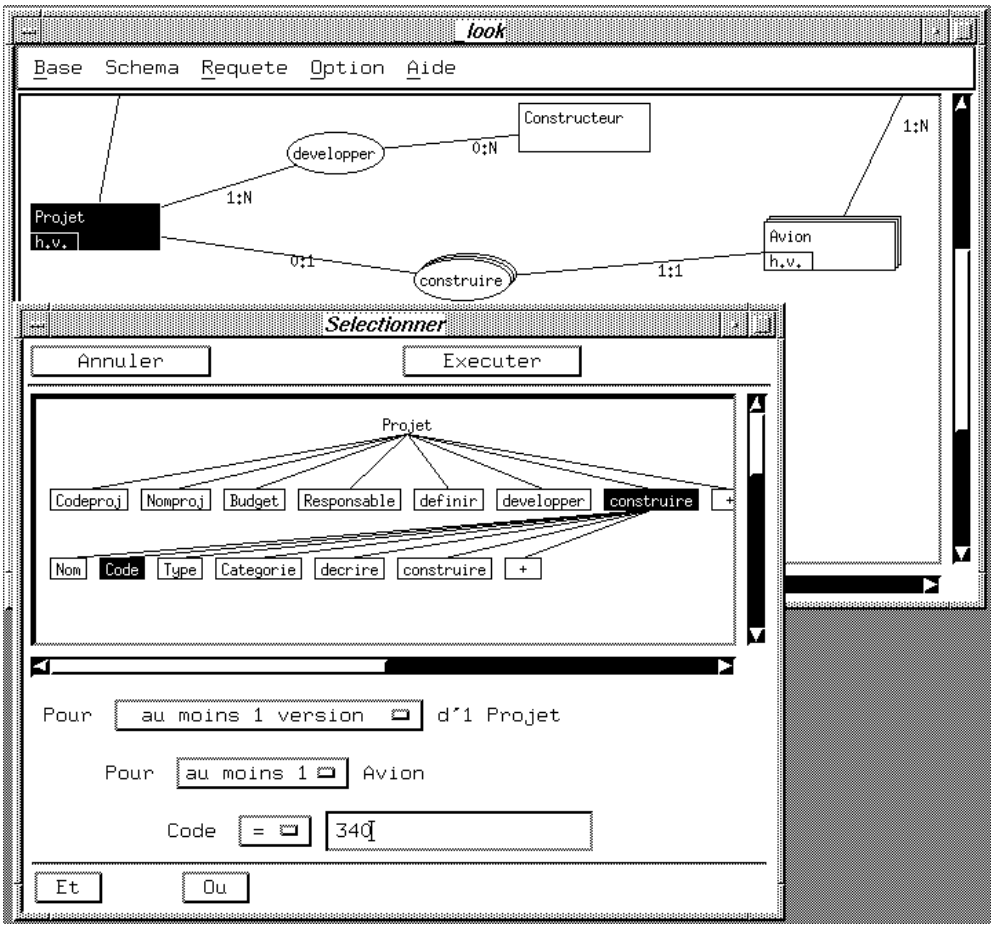
select v
from v **in Untree(Unforest(**Projet**)**
where Default(v**)**

3.6.4.3. Interrogation suivant les liens

Les liens de composition et d'association sont utilisables de deux manières :

- lors de la définition d'un prédicat de sélection pour spécifier des conditions sur les instances liées (exemple IV-7). Lorsque l'utilisateur exécute une Sélection sur une classe, il peut choisir des attributs des classes liées, dans l'arbre représentant la structure de cette classe.
- pour faire des jointures entre classes c'est-à-dire pour conserver dans chacune des classes uniquement les instances liées (exemple IV-8). Pour cela l'utilisateur clique sur le lien entre les deux classes dans le graphe représentant la base.

Exemple IV-7 : Considérons la requête graphique suivante permettant d'obtenir les projets dont toutes les versions ont été associées à la construction d'un avion portant le code 340.

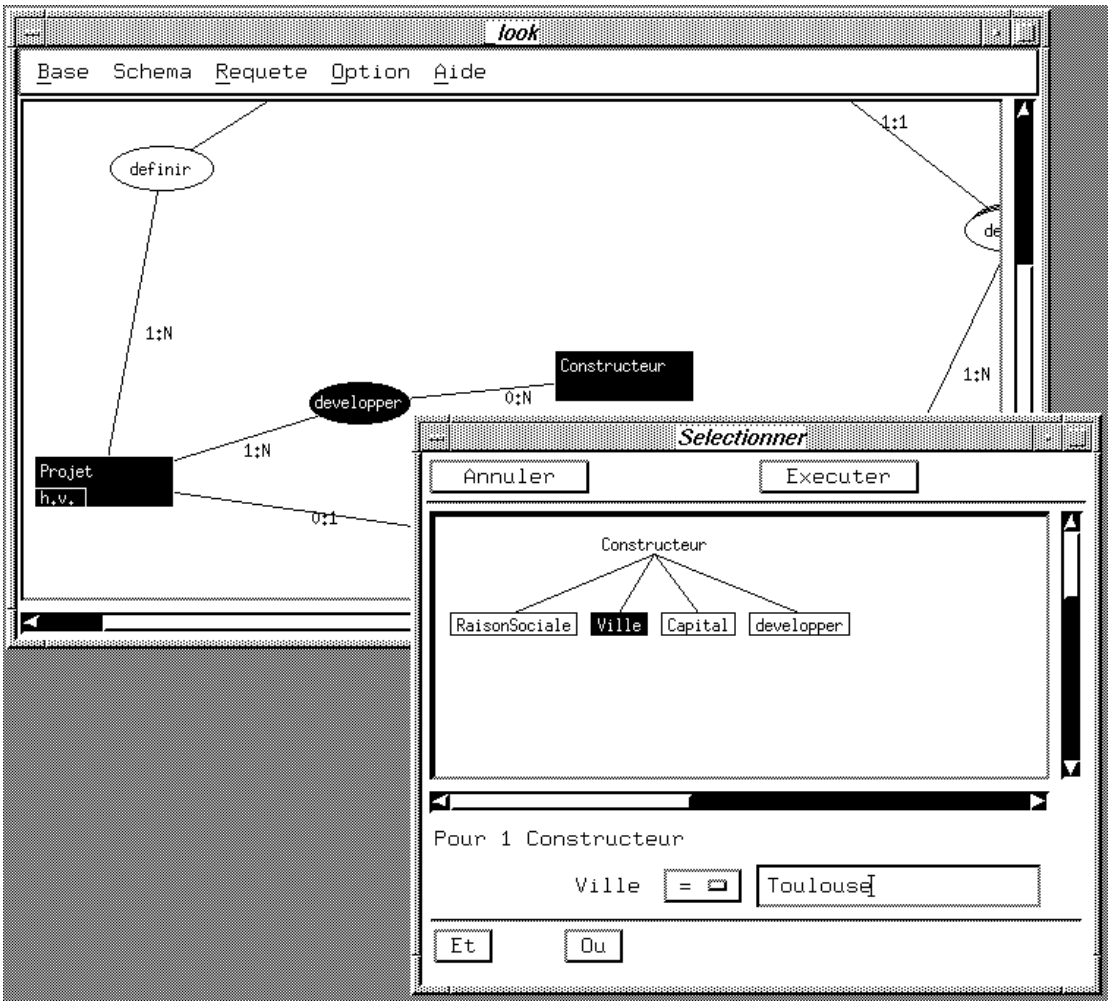


La requête en langage VOSQL correspondante est définie comme suit :

select h
from h **in** Projet
where Forall v **in Untree**(h) : v->construire->Code = 340

Exemple IV-8 : Considérons la requête graphique suivante permettant d'obtenir les constructeurs toulousains et les projets auxquels ils ont participé.

L'utilisateur réalise tout d'abord une sélection sur la classe *Constructeur* pour ne conserver que ceux se situant à Toulouse. Le lien *développer* est sélectionné pour conserver les projets et les constructeurs associés.



La requête en langage VOSQL correspondante est définie comme suit :

select h (ou **select** c si l'on visualise les instances de la classe Constructeur)
from c **in** Constructeur, h **in** Projet
where c->ville = "Toulouse"
and **Exist** v **in Untree**(h) : c in v->développer

3.6.5. La visualisation des instances

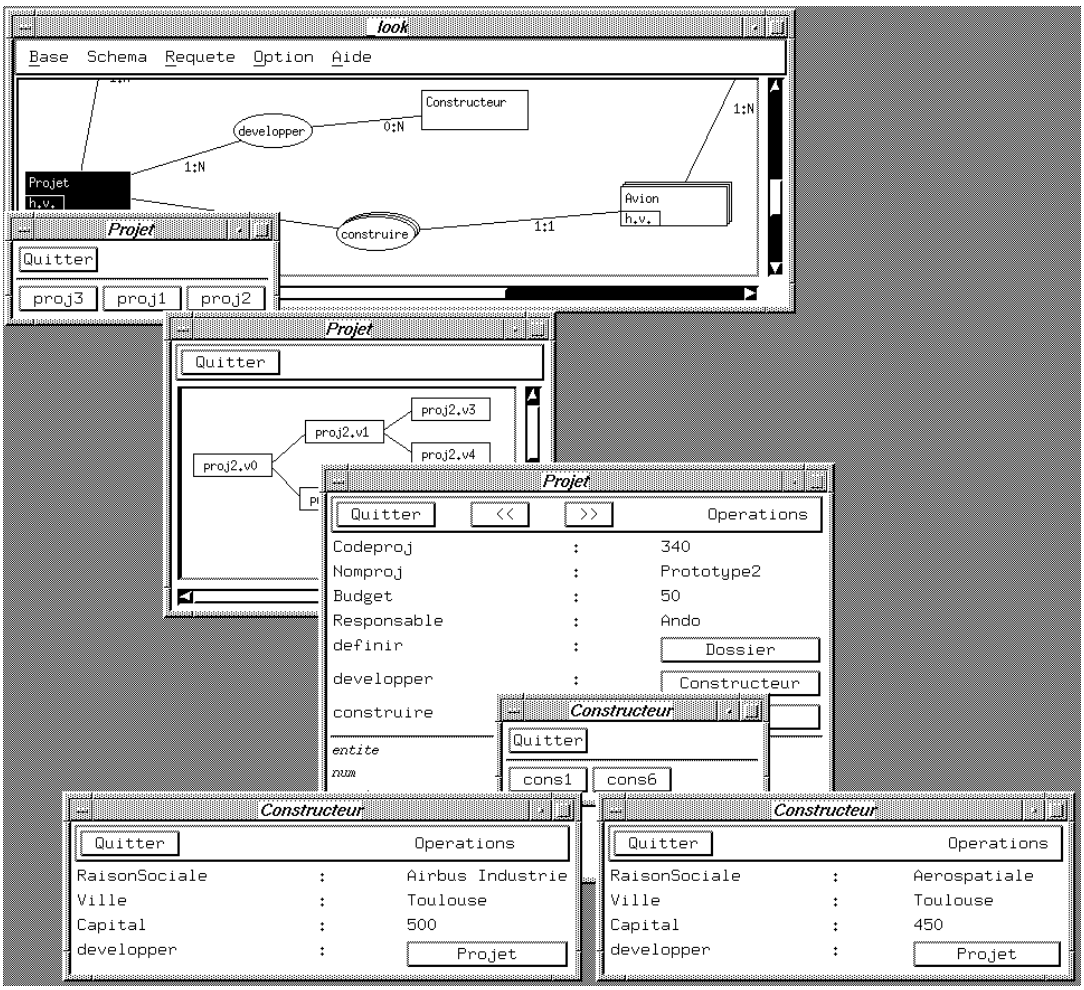
Deux modes de visualisation des instances sont disponibles :

- l'affichage sous forme d'icônes de toutes les entités représentées dans une classe ; à chaque entité correspond une icône. L'utilisateur peut ensuite obtenir la forme développée d'une entité.
- l'affichage directement sous forme développée. L'utilisateur peut passer d'une entité à une autre.

Pour les classes d'objets, l'affichage développé indique les valeurs de l'objet décrivant l'entité. Pour les classes de versions, l'affichage développé visualise la hiérarchie de dérivation de versions décrivant l'entité. Chaque version est représentée par une icône ; à partir de cet icône, on obtient la forme développée de la version correspondante c'est-à-dire ses valeurs. L'utilisateur peut ensuite visualiser les instances liées à la version affichée, en suivant les liens de composition et d'association.

Exemple IV-9 : Considérons la visualisation des instances de la classe *Projet*, selon la démarche suivante :

- affichage sous forme d'icônes de toutes les entités représentées dans la classe *Projet*,
- affichage de la hiérarchie de dérivation correspondant au projet *proj2*,
- visualisation de la forme développée de la version *proj2.v4* c'est-à-dire affichage des valeurs,
- navigation vers les instances liées de la classe *Constructeur*.



4. Bilan

Dans ce chapitre, nous avons proposé des solutions pour mettre en oeuvre et expérimenter notre modèle de données et le langage associé.

Nous avons participé au développement d'une interface graphique, nommée VOHQL (Version and Object Hypertext Query Language) pour la manipulation de bases de données intégrant des versions. Cette interface repose sur le modèle et le langage que nous avons définis. Elle est destinée à des utilisateurs occasionnels.

Le modèle, la gestion des contraintes inhérentes aux liens et le langage sont mis en oeuvre comme des modules indépendants. Ceci facilite le développement d'extensions futures.

Les solutions ne sont pas spécifiques à un système de gestion de bases de données orientées objet particulier. Elles se veulent aisément portables, puisqu'elles sont définies en utilisant les concepts communs aux SGBDOO c'est-à-dire les concepts

objet et un langage de requête. Elles sont conçues comme un ensemble de fonctionnalités permettant d'étendre un SGBDOO fournissant ces concepts, à la gestion de versions d'objets et de classes.

Notre modèle intégrant les versions est décrit à l'aide d'un méta-schéma de classes pour un modèle objet. Le langage est décrit par une grammaire non contextuelle ; celle-ci permet de spécifier les règles de traduction des requêtes exprimées dans notre langage en requêtes exprimées dans le langage du SGBD hôte, suivant le méta-schéma.

De plus, les contraintes inhérentes à notre modèle seront gérées en utilisant les concepts des bases de données actives. Nous utilisons pour cela le système Urdos réalisé au sein de notre équipe (Tchounikine, 93). Un méta-schéma de classes appelé "Schéma Actif" permet de décrire les règles et les événements sous forme d'objets. Ce schéma est mis en oeuvre à l'aide des concepts communs à tous les SGBD orienté objet ; il permet de rendre actif un SGBDOO fournissant les concepts communs.

Un prototype d'interface VOHQL est développé à l'aide du SGBD hôte O2. Il offre une représentation et une manipulation graphique des hiérarchies de dérivation de versions. Il permet la construction graphique des requêtes ; il facilite notamment l'expression des prédicats de sélection. Les requêtes graphiques construites peuvent également être exprimées dans le langage VOSQL.

Une première version est disponible sur station de travail SUN. Elle est réalisée en langages O2C et C, sous l'environnement X Window / Motif et utilise la version 4.5 multi-utilisateurs du système O2 ; elle représente plus de 16000 lignes de programmes. Le prototype permet actuellement :

- de définir textuellement et graphiquement un schéma de base incluant des versions,
- de manipuler (créer, dériver, ...) graphiquement les instances (objets et versions d'objets),
- d'interroger des classes simples : interrogation d'objets, de versions, de hiérarchies de dérivation, interrogation suivant les liens de composition et d'association, interrogation sur plusieurs classes (jointure explicite), interrogation sur les critères internes notamment la date (une partie des opérateurs disponible).

L'intégration du module actif est en cours de réalisation. La réalisation du langage VOHQL, et par conséquent celle du langage VOSQL, doit être étendue pour prendre en compte des requêtes plus complexes comme les requêtes imbriquées (requête utilisant le résultat de sous-requêtes) et les requêtes sur plusieurs versions de classes.

Les principales difficultés de réalisation rencontrées sont dues aux principes de réalisation d'une application externe au SGBD choisi. La mise en oeuvre de traitements complexes s'avère délicate lorsque l'on utilise uniquement les concepts usuels du SGBD c'est-à-dire ceux fournis à un utilisateur. L'utilisation de fonctionnalités propres au SGBD comme un langage de programmation d'applications spécifique aurait peut-être simplifié certains traitements mais aurait rendu la réalisation moins aisément adaptable à un autre SGBD.

Conclusion

Cadre de l'étude

Cette thèse s'inscrit dans le cadre des bases de données orientées objet. Les domaines d'applications liés à la gestion de documentations techniques et à la conception assistée par ordinateur ont besoin de moyens pour modéliser et manipuler les versions dans les bases de données.

Actuellement, certains systèmes offrent des possibilités pour gérer des versions lors de l'implantation d'une base. Ces systèmes permettent généralement de manipuler seulement de simples objets ; ils ne tiennent pas compte des spécificités liées aux versions.

Cette thèse présente notre contribution pour décrire des bases de données intégrant des versions ; elle fournit également des solutions pour une manipulation puissante de ce type de bases de données.

Contribution

Le premier apport de cette thèse consiste à proposer un **modèle conceptuel intégrant les versions**. Ce modèle décrit les versions au niveau élémentaire, c'est-à-dire pour chaque entité du monde réel représentée. D'autres travaux considèrent les versions à d'autres niveaux comme les versions de base entière (Gançarski, 94a) ou d'une partie de base (O2, 96) ; ce type de gestion de version, permettant de se replacer de le cadre des bases de données monoversions, évite certaines limites rencontrées dans la gestion de versions au niveau élémentaire pour les objets complexes. Cependant nous avons choisi la gestion de versions au niveau élémentaire car elle semblait bien adaptée aux besoins auxquels nous voulions répondre, notamment en termes de manipulation ; nous avons donc proposés des solutions pour combler les limites rencontrées.

Notre modèle de données permet de représenter les entités avec ou sans versions. Il combine la modélisation de l'évolution au niveau des instances via les versions d'objets, et au niveau du schéma via les versions de classes. Ce modèle offre une grande puissance de représentation d'entités complexes, telles que des documentations techniques. Pour cela, il définit des relations de composition et d'association :

- entre objets,
- entre versions d'objets,
- entre versions d'objets et objets.

La sémantique de ces relations est affinée en précisant des cardinalités. Ceci permet de décrire avec précision les entités du monde réel et leur évolution.

La gestion de versions au niveau élémentaire (versions d'objets) implique une définition complexe des contraintes d'intégrité sous-jacentes aux relations et leurs cardinalités associées. Ces contraintes sur les instances entraînent des règles suivies par les opérations pour maintenir la cohérence d'une base de données.

Les apports de notre modèle ne sont pas spécifiques au modèle objet sous-jacent. Ils peuvent également être appliqués aux modèles des méthodes de conception existantes fondées sur l'approche objet (Flory, 90), pour les étendre à la gestion des versions. Ces méthodes, telles que OMT (Rumbaugh, 91) et OOA(Coad, 91), pourraient répondre ainsi aux besoins des domaines tels que la gestion de documentations techniques et la conception assistée par ordinateur.

D'autre part, nous proposons un **langage de description et de manipulation** d'une base de données intégrant des versions. Ce langage s'appuie sur les principaux concepts des langages d'interrogation de bases d'objets ; il est en plus étendu à la manipulation des versions d'objets et des versions de classes.

Ce langage uniformise les opérations applicables aux objets et aux versions d'objets, ainsi que les opérations applicables aux classes. Il permet de définir le schéma d'une base suivant les concepts de notre modèle, de le faire évoluer à l'aide d'opérations sur les classes et de manipuler les instances (objets ou versions d'objets).

Notre langage permet principalement d'interroger de manière structurée une base intégrant des versions. Il offre une interrogation uniforme des objets et des versions d'objets.

L'interrogation est de type Select From Where ; elle prend en compte la sémantique propre aux versions d'objets, liée au principe de dérivation de versions. Le langage exploite les différents niveaux d'abstraction liés aux versions c'est-à-dire les concepts de forêt de dérivation, d'arbre de dérivation et de version d'objet. Des prédicats de sélection suivant ces niveaux d'abstraction peuvent notamment être définis.

Le langage permet également une interrogation suivant les critères "internes" propres aux versions d'objets ; il est notamment possible d'exprimer des requêtes suivant le temps, comme pour les langages applicables aux bases de données temporelles.

Il prend également en compte les spécificités liées à la gestion de versions de classes comme par exemple l'interrogation sur plusieurs versions d'une classe.

Expérimentation

Le modèle et le langage proposés dans cette thèse servent de support à une interface graphique pour bases de données intégrant des versions. Nous avons expérimenté des solutions pour la mise à oeuvre du modèle, des contraintes et du langage au sein du prototype d'interface VOHQL (Le Parc, 96a). Les contraintes inhérentes au modèle sont vérifiées en utilisant les concepts des bases de données actives à l'aide de l'outil URDOS (Tchounikine, 93a) développé dans notre équipe.

L'interface graphique VOHQL est destinée à des utilisateurs occasionnels pour faciliter la manipulation de bases de données avec versions. Elle intègre également un module langage textuel pour les utilisateurs plus confirmés. Elle offre une représentation graphique des hiérarchies de dérivation de versions d'objets ; la manipulation des versions s'effectue directement sur cette représentation. Les requêtes sont construites de manière incrémentale, directement sur un graphe représentant la base. L'interface facilite la construction des prédicats complexes de sélection sur les versions, notamment vis à vis des quantificateurs. L'étude de cette interface fait l'objet d'une future thèse au sein de notre équipe (Le Parc, 96a).

Le prototype VOHQL est développé à l'aide du système de gestion de bases de données orientées objet O2. Une première version est disponible sur station de travail SUN. Elle est réalisée en langages O2C et C, sous l'environnement X Window / Motif et utilise la version 4.5 multi-utilisateurs du système O2 ; elle représente plus de 16000 lignes de programmes.

Perspectives

Notre apport pour l'intégration des versions dans les bases de données orientées objet, peut être complété suivant deux axes principaux :

- en amont, en proposant une démarche de conception de bases de données orientées objet intégrant des versions. La démarche aboutira à la conception d'une base suivant le modèle proposé dans cette thèse. Un processus doit permettre notamment de distinguer les classes regroupant des versions d'objets par rapport aux classes d'objets et également de distinguer les classes pour lesquelles on gère des versions au niveau schéma.

  Cet apport sera complété par le développement d'un outil d'aide à la conception de bases de données intégrant des versions. Cet outil permettra d'obtenir une base de données utilisable au travers de l'interface graphique de manipulation VOHQL.

- en aval, en étendant l'interface graphique pour aider l'utilisateur occasionnel à manipuler une base de données intégrant des versions.

L'ergonomie de l'interface graphique utilisateur VOHQL doit être développée pour simplifier la manipulation de versions d'objets et de classes, et permettre une démarche d'utilisation plus naturelle. L'interface devra permettre un processus plus intuitif pour la définition de requête par un utilisateur occasionnel.

L'intégration de ces extensions au prototype développé à l'aide du SGBDO O2 aboutira à un environnement complet allant de la conception d'une base de données orientée objet intégrant la gestion de versions, jusqu'à sa manipulation.

De plus, il serait intéressant d'étendre nos travaux sur le modèle, aux apports de la gestion de contextes notamment l'approche des versions de base de données. Ces travaux proposent en effet des solutions pour simplifier la gestion des versions d'objets complexes ; la récente étude sur les mv-contraintes (Doucet, 96), offre des possibilités intéressantes pour la définition de contraintes notamment d'évolution.

Par ailleurs, les travaux sur les versions proposés dans cette thèse peuvent être complétés par l'intégration de concepts avancés de bases de données développés dans notre équipe, tels que les bases de données orientées objet réparties (Ravat, 96). Une base de données est fragmentée en plusieurs bases suivant plusieurs SGBD répartis sur plusieurs sites. Chaque SGBD local permet de manipuler, de manière transparente, l'ensemble de la base de données. La répartition s'applique également aux versions ; les versions d'objets et/ou de classes peuvent être réparties sur différents sites en fonction de leur utilisation. Le processus de répartition doit tenir compte de paramètres supplémentaires comme l'organisation des versions en hiérarchies de dérivation c'est-à-dire tenir compte des liens de dérivation entre versions d'objets et entre versions de classes. Ce principe répond notamment aux besoins de domaines comme la conception assistée par ordinateur lorsque différentes équipes travaillent sur des sites géographiques différents, chaque équipe définissant ses propres alternatives de conception à l'aide de versions.

ANNEXE I.

Notre approche et l'approche O2 : un exemple

Annexe I. : Notre approche et l'approche O2 : un exemple

Table des matières

Cette annexe décrit la définition de schéma et la manipulation des instances pour le même exemple de base, dans deux environnements différents :

- l'environnement que nous proposons qui gère les versions au niveau de chaque objet de la base,
- l'environnement O2 qui gère des versions d'une partie de base via des objets spécifiques appelés des **configurations**.

1. Gestion des versions dans notre environnement

Dans notre environnement, la gestion de versions s'applique au niveau élémentaire c'est-à-dire au niveau de chaque objet d'une base.

1.1. Création du schéma de la base

Considérons la création des classes suivante décrivant des documents ; un document est constitué d'un ensemble de chapitres, ces derniers pouvant aussi être constitués de chapitres :

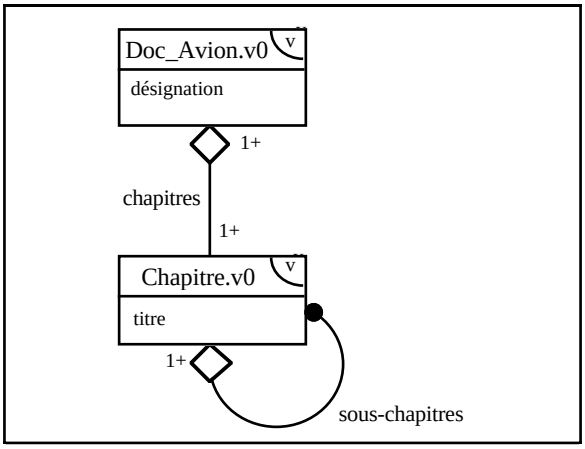
create simple class *Docu_Avion*
contains versions
attributes (*désignation* : *string***)**

create simple class *Chapitre*
contains versions
attributes (*titre* : *string*,
*texte : Text***)**

Composition chapitres : *Doc_Avion* **(**1**,**N**) of** *Chapitre* **(**1**,**N**)**

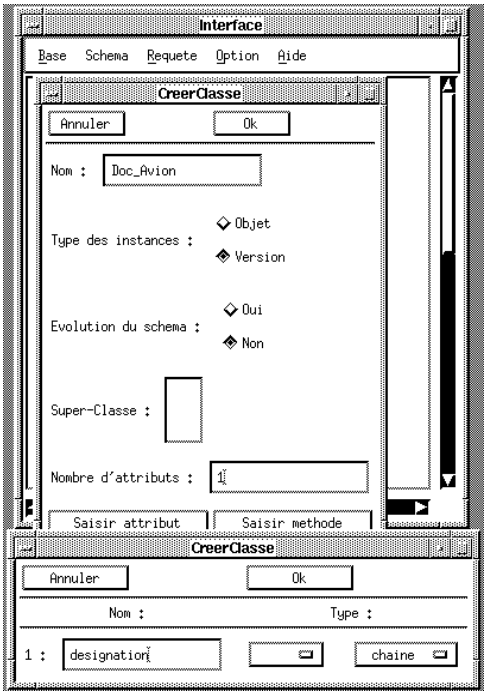
Composition sous-chapitres : *Chapitre* **(**0**,**N**) of** *Chapitre* **(**1**,**N **)**

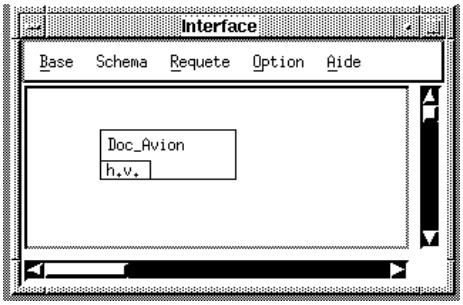
Dans le formalisme OMT étendu aux versions, la base créée est la suivante :



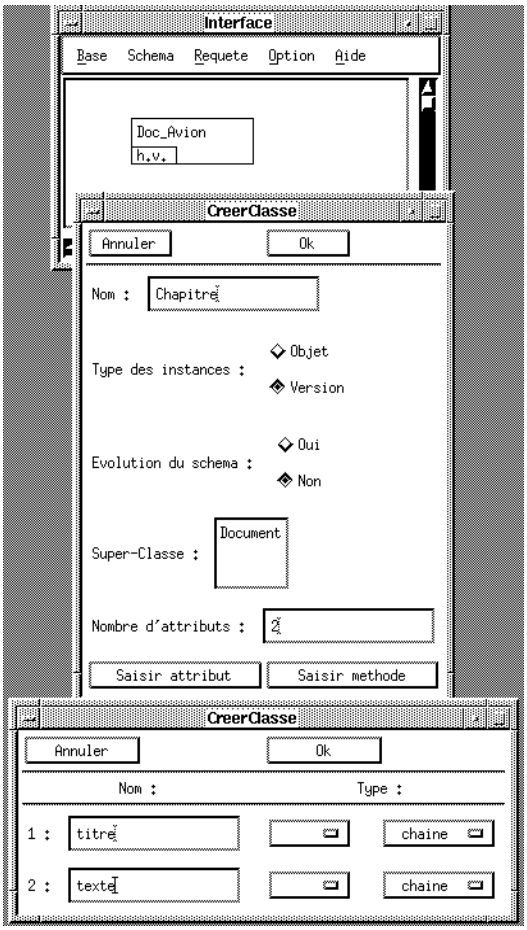
Dans l'interface VOHQL, la création de cette base s'effectue comme suit :

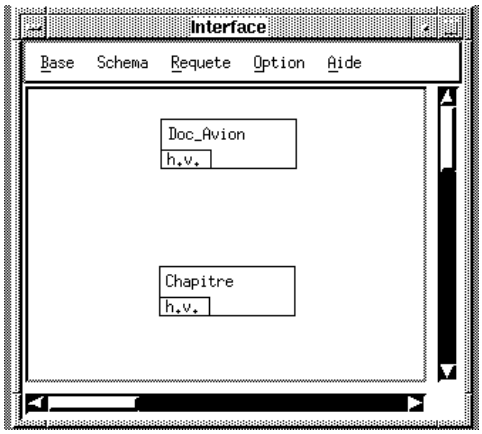
1- Création de la classe Doc_Avion



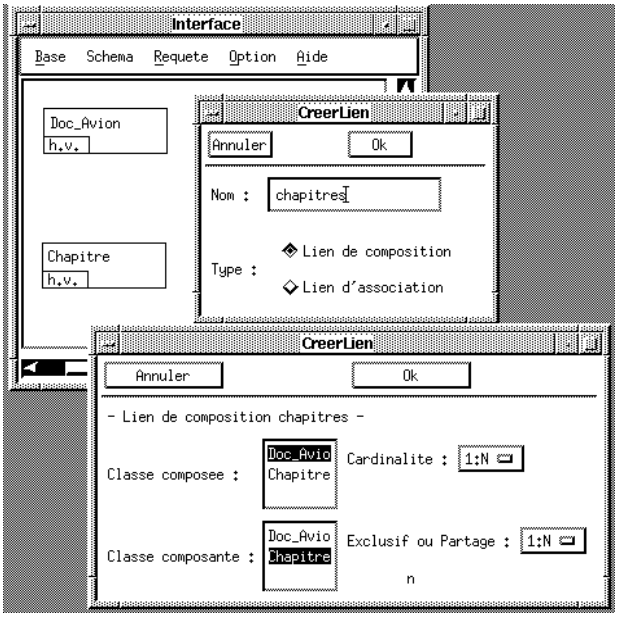


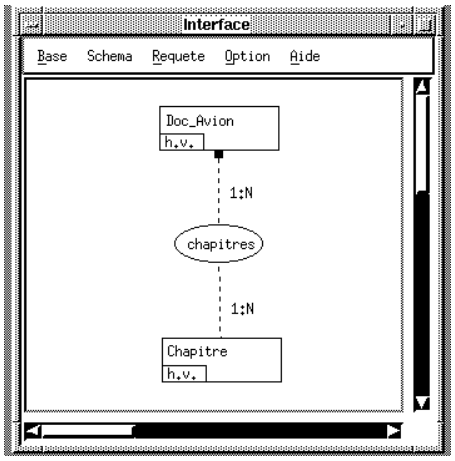
2- Création de la classe Chapitre



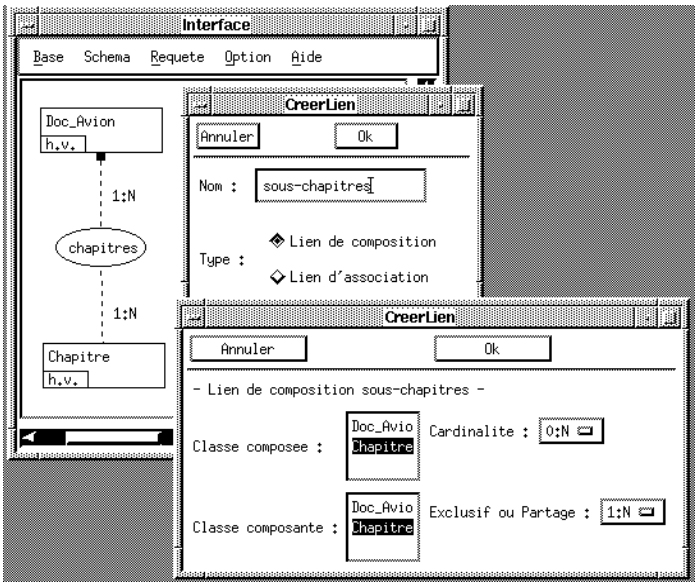


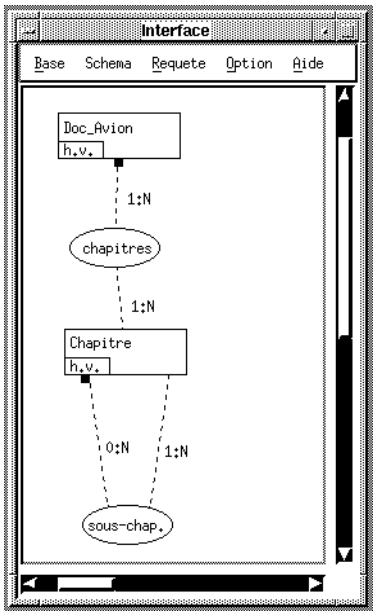
3- Création du lien de composition entre la classe Doc_Avion et la classe Chapitre





4- Création du lien de composition entre la classe Chapitre et elle-même





1.2. Création des instances

La création d'instances complexes peut s'effectuer par imbrication de déclarations ou par l'ouverture d'une transaction. Pour respecter les contraintes liées aux relations définies entre les classes, certaines créations doivent être obligatoirement effectuées dans la même transaction (cf. Chapitre II § 3.6.).

Considérons la création suivante d'une documentation :

begin transaction

-- création de la version v0 de Doc_A320 dans la classe Doc_Avion
create instance *Doc_A320* **in** *Doc_Avion*
values (*désignation* : "Documentation"**)**

-- création de la version v0 de Intro dans la classe Chapitre
create instance *Intro* **in** *Chapitre*
values (*titre* : "introduction"**,** *texte* : intro.text**)**

-- création de la version v0 de Desc dans la classe Chapitre
create instance *Desc***in** *Chapitre*
values (*titre* : "description"**,** *texte* : desc.text**)**

-- création de la version v0 de Bilan dans la classe Chapitre
create instance *Bilan* **in** *Chapitre*
values (*titre* : "bilan"**,** *texte* : bilan.text**)**

-- ajout des versions Intro.v0, Desc.v0 et Conclu.v0 comme constituants (chapitres) de Doc_A320.v0

composition instance chapitres : *Doc_A320.v0* **from** *Doc_Avion*
add *Intro.v0, Desc.v0, Bilan.v0* **from** *Chapitre*

end transaction

Pour les attributs de type texte, l'attribut reçoit le contenu du fichier indiqué. Si aucune valeur n'est spécifiée pour les attributs, la saisie des valeurs est interactive.

Compte tenu des cardinalités fixées pour les liens de composition, les créations des chapitres *Intro*, *Desc* et *Bilan* doivent être réalisées durant la même transaction que la création du document *Doc_A320*.

La cardinalité minimum de 0 pour la composition de chapitres en sous-chapitres autorise à effectuer les créations de sous-chapitres dans des transactions distinctes.

begin transaction

-- création de la version v0 de Part1 dans la classe Chapitre
create instance *Part1* **in** *Chapitre*
values (*titre* : "partie1"**,** *texte* : part1.text**)**

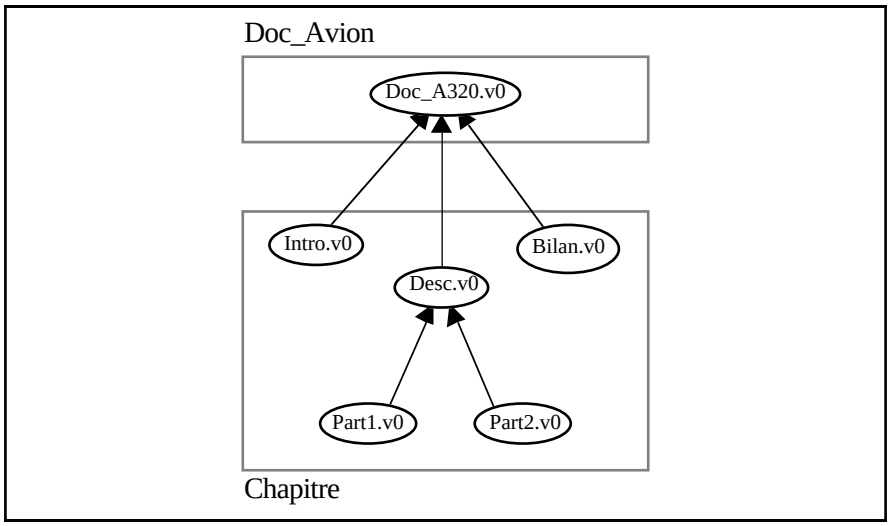
-- création de la version v0 de Part2 dans la classe Chapitre
create instance *Part2* **in** *Chapitre*
values (*titre* : "partie2"**,** *texte* : part2.text**)**

-- ajout des versions Part1.v0 et Part2.v0 comme sous-chapitres de Desc.v0
composition instance sous-chapitres : *Desc.v0* **from** *Chapitre*
add *Part1.v0, Part2.v0* **from** *Chapitre*

end transaction

Cette deuxième partie peut être intégrée dans la première déclaration.

Ces déclarations fournissent :



Lors de la dérivation de versions, les dérivations (descendantes) simultanées au sein d'une hiérarchie de composition doivent être effectuées dans la même transaction.

begin transaction

-- création de Doc_A320.v1 dérivée de Doc_A320.v0 dans la classe Doc_Avion
create instance in *Doc_Avion* **derived from** *Doc_A320.v0*

-- création de Bilan.v1, version dérivée du composant Bilan.v0 dans la classe Chapitre. Créé en même temps que la création de Doc_A320.v1, Bilan.v1 remplace Bilan.v0 comme composant. Les autres composants reste identiques.

create instance in *Chapitre* **derived from** *Bilan.v0*

end transaction

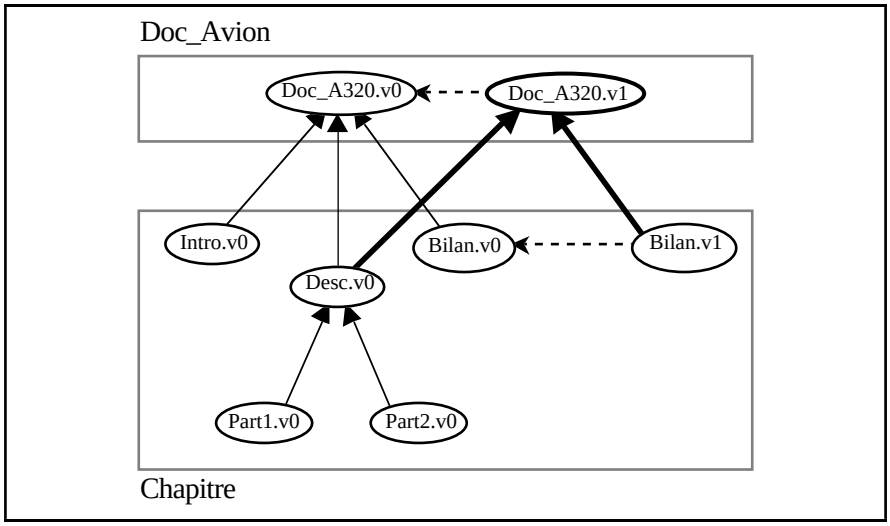
begin transaction

-- suppression de Intro.v0, implicitement conservé comme sous-chapitre lors de création de la version dérivée Doc_A320.v1

composition instance chapitres : *Doc_A320.v1* **in** *Doc_Avion*
delete *Intro.v0* **from** *Chapitre.v0*

end transaction

Cette déclaration fournit le résultat suivant :



La dérivation directe d'un composant implique la dérivation automatique des versions qu'elle compose.

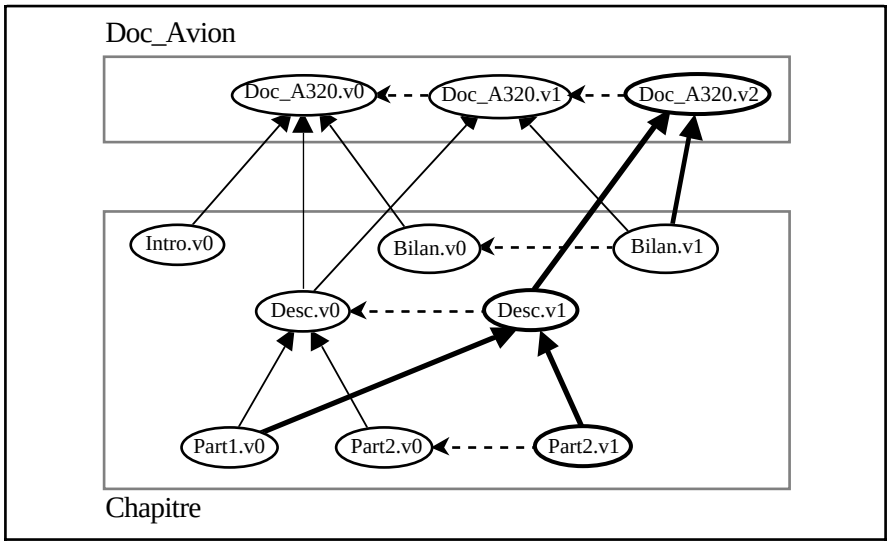
begin transaction

-- création de Part2.v1 dérivée de Part2.v1 dans la classe Doc_Avion
create instance in *Chapitre* **derived from** *Part2.v0*

end transaction

La dérivation de Part2.v0 implique les dérivations ascendantes automatiques de Desc.v0 puis Doc.v1. Les autres composants Part1.v0 et Bilan.v1 sont conservés (cf. chapitre II § 3.6.2.2.2.).

Cette déclaration fournit le résultat suivant :



1.3. Interrogation

A priori, l'interrogation permet d'obtenir des hiérarchies de versions d'objets. Chaque hiérarchie de la classe Doc_Avion décrit ici un document. Il est possible de :

1. Rechercher les hiérarchies dont toutes les versions vérifient une condition.

   Question 1 : Obtenir les documents créés à partir de 1995.

   Requête VOSQL 1 : **select** d
   from d **in** Doc_Avion
   where Forall v in **Untree**(d) : **Date**(v) >= 01/01/1995

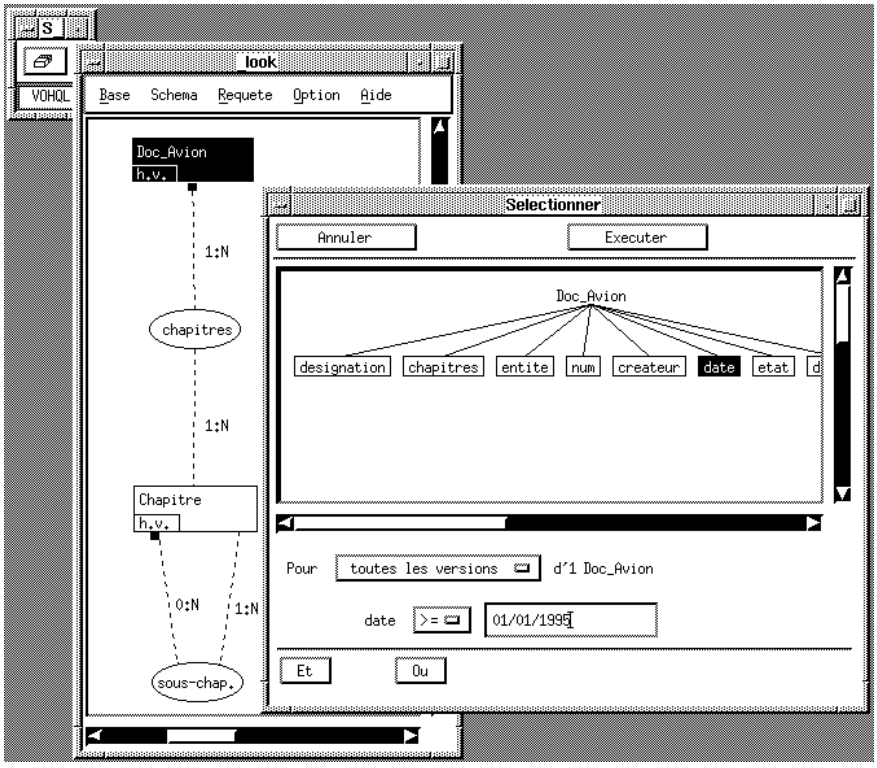
2. Rechercher les hiérarchies dont au moins une version vérifie une condition.

   Question 2 : Obtenir les documents qui existaient déjà en 1995.

   Requête VOSQL 2 : **select** d
   from d **in Doc_Avion**
   where Exist v in **Untree**(d) : **Date**(v) >= 01/01/1995
   and Date(v) <= 12/12/1995

Dans VOHQL, la requête 1, par exemple, se construit comme suit :

- l'utilisateur déclenche l'opération de sélection dans le menu associé au noeud Doc_Avion,
- l'utilisateur spécifie le prédicat de sélection en utilisant un arbre décrivant la structure des documents.



L'interrogation permet également d'obtenir des versions particulières.

3. Rechercher des versions vérifiant une condition donnée
   Question 3: Obtenir, pour tout document désigné par Doc. A320, les versions de 1995.

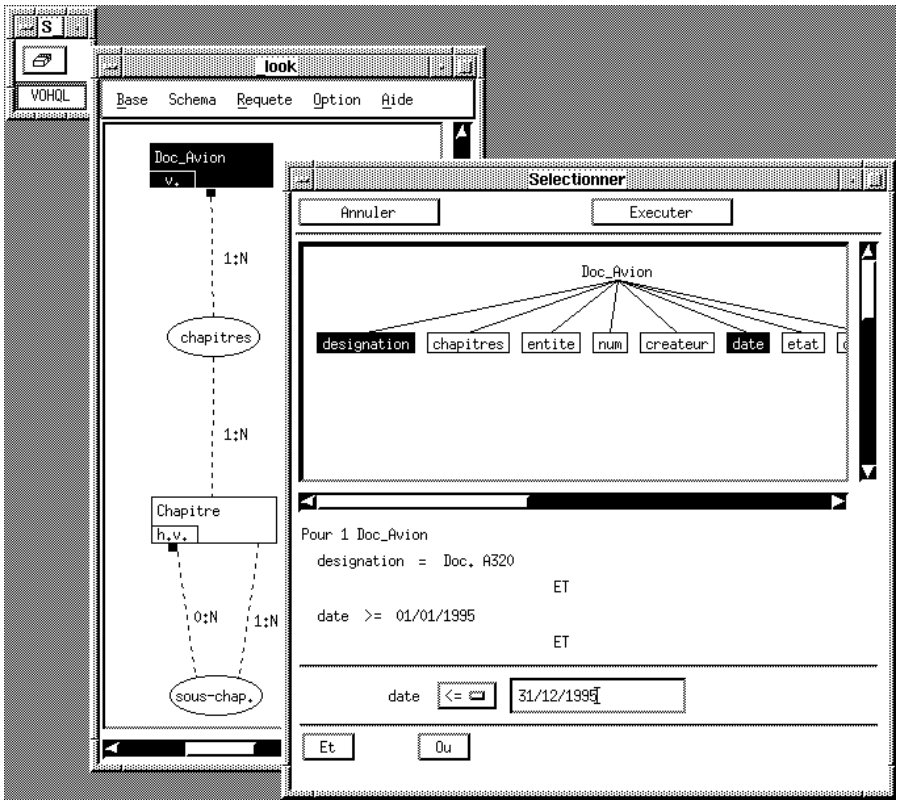
   Requête VOSQL 3 :
```
select v
from v in Untree(Unforest(Doc_Avion))
where Date(v) >= 01/01/1995 and Date(v) <= 12/12/1995
   and v->désignation = "Doc. A320"
```

4. Rechercher une version donnée pour chaque hiérarchie de dérivation.
   Question 4 : Obtenir la dernière version de chaque document.

   Requête VOSQL 4 :
```
select v
from v in Untree(Unforest(Doc_Avion))
where v->num = max( select vh->num
                    from vh in
                     Untree(Unforest(Doc_Avion))
                    where Entity(vh) = Entity(v))
```

Dans VOHQL, la requête 3, par exemple, est construite comme suit :

- l'utilisateur choisit la position **v.** au niveau du noeud Doc_Avion pour indiquer qu'il travaille sur les versions et non sur les hiérarchies (position **h.v.**, cas précédent),
- l'utilisateur spécifie le prédicat de sélection.



2. Gestion des versions dans O2

Dans le système O2 (O2, 96), la gestion de versions ne s'applique pas explicitement aux objets d'une base. Les versions sont gérées uniquement pour des objets spécifiques appelés **configurations**. Une configuration regroupe des objets de la base qui appartiennent à n'importe quelle classe. Les objets appartenant à une configuration peuvent avoir des valeurs différentes dans les différentes versions de la configuration.

2.1. Création du schéma de la base

Dans le système O2, la définition des classes décrivant les documents est réalisée indépendamment de la gestion de versions. On définit ensuite les configurations qui sont des objets pour lesquels on gère les versions ; les configurations sont définies suivants une classe pré-définie dans le système O2.

```
-- définition classique du schéma des classes indépendamment de la gestion de versions
create class C_Doc_Avion public type tuple (
        désignation : string,
        chapitres   : list ( Chapitre ))
end;

create class C_Chapitre public type tuple (
        titre : string,
        texte : Text,
        sous-chapitres   : list ( Chapitre ))
end;

-- définition des ensembles d'objets persistants associés aux classes
name Documents : set(C_Doc_Avion);
name Chapitres  : set(C_Chapitre);

-- récupération des classes pré-définies dans O2 pour la gestion de versions
import schema o2version class o2_Version, o2_list_o2_Version,
   o2_list_Object;

-- définition d'un objet persistant (type configuration) pour lequel on gérera des versions
name Config_Doc: o2_Version;
```

Cette déclaration fournit la base pour gérer des versions de documents dans O2. La version actuellement disponible ne permet pas de définir explicitement des relations de composition et d'association avec des contraintes de cardinalités. Néanmoins, des prototypes sont développés pour la gestion de contraintes via des règles actives (Collet, 96) et pour la gestion graphique de contraintes (Kvedarauskas, 96).

2.2. Création des instances

La création des instances avec gestion de versions comprend :

- la création des objets de la base (du plus élémentaire jusqu'au composé de plus haut niveau),
- la création d'une première version de configuration,
- l'ajout des objets dans la configuration.

La création d'une configuration et des objets qui y seront ajoutés, s'effectue comme suit à l'aide du langage :

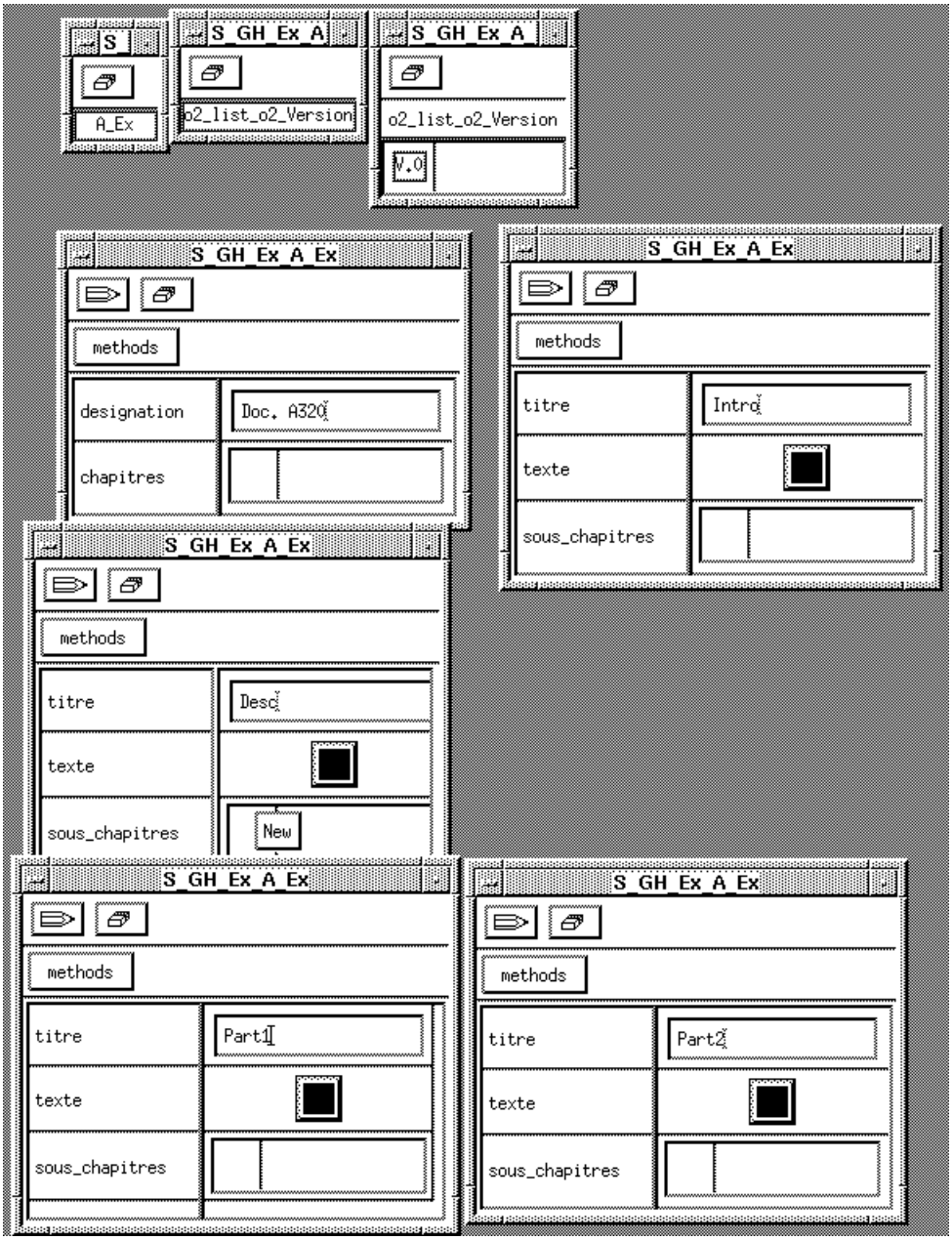
```
run body {
-- déclaration des variables associées à la création des objets
        o2 C_Doc_Avion Doc;
        o2 C_Chapitre Part1, Part2, Intro, Desc, Bilan;

-- création des objets indépendamment de la gestion de versions. Pour un objet complexe,
   création des constituants, des plus élémentaires (Part1 et Part2) jusqu'au composé de
   plus haut niveau (Doc).
        Part1 = new C_Chapitre;
        Part2 = new C_Chapitre;
        Desc = new C_Chapitre;
        -- création avec valeur d'attribut en paramètre
        Intro = new C_Chapitre ("introduction");
        Bilan = new C_Chapitre;
        Doc = new C_Chapitre;

-- ajout des objets Part1 et Part2 comme sous-chapitres de l'objet Desc
        Desc->sous-chapitres+=set(Part1, Part2);

-- ajout des objets Intro, Desc et Bilan comme chapitres de l'objet Doc
        Doc->chapitres+=set(Intro, Desc, Bilan);

-- ajout des objets créés dans les ensembles persistants correspondant (les objets sont rendus
   persistants)
        Chapitres += set(Part1, Part2, Intro, Desc, Bilan);
        Doc_Avions += set(Doc);

-- création de la première version de Config_Doc
        Config_Doc = new o2_Version ("version 1");
-- positionnement sur la première version de Config_Doc
        Config_Doc->select;

-- ajout des objets de la base dans la configuration. Pour l'objet complexe Doc, tous ses
   constituants sont également explicitement ajoutés.
        Config_Doc->append (Doc);
        Config_Doc->append (Intro);
        Config_Doc->append (Desc);
        Config_Doc->append (Bilan);
```

```
Config_Doc->append (Part1);
Config_Doc->append (Part2);
}
```

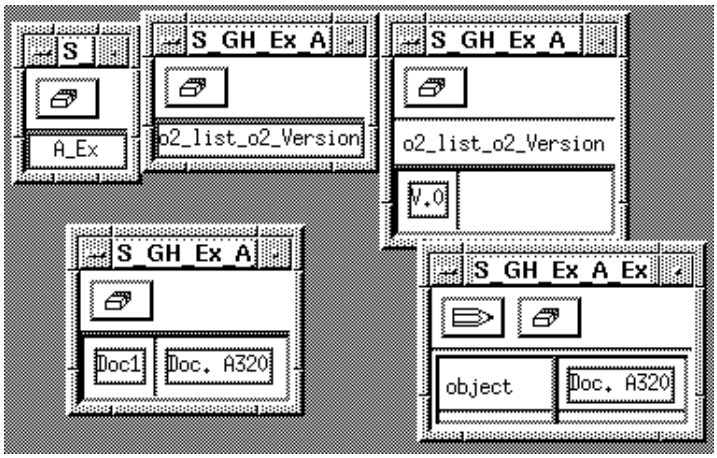
La valorisation des attributs peut s'effectuer lors de la commande *new* en mettant les valeurs en paramètres (ex. création de l'objet Intro dans l'exemple ci-dessus) ; les valeurs sont indiquées dans l'ordre. Pour les attributs de type texte, le contenu du texte peut être lu dans un fichier. Si aucune valeur n'est indiquée la saisie des valeurs est graphique (ex. création des objets Desc, Bilan, Doc, ...).

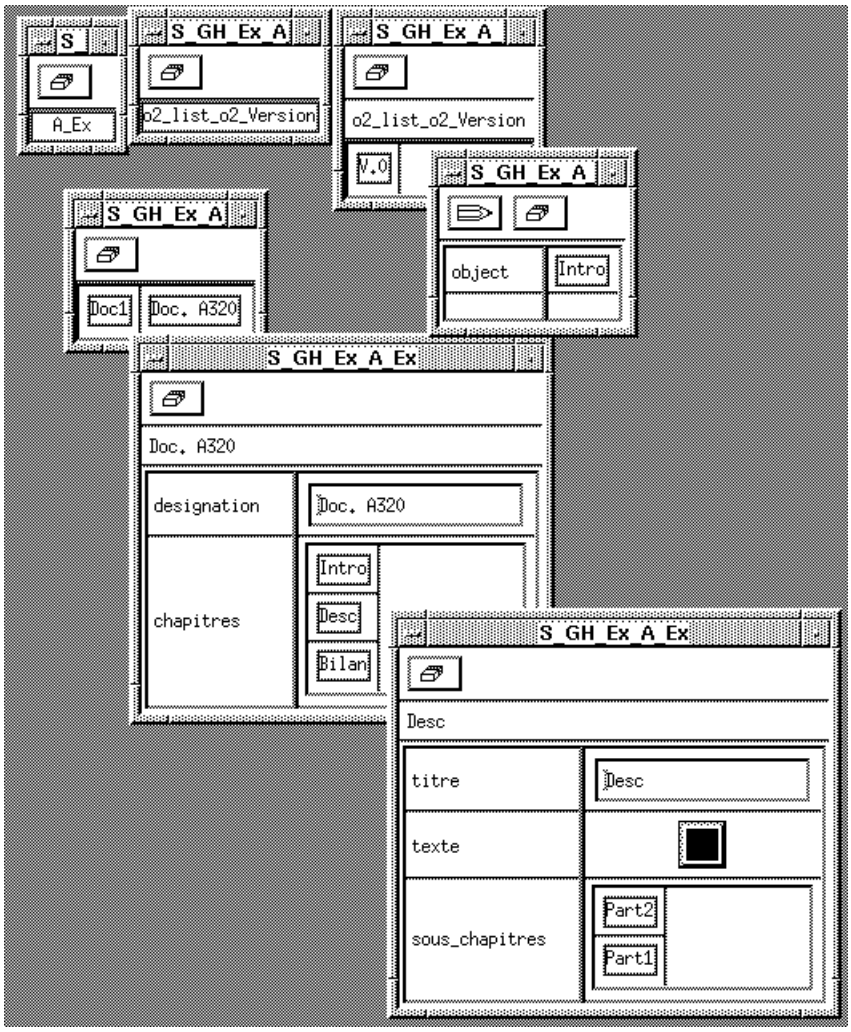
Graphiquement, la création des instances s'effectue comme suit :

- les objets composants sont créés (new) au travers des objets composites.



- les objets appartenant à la configuration sont ajoutés un par un. Pour gérer complètement des versions d'un objet complexe, il faut également ajouter ses composants à la configuration.



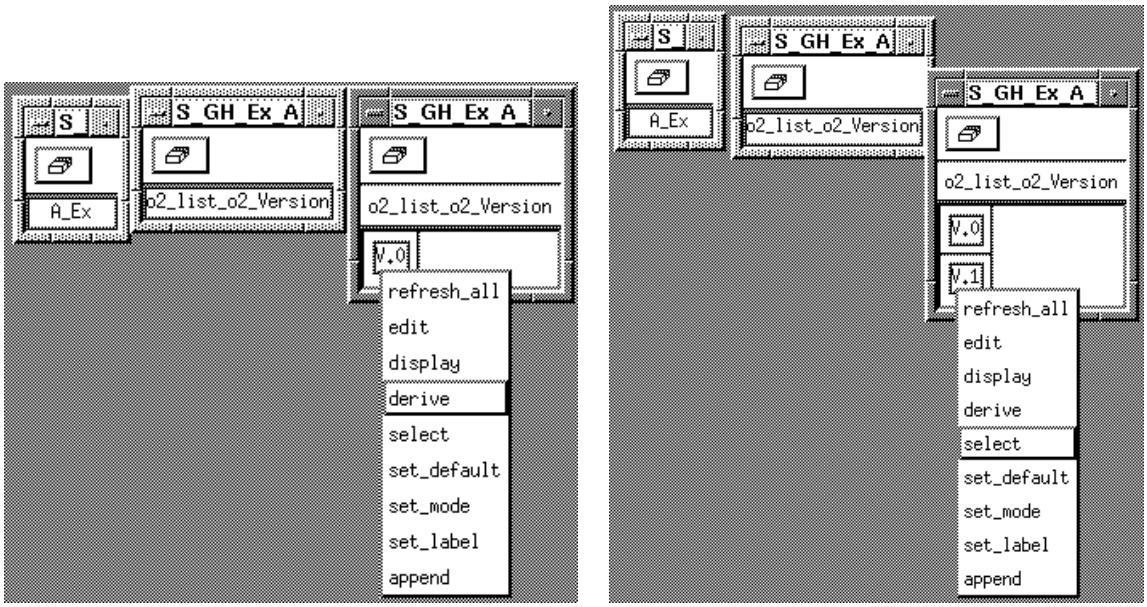


La création d'une nouvelle version de la configuration s'effectue comme suit :

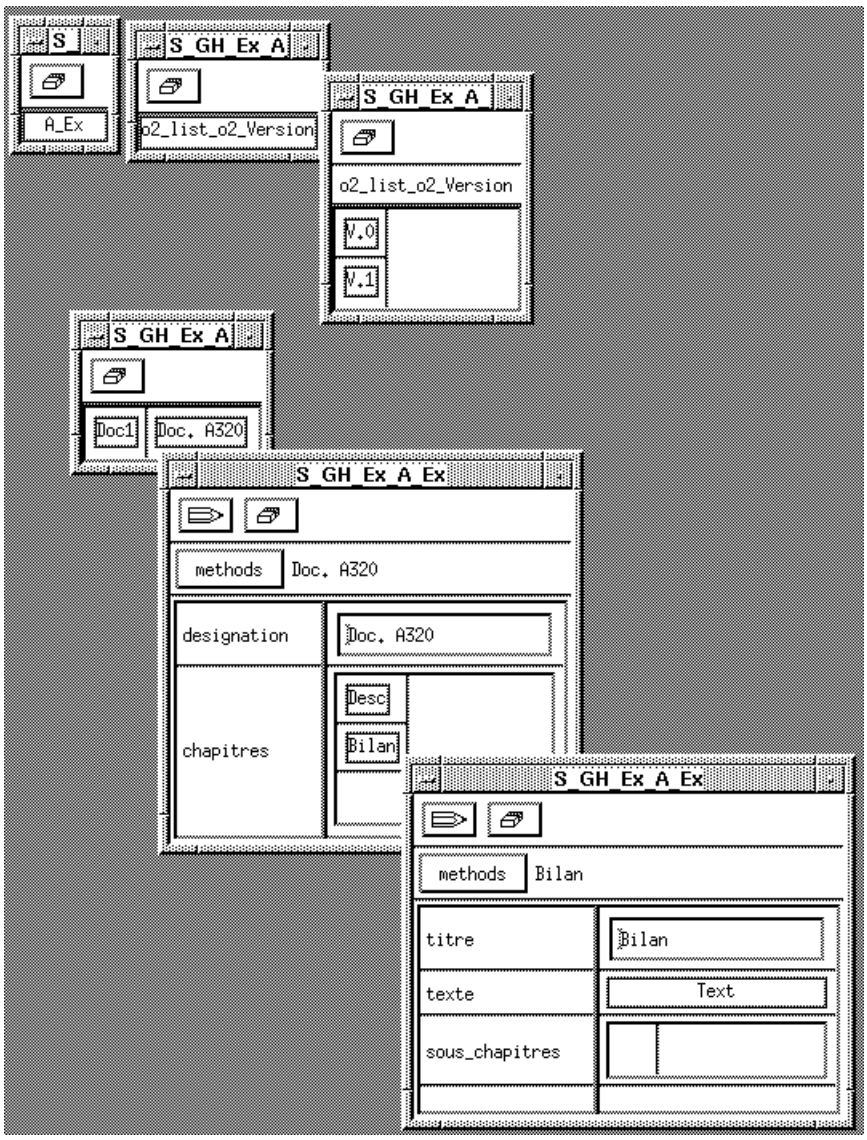
```
run body {
        o2 o2_Version v1;

-- création d'une nouvelle version de la configuration
        v1 = Config_Doc ->derive;
-- positionnement sur la nouvelle version de la configuration
        v1->select;
-- affichage des valeurs objets de la base suivant la version positionnée
        display(Doc_Avions); }
```

Les objets appartenant à la version précédente de la configuration se retrouvent dans la nouvelle version avec les mêmes valeurs.

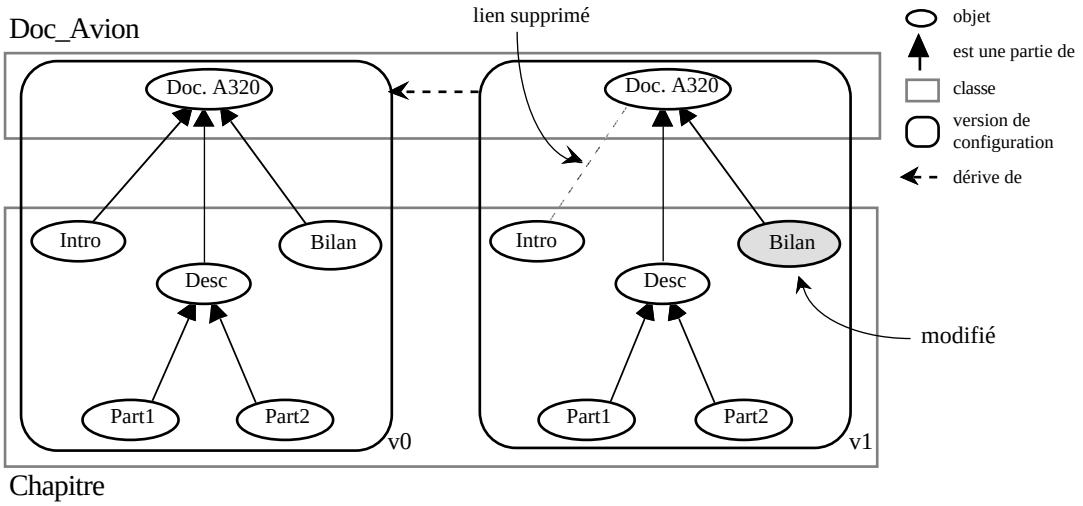
Ce processus peut être réalisée graphiquement en utilisant les icônes représentant les versions de la configuration.



Une fois la version de la configuration sélectionnée, l'utilisateur peut ensuite modifier les objets. Les modifications sont prises en compte pour la version de configuration positionnée et pour celles qui seront dérivées.



La base obtenue est alors la suivante :



Un objet peut ainsi avoir différentes valeurs dans les différentes versions d'une configuration. Les versions pour un objet sont donc gérées implicitement. Cependant, il n'est pas possible de visualiser simultanément les différentes versions d'un objet.

2.3. Interrogation

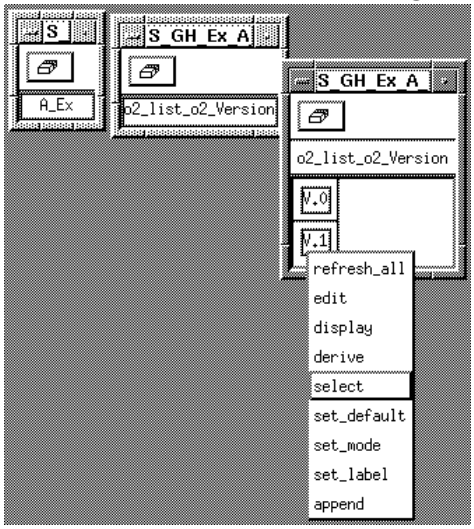
L'interrogation s'effectue toujours dans la version courante de configuration c'est-à-dire la dernière positionnée (méthode **select)**. Une requête s'applique aux objets de la base, pour les valeurs qu'ils possèdent dans la version de configuration positionnée. Il n'est pas possible d'interroger simultanément un objet dans plusieurs versions d'une configuration.

La requête 4 (cf. § 1.3.) est réalisée de la manière suivante :

Question 4 : Obtenir la dernière version pour chaque document.

Requête OQL 4b :

- se positionner sur la dernière version de la configuration,



- exécuter la requête : **select** d **from** d **in** Doc_Avions
  ou simplement le nom de l'ensemble d'objets persistant : Doc_Avions.

Les objets Doc_Avions seront affichés avec leurs valeurs dans la dernière version de configuration définie.

Compte tenu du fait que l'on ne peut obtenir simultanément un objet dans différentes versions, les requêtes 1, 2 et 3 spécifiées dans notre environnement (cf. § 1.3.) n'ont pas d'équivalent dans le système O2.

ANNEXE II.

Extrait de la grammaire LEX&YACC du langage d'interrogation

LEX

```
% %
(select) {return(select);}
(from) {return(from);}
(where) {return(where);}
(in) {return(in);}
(contains) {return(contains);}
(and) {return(and);}
(or) {return(or);}
(Forall) {return(Forall);}
(Exist) {return(Exist);}
(Default) {return(Default);}
(Frozen) {return(Frozen);}
(Date) {return(Date);}
(Creator) {return(Creator);}
(Tree) {return(Tree);}
(Untree) {return(Untree);}
(Forest) {return(Forest);}
(Unforest) {return(Unforest);}
(Set) {return(Set);}
(Flatten) {return(Flatten);}
(during) {return(during);}
(overlap) {return(overlap);}
(before) {return(before);}
(after) {return(after);}
"->" {return(fleche);}
"\"" {return(guille);}
"," {return(virgule);}
";" {return(pointvirgule);}
":" {return(deuxpoint);}
"=" {return(egal);}
"==" {return(ident);}
">" {return(sup);}
">=" {return(supegal);}
"<" {return(inf);}
"<=" {return(infegal);}
"!=" {return(diff);}
"<>" {return(diff);}
"(" {return(parent_ouv);}
")" {return(parent_fer);}
"[" {return(crochet_ouv);}
"]" {return(crochet_fer);}
[a-zA-Z0-9][a-zA-Z0-9#_]* {return(nom);}
% %
```

YACC

```
%start interrogation_vosql
%token parent_ouv parent_fer crochet_ouv crochet_fer
%token select from where in
%token fleche virgule pointvirgule deuxpoint guille
%token and or
%token Tree Untree Forest Unforest Set Flatten
%token Forall Exist
%token egal sup supegal inf infegal diff ident contains
%token during overlap after before
%token Default Frozen Date Creator
%token nom
```

```
% %
interrogation_vosql   :  req_struct
                         ;

req_struct  :   struct_op parent_ouv req_struct parent_fer
            |   req_simple
            ;

req_simple   :  select partie_select from partie_from partie_where
             ;

partie_select  :  def_select suite_select
               ;

def_select  :  nom partie_fleche
            ;

partie_fleche   :  fleche nom
                |
                ;

suite_select  :          virgule def_select
              |
              ;

partie_from   :  def_from suite_from
              ;

suite_from    :  virgule def_from
              |
              ;

def_from   :  nom in portee_req
           ;

portee_req  :  struct_op parent_ouv portee_req parent_fer
            |  nom
            |  parent_ouv interrogation_vosql parent_fer
            ;

partie_where   :  where predicat
               |
               ;

predicat  :  parent_ouv predicat parent_fer suite_predicat
          |  quantif nom in ensemb deuxpoint predicat
          |  condition suite_predicat
          ;

suite_predicat   :  op_logique predicat
                 |
                 ;

ensemb  :  nom part_flech
        |  parent_ouv interrogation_vosql parent_fer
        ;

condition   :  ensemb contains guille nom guille
            |  ensemb comp_ens ensemb
            |  guille nom guille in ensemb
            |  fonct_bool parent_ouv nom part_flech parent_fer
            |  Creator parent_ouv nom part_flech parent_fer comp nom part_flech
            |  Creator parent_ouv nom part_flech parent_fer comp guille nom guille
            |  Date parent_ouv nom part_flech parent_fer pred_temp
            ;
```

```
pred_temp  :  comp nom
           |  comp_int crochet_ouv nom virgule nom crochet_fer
           ;

part_flech  :  fleche nom part_flech
            |
            ;

quantif  :       Forall  | Exist
         ;

comp_ens  : in
          | contains
          | comp
          ;

comp  :  egal
      |  sup
      |  supegal
      |  inf
      |  infegal
      |  diff
      |  ident
      ;

comp_int  :  during
          |  overlap
          |  before
          |  after
          ;

struct_op  :  Tree
           |  Untree
           |  Forest
           |  Unforest
           |  Set
           |  Flatten
           ;

op_logique  :  and | or
            ;

fonct_bool  :  Default | Frozen
            ;
```

Références bibliographiques

(Adiba, 87) M. Adiba, N. B. Quang, C. Collet - *Aspects temporels et dynamiques dans les bases de données* - Technique et Science Informatiques, vol. 6, N°5, pp 457-478, 1987.

(Adiba, 89) M. Adiba, M. C. Fauvet - *Objets, Historiques et Versions* - Rapport Aristote - RAP003, Institut IMAG, Grenoble France, Déc. 1989.

(Agrawal, 91) R. Agrawal, S. Buroff, N. Gehani, D. Sasha - *Object Versioning in Ode* - 7th Int. Conf. on Data Engineering (ICDE'91), Kobe, Japan, pp 446-455, Apr. 1991.

(Aho, 77) A. Aho, J. Ullman - *Principles of Compiler Design* - Addison-Wesley Publishing Company, 1977.

(Allen, 84) J. F. Allen - *Towards a general theory of action and time* - Artificial Intelligence Journal, vol. 23, 1984.

(Amghar, 94) Y. Amghar, A. Flory - *Contraintes d'intégrité dans les bases de données orientées objet* - Ingénierie des Systèmes d'Information, Ed. Hermès, vol. 2, n° 5, pp 507-532, 1994.

(Andonoff, 93a) E. Andonoff, C. Morin, C. Mendiboure, V. Rougier, G. Zurfluh - *OHQL : un langage graphique pour l'interrogation d'une base de données* - Ingénierie des Systèmes d'Information, vol. 1, n° 2, pp 203-228, 1993.

(Andonoff, 93b) E. Andonoff, G. Hubert, C. Sallaberry - *CHOLQ : an object-oriented database querying environment* - 6th Int. Conf. on Software Engineering & its Applications, Paris, France, pp 203-214, Nov. 1993.

(Andonoff, 94a) E. Andonoff, G. Hubert, C. Sallaberry - *A graphic tool for complex and multimedia database querying* - 1st Int. Workshop on Information Technology (BIWIT'94), Biarritz, France, pp 195-208, Feb. 1994.

(Andonoff, 94b) E. Andonoff, G. Hubert, C. Sallaberry - *OHQL : A Database Querying Environment Intended For Naive Users* - 2nd East-West Int. Conf. on Human-Computer Interaction (EWHCI'94), St. Petersbourg, Russia, pp I-119-128, Aug. 1994.

(Andonoff, 95) E. Andonoff, G. Hubert, A. Le Parc, G. Zurfluh - *Modelling Inheritance, Composition and Relationship Links between Objects, Object Versions and Class Versions* - 7th Int. Conf. on Advanced Information Systems Engineering (CAiSE'95), Jyväskylä, Finland, Lecture Notes in Computer Science n° 932, pp 96-111, June 1995.

(Andonoff, 96) E. Andonoff, G. Hubert, A. Le Parc, G. Zurfluh - *Integrating Versions in the OMT Models* - 15th Int. Conf. on Conceptual Modeling (ER'96), Cottbus, Germany, LNCS n° 1157, pp 472-487, Oct. 1996.

(Atkinson, 89) M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier et S. Zdonik - *The object-oriented database manifesto* - 1st Int. Conf. on Deductive and Object Oriented Database (DOOD'89), Kyoto, Japan, pp 40-57, 1989.

(Atwood, 86) T.M. Atwood - *An object-oriented DBMS for design support applications* - Int. Conf. COMPINT, Montréal, Canada, pp 299-307, 1986.

(Atwood, 94) T. M. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade - *The Object Database Standard, ODMG-93* - Ed. by R. Cattel, Morgan Kaufmann Publishers, 1994.

(Bancilhon, 92) F. Bancilhon, C. Delobel, P. Kanellakis - *Building an Object Oriented Database System : The Story of O2* - Morgan Kaufmann Publishers, San Mateo, California, 1992.

(Banerjee, 87) J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth - *Semantics and Implementation of schema evolution in object oriented databases* - SIGMOD Record, vol. 16, N°3, pp 311-322, Dec. 1987.

(Beech, 88) D. Beech, B. Mahbod - *Generalized Version Control in an Object Oriented Database* - 4th Int. Conf. on Data Engineering (ICDE'88), Los Angeles, California, pp 14-22, 1988.

(Ben Amouzegh, 86) C. Ben Amouzegh, C. Chrisment, G. Zurfluh - *Bases d'Informations Généralisées - Concept de Version* - 2èmes Journées de Bases de Données Avancées (BDA'86), Giens, France, pp 5-19, 1986.

(Ben-Zvi, 82) J. Ben-Zvi - *The Time Relational Model* - PhD Thesis, Computer Science Department, UCLA, 1982.

(Bensadoun, 89) O. Bensadoun, C. Chrisment, G. Pujolle, G. Zurfluh - *Aspects dynamiques dans les bases de documents* - 5èmes Journées de Bases de Données Avancées (BDA'89), Genève, Suisse, pp 291-308, 1989.

(Bertino, 92) E. Bertino - *A View Mechanism for Object-Oriented Databases* - 3rd Int. Conf. on Extending Database Technology (EDBT'92), Vienna, Austria, LNCS N° 580, pp 136-151, March 1992.

(Björnerstedt, 89) A. Björnerstedt, C. Hultén - *Version control in an object oriented architecture* - Object oriented Concepts, Applications and Databases, Ed. W. Kim and F. Lochovsky, Addison-Wesley, pp 451-486, 1989.

(Bouaziz, 95) T. Bouaziz - *Classification des contraintes d'intégrité dans les bases de données orientées objet* - Ingénierie des Systèmes d'Information, Ed. Hermès, vol. 3, n° 6, pp 713-737, 1995.

(Bounaas, 95) F. Bounaas - *Gestion de l'évolution dans les bases de connaissances : une approche par les règles* - Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1995.

(Canillac, 91) M. Canillac - *Un modèle et une interface hypertexte pour les bases de données orientées objet* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, Juillet 1991.

(Cellary, 90) W. Cellary, G. Jomier - *Consistency of Versions in Object Oriented Databases* - 16th Int. Conf. on Very Large Data Bases (VLDB'90), Brisbane, Australia, pp 432-441, 1990.

(Cellary, 91) W. Cellary, G. Jomier, T. Koszlajda - *Formal Model of an Object-Oriented Database with Versioned Objects and Schema* - 2nd Int. Conf. on Database and Expert Systems Applications (DEXA'91), Berlin, Germany, pp 237-244, 1991.

(Chakravarthy, 93) S. Chakravarthy, S.-K. Kim - *Resolution of Time Concepts in Temporal Databases* - Technical Report UF-CIS-TR-93-004, Department of Computer and Information Sciences, University of Florida, Gainesville, Jan. 1993.

(Chou, 86) H.-T. Chou, W. Kim - *A Unifying framework for version control in CAD environment* - 12th Int. Conf. on Very Large Data Bases (VLDB'86), Kyoto, Japan, pp 336-344, 1986.

(Chou, 88) H.-T. Chou, W. Kim - *Versions and Change notification in an object oriented database system* - 25 th ACM/IEEE Design Automation Conf., Anaheim, California, pp 275-281, June 1988.

(Chrisment, 91) C. Chrisment, G. Zurfluh - *Stockage de documents : Base de données, version de documents et stockage physique* - Service du Traitement de l'Information - Juin 1991.

(Clamen, 92) S. M. Clamen - *Type Evolution and Instance Adaptation* - Tehnical report N° CMU-CS-92-133, School of Computer Science, Carnegie Mellon University Pittsburgh, PA 15213-3890, 1992.

(Clamen, 94) S. M. Clamen - *Schema Evolution and Integration* - Distributed and Parallel Databases Journal, 2, Kluwer Academic Publishers, pp 101-126, 1994.

(Coad, 91) P. Coad, E. Yourdon - *Object-Oriented Design*- Prentice-Hall, 1991.

(Comparot, 94) C. Comparot-Poussier - *Hyperbase : formalisation et architecture* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1994.

(Collet, 96) C. Collet, T. Coupaye, C. Roncansio - *NAOS : Native Active Object-oriented database System* - Journée O2 Technology, XIV^{e} Congrès INFORSID, Bordeaux, France, Juin 1996.

(Date, 82) C. J. Date - *An introduction to database systems* - Addison Wesley Publishing Company, 3rd edition, 1982.

(Dattolo, 95) A. Dattolo, V. Loia - *Hypertext Version Management in an Actor-based Framework* - 7th Int. Conf. on Advanced Information Systems Engineering (CAiSE'95), Jyväskylä, Finland, LNCS n° 932, pp 112-125, June 1995.

(Dayal, 88) U. Dayal, A. Buchman, D. McCarthy - *Rules are object too : a knowledge model for an active OODS* - Advances in OODS, Sept. 1988.

(Diaz, 91) O. Diaz, N. Patton, P. Gray - *Rule management in OODB : A uniform approach* - 17th Int. Conf. on Very Large Data Bases (VLDB'91), Barcelona, Spain, Sept. 1991.

(Dijkstra, 93) J. Dijkstra - *On Complex objects and versioning in complex environments* - 12th Int. Conf. on the Entity-Relationship Approach (ER'93), Arlington, Texas, LNCS N° 823, pp 13-23, Dec. 1993.

(Dittrich, 88) K. R. Dittrich, R. A. Lorie - *Version support for engineering database systems* - Transactions on Software Engineering, vol. 14, N° 4, pp 429-437, Apr. 1988.

(Dittrich, 90) K. R. Dittrich, W. Gotthard, P. C. Lockermann - *Complex Entities for Engineering Applications* - Research Foundation in OO and Semantic DBMS, Prentice Hall, Ed. by A. Cardenas, D. McLeod, pp 303-321, 1990.

(Djeraba, 93) C. Djeraba - *Quelques liens sémantiques dans un Système à Base de Connaissances* - Thèse de doctorat, Université C. Bernard - Lyon I, Lyon, France, 1993.

(Doucet, 96) A. Doucet, S. Gançarski, G. Jomier, S. Monties - *Maintien de la cohérence dans une base de données multiversion* - 12èmes Journées de Bases de Données Avancées (BDA'96), Cassis, France, pp 181-201, 1996.

(Dubac, 95) C. Dubac - *Adaptation de l'interface OHQL à un contexte industriel : manipulation de données par des utilisateurs non-informaticiens* - Rapport de DEA, IRIT, Toulouse, France, 1995.

(Ecklund, 87) D. J. Ecklund, E. F. Ecklund, R. O. Eifrig, F. M. Tonge - *DVSS : A distributed Version Storage Server for CAD Applications* - 13th Int. Conf. on Very Large Data Bases (VLDB'87), Brighton, UK, 1987.

(Edelweiss, 93) N. Edelweiss, J. Palazzo M. de Oliveira, B. Pernici - *An Object-Oriented Approach to a Temporal Query Language* - 5th Int. Conf. on Database and Expert Systems Applications (DEXA'94), Athens, Greece, LNCS N° 856, pp 225-235, 1994.

(Escamilla, 90) J. Escamilla, V. Favier, P. Jean, G. T. Nguyen, D. Rieu - *Représentation de connaissances dynamiques dans SHERPA* - VIII^{e} Congrès INFORSID, Biarritz, France, Tome 1, Ed. Eyrolles, Mai 1990.

(Estublier, 94) J. Estublier, R. Casallas - *The Adele Configuration Manager* - Configuration Management, ed. by W. Tichy, Software Trend Serie, 1994.

(Fauvet, 92) M. C. Fauvet - *Versions and Histories in Object-Oriented Applications* - 7th Int. Brazilian Databases Conf., Porto Alegre, Brazil, May 1992.

(Flory, 90) A. Flory, C. Rolland - *Conception des systèmes d'informations : Etat de l'art et nouvelles perspectives* - Nouvelles perspectives des Systèmes d'Informations, éd. Eyrolles, pp 3-40, 1990.

(Gadia, 92) S. K. Gadia - *A Seamless Generic Extension of SQL for Querying Temporal Data* - Technical Report, TR-92-02, Computer Science Department, Iowa State University, March 1992.

(Gançarski, 94a) S. Gançarski - *Versions et Bases de Données : modèle formel, supports de langage et d'interface-utilisateur* - Thèse de doctorat, Université Paris XI Orsay, Paris, France, 1994.

(Gançarski, 94b) S. Gançarski, G. Jomier - *Managing Entity Versions within their Contexts : A Formal Approach* - 5th Int. Conf. on Database and Expert Systems Applications (DEXA'94), Athens, Greece, LNCS N° 856, pp 400-409, Sept. 1994.

(Giraudin, 95) J. P. Giraudin - *Evaluation de méthodologies orientées objet* - Cours du XIIIème Congrès INFORSID, Grenoble, France, Mai 1995.

(Guetari, 96) R. Guetari, T. Nguyen - *A computer-aided specification tool for concurrent engineering design* - 3th Int. Conf. on Concurrent Engineering, Toronto, Canada, pp 121-127, 1996.

(Härder, 87) T. Härder, K. Meyer-Weneger, B. Mitschang, A. Sikeler - *PRIMA : A DBMS Prototype Supporting Engineering Applications* - 13th Int. Conf. on Very Large Data Bases (VLDB'87), Brighton, UK, pp 433-442, 1987.

(Hubert, 94) G. Hubert, F. Ravat - *Interrogation graphique de bases de données orientées objet réparties* - Conf. Représentation par Objet (RPO'94), 14èmes Journées internationales IA'94, Paris, France, pp 115-124, Mai 1994.

(Hubert, 95a) G. Hubert - *Objets, versions d'entités et versions de classes : Etude des liens d'héritage, de composition et d'association* - Actes de séminaire : 1ère Journée des Jeunes Chercheurs en Modélisation et Environnements de développement, GDR-PRC BD, Grenoble, France, pp 103-120, Jan. 1995.

(Hubert, 95b) G. Hubert, A. Le Parc - *Etude de la composition dans un modèle objet intégrant des versions d'entités et des versions de classes* - XIIIème Congrès INFORSID, Grenoble, France, pp 277-296, Mai 1995.

(Jensen, 94) C. S. Jensen, J. Clifford, S. K. Gadia - *A Glossary of Temporal Database Concepts* - SIGMOD Record, vol. 23, N°1, March 1994.

(Käfer, 92) W. Käfer, H. Schöning - *Mapping a Version Model to a Complex-Object Data Model* - 8th Int. Conf. on Data Engineering (ICDE'92), Tempe, Arizona, pp 348-357, Feb. 1992.

(Katz, 86) R. H. Katz, E. Chang, R. Batheja - *Version Modelling concepts for computer aided design databases* - Int. Conf. ACM SIGMOD, Washington, D. C., pp 379-386, May 1986.

(Katz, 90a) R. H. Katz - *Toward a unified framework for version modelling in engineering databases* - ACM Computing Surveys, vol 22, N° 4, pp 375-408, 1990.

(Katz, 90b) R. H. Katz, E. Chang - *Managing Change in Computer-Aided Design Databases* - Research Foundation in OO and Semantic DBMS, Prentice Hall, Edited by A. Cardenas, D. McLeod, pp 267-282, 1990.

(Katz, 90c) R. H. Katz, T. F. Chiueh - *Papyrus : A Structured History Database for VLSI Design Flow Management* - Technical report, University of Berkeley, USA, Sept. 1990.

(Khosafian, 86) S.N. Khoshafian, G. P. Copeland - *Object identity* - 1st Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, pp 406-416, 1986.

(Kim, 87) W. Kim, J. Banerjee, H.-T. Chou, J.Garza, D. Woelk - *Composite Object Support in an Object-Oriented Database System* - 2nd Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, pp 118-125, Oct. 1987.

(Kim, 88) W. Kim, H.-T. Chou - *Versions of Schema For Object-Oriented Databases* - 14th Int. Conf. on Very Large Data Bases (VLDB'88), Los Angeles, California, Aug. 1988.

(Kim, 89a) W. Kim, E. Bertino, J. F. Garza - *Composite Object Revisited* - , SIGMOD Record, vol. 18, N° 2, pp 337-347, June 1989.

(Kim, 89b) W. Kim, N. Ballou, H.-T. Chou, J. F. Garza, D. Woelk - *Features of the ORION Object-Oriented Database* - Object oriented Concepts, Applications and Databases, Ed. W. Kim and F. Lochovsky, Addison-Wesley, pp 251-282, 1989.

(Kimball, 91) J. Kimball, L. Aaron - *Epochs, configuration schema and version cursors in the KBSA framework CCM model* - 3rd Int. Workshop on Software Configuration Management, Trondheim, Norway, pp 33-42, 1991.

(Kline, 93) N. Kline - *An Update of the Temporal Database Bibliography* - SIGMOD Record, vol. 22, n° 4, Dec. 1993.

(Kvedarauskas, 96) D. Kvedarauskas, P.-Y. Policella - *Visual Constraint* - Journée O2 Technology, XIV^{e} Congrès INFORSID, Bordeaux, France, Juin 1996.

(Landis, 86) G. S. Landis - *Design Evolution and History in an object oriented CAD/CAM database* - IEEE COMPCON, San Francisco, California, pp 297-303, 1986.

(Le Parc, 96a) A. Le Parc - *VOHQL : une interface d'interrogation de bases de données intégrant le concept de version* - Actes de séminaire : 2ème Journée des Jeunes Chercheurs en Modélisation et Environnements de développement, GDR-PRC BD, Paris, France, pp 55-74, Janv. 1996.

(Le Parc, 96b) A. Le Parc - *Un modèle et une algèbre pour bases de données orientées objet intégrant la gestion de versions* - Rapport interne, IRIT, 96-26-R, Toulouse, France, pp 1-52, Sept. 1996.

(Lerner, 90) B. Lerner, A. Habermann - *Beyond Schema evolution to Database Reorganisation* - SIGPLAN Notices, 25(10), pp 67-76.

(Lerner, 94) Barbara Staudt Lerner - *Type Evolution Support for Complex Type Changes* - Technical Report 94-71, Univ. of Massachusetts, Oct. 1994.

(McKenzie, 91) E. McKenzie, R. T. Snodgrass - *Supporting Valid Time in an Historical Relational Algebra : Proofs ansd Extensions* - Technical Report TR-91-15, Department of Computer Science, University of Arizona, 1991.

(Miller, 89) D. B. Miller, Stockton, C. W. Krueger - *An inverted approach to configuration management* - 2nd Int. Workshop on Software Configuration Management, Princeton, New Jersey, pp 1-4, 1989.

(Monk, 92) S. R. Monk, I. Sommerville - *A Model for Versioning of classes in object oriented databases* - 10th Brit. Nat. Conf. on Databases (BNCOD'92), Aberdeen, UK, pp 42-58, 1992.

(Monk, 93) S. Monk, I. Sommerville - *Schema Evolution in OODBs Using Class Versioning* - SIGMOD Record, vol. 22, N° 3, pp 16-22, Sept. 1993.

(Navathe, 89) S. B. Navathe, R. Ahmed - *A temporal relational model and a query language* - Information Sciences, vol. 49, n° 2, pp 147-175, 1989.

(Nguyen, 89) G. .T. Nguyen, D. Rieu - *Schema evolution in object oriented database systems* - Data & Knowledge Engineering, North Holland, vol. 4, N°1, pp 43-67, Juil. 1989.

(Nguyen, 93) G. T. Nguyen - *SHOOD : plate-forme pour la conception assistée* - Ingénierie des Systèmes d'Information, Ed. Hermès, vol. 1, n° 3, 1993.

(O2, 96) O2 - *The O2 user's manual, version 4.6* - O2Technology, Versailles, 1996.

(Palisser 89) C. Palisser - *Charly, un gestionnaire de versions pour la CAO en architecture* - Thèse de doctorat, Université d'Aix-Marseille III, Marseille, France, 1989.

(Palisser, 90) C. Palisser - *Le Modèle de Versions du Système CHARLY* - 6èmes Journées de Bases de Données Avancées (BDA'90), Montpellier, France, pp 213-230, 1990.

(Penney, 87) D. J. Penney, J. Stein - *Class Modification in the GemStone object oriented Database System* - 2nd Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, pp 111-117, Oct. 1987.

(Pissinou, 92) N. Pissinou, K. Makki - *A Framework for Temporal Object Databases* - 1st Int. Conf. on Information and Knowledge Management (CIKM'92), Baltimore, Maryland, LNCS n° 752, pp 86-97, Nov. 1992.

(Ravat, 96) F. Ravat - *OD^3 : contribution méthodologique à la conception de bases de données orientées objet réparties* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, Sept. 1996.

(Reichenberg, 89) C. Reichenberg - *Orthogonal version management* - 2nd Int. Workshop on Software Configuration Management, Princeton, New Jersey, pp 137-140, 1989.

(Rieu, 85) D. Rieu - *Modèle et fonctionnalités d'un SGBD pour les applications CAO* - Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1985.

(Rieu, 86) D. Rieu - *Nature, Etat et Dynamicité de l'objet CAO* - 2èmes Journées de Bases de données Avancées (BDA'86), Giens, France, pp 21-39, 1986.

(Rieu, 91) D. Rieu, G. T. Nguyen, A. Culet, J. Escamilla, C. Djeraba - *Instanciation multiple et classification d'objets* - 7èmes Journées de Bases de données Avancées (BDA'91), Lyon, France, 1991.

(Rochfeld 92) A. Rochfeld - *Les méthodes de conception orientées objet* - Conférence invitée, Xe Congrès INFORSID, Clermont-Ferrand, France, Mai 1992.

(Rochkind, 75) M. J. Rochkind - *The Source Code Control System* - IEEE Transactions on Software Engineering, vol. 1, n° 4, pp 364-370, Dec. 1975.

(Roddick, 93) J. F. Roddick, N. G. Craske, T. J. Richards - *A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models* - 12th Int. Conf. on the Entity-Relationship Approach (ER'93), Arlington, Texas, LNCS N° 823, pp 137-148, 1993.

(Rose, 91) E. Rose, A. Segev - *TOODM - A Temporal Object Oriented Data Model with Temporal Constraints* - 10th Int. Conf. on the Entity-Relationship Approach (ER'91), San Mateo, California, pp 205-229, 1991.

(Rumbaugh 91) J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen - *Object-oriented modelling and design* - Prentice-Hall publishing company, Englewood Cliffs, 1991.

(Sallaberry, 92) C. Sallaberry - *CHOLQ : une interface de manipulation de bases de données orientées objet pour non-spécialistes : mise en oeuvre dans le cadre d'une application industrielle* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1992.

(Sarda, 90) N. L. Sarda - *Extensions to SQL for historical databases* - IEEE Transactions on Knowledge and Data Engineering , vol. 2, n° 2, pp 220-230, July 1990.

(Scherrer, 93) S. Scherrer, A. Geppert, K. R. Dittrich - *Schema Evolution in NO2* - Technical Report Nr. 93.12, Institut für Informatik der Universtät Zürich, April 1993.

(Scheck, 82) J. Scheck, P. Pistor - *Data structures for an integrated database management and information retrieval system* - 8th Int. Conf. on Very Large Data Bases (VLDB'82), Mexico, Mexico, Sept. 1982.

(Sciore, 91) E. Sciore - *Using Annotations to Support Multiple Kinds of Versioning in an Object Oriented Database System* - ACM Transactions on Database Systems, vol. 16, N° 3, pp 417-438, Sept. 1991.

(Skarra, 86) A. H. Skarra, S. B. Zdonik - *The management of changing types in an object oriented database* - 1st Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, 1986.

(Snodgrass, 87) R. Snodgrass - *The Temporal Query Language TQuel* - ACM Transactions on Database Systems, vol. 12, n° 2, pp 247-298, June, 1987.

(Snodgrass, 94) R. Snodgrass et al - *TSQL2 Language Specification* - SIGMOD Record, vol. 23, N°1, March 1994.

(Souza, 93) C. Souza dos Santos, S. Abiteboul, C. Delobel - *Virtual Schemas and Bases* - 9èmes Jounées de Bases de Données Avancées (BDA'93), Toulouse, France, Sept. 1993.

(Talens, 93) G. Talens, C. Oussalah, M. F. Colinas - *Versions of Simple and Composite Objects* - 19th Int. Conf. on Very Large Data Bases (VLDB'93), Dublin, Ireland, pp 62-72, 1993.

(Talens, 94) G. Talens-Barthalay - *Gestion de versions d'objets simples et composites* - Thèse de doctorat, Université Montpellier II, Montpellier, France, 1994.

(Tan, 89) L. Tan, T. Katayama - *Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution* - Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan, pp 241-258, 1989.

(Tansel, 93) A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass - *Temporal Databases : Theory, Design and Implementation* - Database Systems and Applications, Benjamin/Cummings, 1993.

(Tchounikine, 93a) A. Tchounikine - *Activité dans les Bases de Données Objets : Le concept de Schéma Actif* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1993.

(Tchounikine, 93b) A. Tchounikine - *Un outil logique pour l'introduction de la composante active dans une base de données orientée objet* - XIe Congrès INFORSID, Lille, France, pp 161-180, 1993.

(Theodoulidis, 94) B. Theodoulidis, A. Ait-Braham, G. Karvelis - *The ORES Temporal DBMS and the ERT-SQL Query Language* - 5th Int. Conf. on Database and Expert Systems Applications (DEXA'94), Athens, Greece, LNCS N° 856, pp 270-279, 1994.

(Thomazeau, 93) J. Thomazeau - *Une interface multimodale pour l'interrogation d'une base d'objets complexes* - Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1993.

(Tichy, 89) W. F. Tichy - *Software configuration management : tutorial notes nr. 3* - 11th Int. Conf. on Software Engineering, Pittsburgh, Pennsylvania, 1989.

(Urtado, 96) C. Urtado, C. Oussalah - *Propagation de versions dans les objets complexes* - XIVe Congrès INFORSID, Bordeaux, France, pp 331-350, 1996.

(Wuu, 92) G. Wuu, U. Dayal - *A Uniform Model for Temporal Object-Oriented Databases* - 8th Int. Conf. on Data Engineering (ICDE'92), pp 584-593, 1992.

(Zdonik, 86) S. B. Zdonik - *Version management in an Object Oriented Database* - Int. Workshop on Advanced programming Environments, Trondheim, Norway, LNCS N° 244, 1986.

(Zicari, 91) Roberto Zicari - *A Framework for Schema Updates In An Object-Oriented Database System* - 7th Int. Conf. on Data Engineering (ICDE'91), Kobe, Japan, pp 2-13, April 1991.

Résumé

Cette thèse s'inscrit dans le domaine des bases de données orientées objet ; elle propose des solutions pour décrire et manipuler des bases de données intégrant des versions.

Le **concept de version** est nécessaire dans de nombreux domaines d'application comme la gestion de documentations techniques, la conception assistée par ordinateur et le génie logiciel. Les versions permettent notamment de conserver et manipuler l'évolution des entités du monde réel gérées dans de tels domaines.

Différentes gestions de versions sont possibles. Certains travaux gèrent des versions de base ou d'une partie de base pour décrire l'évolution globale d'une base de données ; notre étude s'intéresse, quant à elle, à la représentation de l'évolution de chaque entité décrite dans la base, de manière indépendante.

Nous proposons, d'une part, un **modèle conceptuel** intégrant la gestion de versions d'objets et de classes. Les relations de composition et d'association, dont la sémantique est affinée à l'aide de cardinalités, intègrent les versions pour des entités complexes. De telles relations, incluant les versions, induisent des contraintes d'intégrité structurelle complexes, dont nous faisons l'étude.

D'autre part, nous proposons un **langage** pour manipuler ce type de bases de données. Ce langage permet notamment une interrogation de type Select From Where qui prend en compte les spécificités liées aux versions ; les différents niveaux d'abstraction liés aux versions c'est-à-dire les forêts de dérivation, les arbres et les versions, peuvent être exploités lors d'une interrogation.

Une réalisation du modèle et du langage est effectuée au sein d'un prototype. Ce prototype est une **interface** destinée à des utilisateurs occasionnels, en permettant de manipuler graphiquement une base de données intégrant des versions.

Mots-clés

Gestion de versions - Bases de données orientées objet - Modèle conceptuel - Contraintes d'intégrité structurelle - Langage - Interface graphique - SQL objets et versions

Abstract

This thesis concerns object oriented databases; it proposes solutions to model and manage databases integrating versions.

The **concept of version** is needed in various application fields such as technical documentation management, computer aided design and software engineering. Versions permit notably to keep and manage the evolution of the real world entities handled in such fields.

There are different ways for versioning. Some works chose to describe the global evolution of a database; they manage versions of the whole database or versions of a database subpart. Our study focuses on representing independently the evolution of each entity described in the database.

On the one hand, we propose a **conceptual model** extended to the versioning of objects and classes. Composition and relationship links, whose semantics are refined by cardinalities, integrate versioning for complex entities. Such links, including versions, induce complex contraints for structural integrity.

On the other hand, we propose a **language** to manage this kind of databases. Particularly, this language provide a SelectFromWhere-type querying which take into account the specificities of versions; a query can take the most of the different abstraction levels related to versions that is to say derivation forests, trees and versions.

The model and the language are realized within a prototype. This prototype is an end-user **interface** which provides a graphical management of databases integrating versions.

Keywords

Versioning - Object-oriented databases - Conceptual model - Structural integrity - Language - Version and object SQL - Graphical interface